AN ALGEBRAIC ANALYSIS OF STORAGE FRAGMENTATION

TERRY BETTERIDGE

Ph. D. Thesis September 1979

University of Newcastle upon Tyne

ABSTRACT

Storage fragmentation, the splitting of available computer memory space into separate gaps by allocations and deallocations of various sized blocks with consequent loss of utilisation due to reduced ability to satisfy requests, has proved difficult to analyse. Most previous studies rely on simulation, and nearly all of the few published analyses that do not, simplify the combinatorial complexity that arises by some averaging assumption.

After a survey of these results, an exact analytical approach to the study of storage allocation and fragmentation is presented. A model of an allocation scheme of a kind common in many computing systems is described. Requests from a saturated first come first served queue for varying amounts of contiguous storage are satisfied as soon as sufficient space becomes available in a storage memory of fixed total size. A placement algorithm decides which free locations to allocate if a choice is possible. After a variable time, allocated requests are completed and their occupied storage is freed again. In general, the available space becomes fragmented because allocated requests are not relocated or moved around in storage.

The model's behaviour and in particular the storage utilisation are studied under conditions in which the model is a finite homogeneous Markov chain. The algebraic structure of its sparse transition matrix is discovered to have a striking recursive pattern, allowing the steady state equation to be simplified considerably and unexpectedly to a simple and direct statement of the effect of the choice of placement algorithm on the steady state. Possible developments and uses of this simplified analysis are indicated, and some investigated. The exact probabilistic behaviour of models of relatively small memory sizes is computed, and different placement algorithms are compared with each other and with the analytic results which are derived for the corresponding model in which relocation is allowed.

ACKNOWLEDGEMENTS

I owe much to Brian Randell, who supervised this work, not least for his thoughtful and ever correct guidance and advice. Besides giving many hours of his own time he provided many opportunities for buttonholing any visiting colleague who might have been in a position to consider and offer advice on my problems. His unfailingly enthusiastic encouragement has never been less than cheerfully given, although on some occasions it was expressed in only a single word; I hope he finds that at least some of the "Numbers" contained here partly satisfy him.

Brian is also responsible for introducing and encouraging me to open this Pandora's box of storage fragmentation puzzles, each one big enough to be climbed up and walked about upon. I must therefore thank him for having caused me to spend many hours in such wandering whilst wondering how on earth I could find a way into any of them. But he did not tell me that, like Alice, I would eventually fall down a rabbit hole and discover that the algebraic labyrinth of warrens inside which connects all the puzzles together is almost half as curious as her Wonderland, or that like her, I would then find that the adventure becomes (exponentially) "curiouser and curiouser". I am grateful for this lack of warning, as I am for all his other wise guidance.

I am equally grateful to Ewan Page and Elizabeth Barraclough for originally creating and giving me the opportunity of doing this work under the Staff Ph.D. regulations of this University. Their sustained support that I should spend some part of my time in this way has been essential to me, and I am thankful that they always gave it freely.

I have been helped in numerous ways by very many of my present and former colleagues in the Computing Laboratory, who have discussed storage fragmentation problems with me. They are so many that I hope they will forgive me for not naming them all. They have provided a lot of ideas, information and advice, and it is a pleasure to be able to thank them here for their invaluable help and encouragement.

I owe a special debt to Brian, Ewan, and to Peter King for closely reading and criticising various draft versions of this thesis. They missed little and added much, and their critical remarks were always constructive and taught me a great deal. I hope they will find that the present version is, as I believe, very much improved as a result.

To Elizabeth, and to Judy Hunter and my other colleagues who provided the opportunity and initiated me into the mysteries of the processes by which this thesis was produced, I give thanks. I appreciate and value very much their friendly and helpful spirit of cooperation.

None of this would have been possible without the loving support of my wife Mary. Her saintly patience in putting up with so many hours of my time spent on this thesis has been matched by her interest, enthusiasm and hope for its progress which she has shared with me. I could wish for no more happiness for her from this work than I myself have from knowing that any achievement which may be represented in the following pages is consequently her achievement also.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 : Introduction

Storage fragmentation is the term used to describe the splitting of the space available for allocating requests for computer storage, or memory, into separate non-contiguous areas inbetween blocks already allocated to requests which cannot be moved around in the storage once they have been allocated. This thesis describes and analyses a model of a storage allocation scheme of a kind which is common in many computing systems. A storage, or memory, consists of a fixed integer amount N of contiguous "words", numbered from 1 to N say. A word cannot be split for the purpose of allocating storage. The unit of allocation in a practical application might be a word, a page, or some other fixed size unit of memory, and "word" is used here to include all these terms. Requests from a saturated or never empty queue for varying integer-sized blocks of contiguous words are satisfied as soon as sufficient space becomes available. An allocation placement algorithm decides which available, or free, locations to allocate if a choice is possible. Blocks remain allocated for a variable time, after which the requests are completed and their occupied storage is freed again. In general, the available space becomes fragmented because allocated requests are not relocated or moved around in storage.

1.1 The origin and purpose of this investigation, and its significance

The original purpose of this investigation was to provide if possible a theoretical analysis of storage allocation and fragmentation, sufficient to predict in advance such quantities as the likely storage utilisation (the fraction of memory allocated to requests) and amount of fragmentation that can be expected to occur, given such necessary parameters as the total storage size, the method of deciding where to choose to put new requests, some information about the expected duration of allocated requests in storage and the probability distribution of the size of a new request. A particular stimulus to this investigation was given by Randell (1969), who observed some unexpected fragmentation behaviour when performing simulations of a storage allocation scheme. He noticed that the attempt to reduce fragmentation and increase utilisation by rounding up each storage request size to the next nearest multiple of a fixed quantum before allocating the request, was in practice more than offset by the consequent loss of utilisation caused by the rounding up process. Increasing the quantum size appeared to make this net loss steadily worse rather than better. Randell found this observed result to be intuitively surprising, and called for analytical confirmation to discover its cause and the region of its validity.

Storage fragmentation has proved to be quite difficult to analyse. The available theoretical analysis of such a general storage allocation system has, at least until very recently, in Randell's words been "sorely lacking". Most previous studies have relied on simulation, and most of the few that do not, have simplified the combinatorial complexity that arises by at least one approximating assumption about the average behaviour of a large number of possible cases, even after having already made such mathematically tractable assumptions as a uniform or negative exponential distribution of request size.

The analysis presented in this thesis has similarly had to make some initial simplifying assumptions for tractability, in particular that request sizes are identically and constantly distributed independent of each other, that the distributions of request size and duration (amount of time spent in the memory) are independent and that the requested duration in memory is in fact distributed such that at any moment any allocated block in storage is equally likely to be the next to be deallocated. The last requirement can be met for instance by assuming a negative exponential distribution of request duration. The plausibility of these assumptions is argued in chapter 3, where they are in fact shown to be sufficient, with suitable definitions of state and transition between states, for the model to be considered as a finite discrete-time Markov chain. When this chain is ergodic, as in general it is, its equation of steady state has a unique eigensolution which is the dominant eigenvector of steady state probabilities.

Unlike most previous studies, the subsequent analysis is then exact as it makes no further approximating or averaging assumptions and also it does allow any general request size distribution to be assumed. The apparently inevitable combinatorial explosion which results for increasing total memory size is managed by noticing that certain aspects of the model constantly recur whatever the total size of memory may be, and by studying the properties of the model which cause this to be so.

The storage utilisation in the steady state is usually of interest in any practical storage allocation scheme in which space is either expensive or at a premium. Properly defined, the expected amounts of storage utilisation and fragmentation (wasted space due to rounding as well as space unused because it is scattered between the blocks) can be calculated directly as linear functions of a steady state eigenvector of probabilities. Consequently it is interesting to see how much information about this equilibrium eigenvector can be discovered from the knowledge of the structure of the transition matrix, which in turn depends in part upon the allocation algorithm and request size probability distribution. There is thus a direct link between these last two and quantities such as the expected utilisation and fragmentation, and it is one achievement of the present work to discover that there is a short, simple and direct algebraic statement of this link, the reduced steady state equations.

The transition probability matrix of the Markov chain is in fact discovered to have an algebraic expansion as sums and products of much simpler matrices of probabilities, each of which represent a single momentary stage in a complete transition. These simpler matrices have a striking recursive structure which can partly be seen as visual patterns, when the states are ordered in ways which are related to the properties of the model which are responsible for this structure. These simple component matrices can still be specified exactly in the general case of arbitrarily large memory size by recurrence relations which express their form in terms of smaller versions of themselves and each other. This algebraic expansion of the transition matrix allows the equation of steady state to be rearranged and substantially simplified to a much simpler form in which the effect of the choice of placement algorithm and request size distribution on the eigenvector of steady state probabilities is stated quite directly.

This reduced equation and the algebraic expansion of the transition matrix, both expressed in terms of simple matrices with recursive structures, can be used as starting points for discovering the model's behaviour. As an example of the possible uses of the reduced steady state equations, the constraints which must be satisfied regardless of the allocation algorithm are obtained by the discovery that groups of states may be defined to ignore any differentiation which the choice of allocation algorithm may introduce. When this is done the steady state equations reduce further to a statement of the constraints which the modified vector of grouped steady state probabilities must satisfy irrespective of the allocation algorithm actually used. Both the expected utilisation and fragmentation are still expressible as linear functions of the grouped steady state probabilities. Another rather different example of the use of the structure of the transition matrix occurs in an implementation of the power method. This is used to gather whatever exact numerical results may be possible for memory sizes up to as large as can be managed, by using a knowledge of the structure to avoid storing the transition matrix and thereby allowing larger models to be computed.

The rest of this chapter expands on the above outline by summarising the work of each subsequent chapter according to the order in which each is arranged.

1.2 Relation of the present analysis to existing work

Chapter 2 summarises the existing work that has been done on this kind of storage allocation problem, and describes different features of the storage allocation schemes that have been studied. It also surveys possible connections with some other related areas of study such as random space filling problems in which the fragmentation of the available space is of interest. With the few exceptions that are noted, there is not much useful mathematical analysis of this general kind of problem, which perhaps indicates that solutions are hard to find. Most authors who have tried to produce positive predictions about particular allocation systems have used simulation to estimate model behaviour and performance. The mathematical analysis that has been done is mainly either quite short and not in general very precise, or else it is longer and more detailed, and the conclusions harder to reach. Knuth's (1968) well known fifty per cent rule, for example, makes some assumptions about averages and so produces a conclusion which is an approximation in general. The more detailed analyses which make less assumptions about averages, are longer and have more difficulty in carrying the mathematical complexities which arise to a useful conclusion. A good example is Purdom and Stigler's (1970) analysis of the buddy system. By assuming Poisson arrivals and exponential service times they were able to estimate the average number of available blocks at the smallest level of size, and to estimate the approximate relationships between block lifetimes on adjacent pairs of level sizes, but were unable to combine these relationships for the model as a whole except by considering each pair of levels independently. The analysis presented in this thesis is of this second type, for if nothing else it is certainly more detailed, longer and more complicated than the fifty per cent rule, for example. It provides a particular mathematical foundation for analysing the problem, and part of its contribution is that one aspect at least of the analysis unexpectedly simplifies to a sufficient degree that it is not unreasonably optimistic to expect useful results will be possible besides those presented here.

1.3 Analysis based on a Markov model of storage allocation

The model of storage allocation to be used as a vehicle for study in this thesis is defined in chapter 3. Certain assumptions about the distributions of request size and duration in storage are made, and definitions of state and transition between states are given, with which the model is proved to be a finite homogeneous Markov chain. These assumptions are not unreasonable for a wide range of situations in which storage allocation and fragmentation are of interest, and some arguments to support this are presented. Although unusual circumstances are shown to be possible in which the Markov chain is not so well behaved (it is not always irreducible), it is proved to be ergodic in general. The transition probability matrix then has a unique eigenvector of equilibrium or steady state probabilities to which the time-dependent vector of state probabilities always converges as time passes whatever its starting value at some initial time origin.

1.4 Number of states, sparseness, and comparison with relocation

Chapter 4 contains a preliminary analysis of this particular storage allocation model just defined, and introduces some more terms and notation needed in the following chapters. The number of states in the Markov chain is shown to increase exponentially with memory size and in fact it takes on values which are every second term from the Fibonacci sequence. The matrix of transition probabilities is seen to be sparse, as on average comparatively few other states can be reached from any given state in one transition.

An easier to analyse variation of the model in which relocation is allowed, and which is interesting because it provides an upper bound to the storage utilisation in the non-relocating model, is considered. Randell (1969) also used this relocating variation to compare the performance obtained with different allocation algorithms and request distributions. An example of this relocating variation of the model is presented and closed form expressions are obtained for the expected equilibrium values of the utilisation and fragmentation.

1.5 Algebraic expansion, simplification and use of the steady state equation

The heart of the work in this thesis is in chapter 5. Because this material is new and therefore more unfamiliar than that of the (necessarily partly introductory) preceding two chapters, the exposition and proofs here are a little longer in style than in chapters 3 and 4. This extra care may seem in places, perhaps especially in the first section, to be longwinded or to be proving the obvious. It has been taken nevertheless because of the importance of what is contained here to the rest of the work in the remaining chapters. Section 5.1 reveals the recursive structure of the transition probability matrix of the Markov chain. This structure is used in section 5.2 to transform the steady state transition equation to the reduced form already mentioned above, which represents the choice of allocation algorithm in a very simple and direct way. The properties of this simple form and the prospects for its use are examined in section 5.3. The example of this use which is given is to produce by a further reduction the allocation-independent constraints already referred to. It was discovered that these could be achieved by grouping the states, guided by the choices indicated by the matrices which represent the action of the allocation algorithm. The rules for constructing these new state groups and the matrix of the constraining equations without prior reference to the transition matrix, are set out in detail.

This reduced set of equations contains some degrees of freedom as there are fewer constraints than grouped states, and so it is possible for example to consider using linear programming techniques to discover how utilisation and fragmentation may vary when the allocation algorithm is chosen with complete freedom. Interestingly, there are many fewer degrees of freedom in these further reduced equations governing the grouped state probabilities than there appear to be in the allocation algorithm specification, implying that many of the choices possible when selecting an algorithm are redundant or immaterial as far as utilisation and fragmentation are concerned.

1.6 Exact numerical results for small memories, and their usefulness

Chapter 6 presents the attempts based on the power method which have been made to calculate exact numerical solutions for models of small memory sizes, and the results which have been obtained. The means used to obtain results for as large a memory size as possible are explained. The most successful of these which allows the largest possible size so far achieved to be computed (12 words), uses the sparse structure of the transition probability matrix set out in chapter 5 to avoid storing the matrix anywhere at all. The results from three allocation algorithms and two request distributions are compared with each other and with the corresponding upper bound results for the relocating model from chapter 4. Even such small memory sizes can produce useful results, not least because the unit of allocation in a practical situation might be large, a page or a disc track for instance, so that the number of "words" in such a memory might not be very great. As an example of the possible usefulness of these results even for these small memory sizes, Randell's observations (above) on the effect on storage utilisation of increasing the quantum size when requests are being rounded up before allocation, are successfully demonstrated and corroborated.

1.7 The beginnings of further development and use of the analysis so far

The development of an algebraic means of analysing storage allocation and fragmentation to the state described in the preceding chapters has the happy consequence that there are now a number of different ways in which to apply the analysis and to develop it further. Before the conclusions in chapter 8 these possibilities are examined in chapter 7, and most of them give indications that more interesting results may well be obtained to add to those already described here. It is possible, although perhaps not very likely as the first section in this chapter indicates, that a better method than the straightforward iterative power method may be found to determine the steady state probability eigenvector. Certainly it seems that more use might be made of the exact converged values obtained for small memory sizes.

There is undoubtedly more to be discovered in the way in which the recursive structure of the transition matrix determines that of the steady state equations, and further study should lead to greater insight into the properties of these equations and their solution. Not the least interesting possibility is of applying the presently developed analysis to other similar models of storage allocation. There are a number of ways in which the saturated queue model defined and studied in detail in the earlier chapters can be modified to match the variety of allocation schemes which exist in practice. Chapter 7 surveys some of these variations. The algebraic structure of the transition matrix simplifies quite a lot when the queue of requests is specified as being unsaturated, as this chapter shows, and since unsaturated storage allocation models have occurred more often in previously published work it will be interesting to see how the analytical methods presented here can be applied to this model and to compare any results obtained.

1.8 Previous work which has been assimilated into this thesis

The author has previously written a number of papers (1971, 1973, 1973a, 1974, 1974a, 1977) as the analysis of the storage allocation problem has been developed. Almost all of their contents appear and have been extended in this thesis in some form, although the converse is most certainly not true; there is much extra work reported here which has not previously been published anywhere else. The (1971) report was concerned with the analysis of what happens when relocation is allowed, and its results are incorporated and extended in section 4.4 of the present work. The (1973) report which was subsequently published with a few additions (1974) was concerned with the definition of the model, some of its properties as a Markov chain, and the first implementation to obtain numerical results. It is covered by chapters 3 and 4, and parts of chapter 6. The (1973a) report began the algebraic analysis of chapter 5, and this was subsequently extended (1977). The (1974a) report documents the author's failure to achieve much progress with an extension of the model definition to infinite memory sizes, summarised here in chapter 7 section 7.3.

Chapter 2 : A survey of previous work

Before attempting an analysis of storage fragmentation it is of course necessary to study the work of others who have considered the same or a similar problem. This is partly to avoid needless duplication of effort, but mainly because it is wise to judge what progress has already been made, what might possibly be done that would be useful, and how it could be achieved by using existing results where possible and appropriate. The purpose of the present chapter is to make such a survey of the previous efforts that have been made to analyse the generally difficult problems of storage fragmentation.

Storage fragmentation occurs as one aspect of a much more general class of problems which are concerned with how to allocate or divide up some kind of fixed storage space among a number of requests for it, often with the aim of doing this as well as possible with respect to some measure of efficiency. Usually in this kind of problem there are some rules or constraints as to how the allocation should or must be performed. The most common of these is exclusiveness, that the space reserved for the use of one request can not be used for another until the first has finished with it. Also very frequent is the requirement that the space allocated to one request should not be scattered but instead has to be compact or contiguous, occurring all together in one piece in some sense. When there is also variability of the size (and possibly the shape) of the requested space and of the time for which it may be required, and when the space that has been allocated to a request is not easily moved around, these are the general conditions in which the available space will be split up or fragmented into separate pieces. Since in these conditions the fragmented space is generally less able to satisfy requests than it would be if it could be all collected together, the question arises of how best to allocate or give out the space to minimise this effect.

The allocation of requests for space for storing programs and data in computer memories is an obvious example, and many aspects of computer memory allocation have been studied since the earliest days of computing. Knuth (1968), pages 456 - 461 gives a survey quoting many developments from 1946 onwards, and Randell and Kuehner (1968) also give a similar explanatory survey. From both these sources it is clear that the term "storage allocation" was sometimes used in the early days of computing to mean much more than it usually does now. Indeed in a few places where it was used a more appropriate present-day term would now be "operating system"; see the description by Maher (1961) of storage allocation in the Burroughs B5000 computer, for example. (The storage allocation routine loads programs, assigns memory, assigns I/O buffers (with semi-automatic tanking?!), assigns I/O units, configures the system when the B5000 is switched on, protects storage and relocates address constants.)

With such a general problem it is not surprising to find that storage fragmentation effects have been studied in a variety of other guises. Most notably these include space-filling problems, the practical origin of which include in their range of scope a desire to know the best shape for a lump of coke so as to get the most fuel into a given volume such as a railway wagon, Renyi (1958) below, to a requirement (Page (1959), also see below) to estimate the likely amount of hydrogen deposited in molecular doublets as a thin film on a rectangular lattice surface, and the amount subsequently adsorbed from it by mercury. Because of the difficulty of the analysis both Renyi and Page managed only to consider the one-dimensional case instead of the original three- and two-dimensional problems. In such a general sense of course the problem is an everyday one, familiar for example to the housewife arranging groceries on the pantry shelf or the building site foreman deciding where to store steelwork temporarily on a building site. (An interesting example of this last problem is given by Lovell (1968).) This thesis is concerned mainly with expressing the problem in terms of the application to computer memory management however, and this will be clear from the next and subsequent sections of this chapter.

2.1 External and internal fragmentation, and Randell's observation

Randell (1969) published the results of some simulation experiments which were intended to investigate the various factors which might affect storage fragmentation, in a paper the importance of which can partly be judged by the regularity with which it has been referenced by subsequent authors. He distinguished and defined two kinds of fragmentation. In the present thesis, the external fragmentation EF of a storage memory at any particular instant, containing available or free storage possibly separated by allocated or unavailable blocks, is here defined to be the fraction (expressed as a ratio between 0 and 1) of the whole memory which is available at that instant. The complementary quantity storage utilisation is defined as the fraction of the whole memory which is allocated, so that always

$$\text{external fragmentation EF} = 1 - \text{storage utilisation} \qquad \ldots.\ 2.1$$

This is the definition of external fragmentation, as a loss in storage utilisation caused by the separation of the available space, which has been used by nearly all authors most of whom have referred to it as being due to Randell. In fact the definition he gave, which is harder to work with directly, is slightly different. He defined external fragmentation as the difference in storage utilisation between two experiments, one using a non-relocating placement algorithm and an otherwise identical experiment in which allocated blocks were always moved (relocated) so as to collect together the free space whenever necessary. This difference can be derived from the definition given above if the storage utilisation in the relocating model is known. Randell's definition is perhaps more correct as it recognises that even when blocks can be moved as required, the last allocated block will not in general be a perfect fit. In fact in most practical situations where block sizes are generally small compared to the total size of memory, the difference is slight since storage utilisation in a relocation scheme will be close to the maximum if the queue of requests is saturated. Randell concentrated in his simulations on the effect of rounding up requests for storage to the nearest multiple of a given allocation quantum, in order to assess the effect of reducing the number of different sizes of blocks coexisting in storage. This introduces a second kind of loss of storage utilisation, which he called internal fragmentation, defined to be at any instant the fraction of total memory within the allocated blocks wasted by the rounding process. Some subsequent authors have defined this term to be the ratio of the space wasted by rounding to the total allocated space, but Randell's original definition will be adopted here as it is simpler to use. The alternative definition is obviously given by the expression IF/(1-EF), where IF is the internal fragmentation as defined by Randell and EF is defined in equation 2.1 above. The difference in numerical value will not be slight if the unallocated space fraction EF is significant.

Clearly in a storage allocation system in which both external and internal fragmentation are occurring, the fraction of memory actually "in use" at any moment by being allocated to the original requests before rounding, is given by proper storage utilisation (with rounding) = (1 - EF) - IF 2.2

so that the term "storage utilisation" defined above as the space allocated to blocks, and therefore including the wasted space due to rounding, is misleading when request sizes are being rounded up before allocation is being performed. In fact internal fragmentation is only considered explicitly at two specific places (sections 4.4 and 6.5.3) below, since most of this thesis is concerned with the more difficult task of examining the behaviour which produces external fragmentation, which logically has to be considered first anyway. It is therefore convenient to keep the term "utilisation" as defined, the fraction of memory allocated to blocks as though no rounding is taking place, and to remember to subtract the term for internal fragmentation when the amount of rounding up is subsequently brought in to the argument.

To illustrate what was discovered as the allocation quantum was increased, one of the diagrams from Randell's paper is reproduced in chapter 6 as figure 6.8. As mentioned in the introduction to chapter 1, all of the simulations which he reported showed the unexpected result that as the allocation quantum increased the loss of utilisation due to increased internal fragmentation distinctly outweighed the gain due to decreased external fragmentation. The proper or combined loss went up rather than down as a result, and Randell called for an analytical investigation to confirm this. The present thesis, the contained results and any subsequent work exist largely as a consequence of this appeal although the scope of the investigation has widened considerably beyond an attempt to answer this direct question alone.

2.2 Some different possible allocation algorithms

Various methods have been used to decide where to allocate a new request for storage in a given memory configuration when a choice is possible. Perhaps the best known of these are first fit and best fit, described by Knuth (1968), but certainly used long before then. Collins (1961) for example reports playing experimental "games" to compare the first, best, worst and random fit algorithms, but he gave no results other than to say that best fit lasted longer before overflow occurred than the others. It seems likely that the first and best fit algorithms were probably among the first placement rules to be used in storage allocation schemes in which fragmentation could occur.

2.2.1 First and best fit, and some variations

Consider the words of memory to be ordered from 1 to N where there are N words in the memory, or "left to right" in a pictorial sense. Then first fit searches from left to right, and allocates the request at the left end of the first gap encountered which is large enough to receive it. Obvious modifications to this rule are to search from right to left, or to begin the search at the point where it stopped last time (this last algorithm is known as modified first fit). Best fit allocates the request into the smallest gap which is sufficiently large enough to receive it, not necessarily the first encountered. If there is more than one such gap of this smallest size, then one of them is chosen by some rule such as the leftmost or first encountered in a left to right search. Figure 3.1 at the beginning of the next chapter for instance illustrates an example of both of these algorithms in operation, and shows a situation in which they will perform different actions to each other. Knuth performed a number of experiments to compare first fit and best fit, and commented that in general first fit was better since best fit tended to produce more smaller gaps than first fit.

However; Shore (1975) has also compared these two algorithms and has a different explanation of the comparison, that first fit performs better than best fit because small blocks and gaps tend to congregate at the beginning or left hand end of memory, and larger blocks and gaps at the other end, so that first fit is more likely to be able to satisfy a large request. Shore found that best fit was generally better than first fit for the uniform request size distribution, and that although the comparison was reversed for the exponential distribution, as the maximum request size was reduced in his experiments the difference became less until best fit performed better than first fit for the exponential distribution as well. Shore pointed out that the relative performance depends strongly on the request distribution. He reported that, as Randell (1969) had also discovered, the variation in utilisation caused by changing the allocation algorithm did not often cause a difference of more than one or two per cent, whereas the change in request distribution for the same algorithm can produce differences of five or sometimes ten per cent in the utilisation. Shore proposed a combined algorithm, "best then first" fit, in which memory is divided into two regions. The algorithm tries best fit in the first or left hand region, and then only if necessary, first fit in the right hand region. He argued that if the boundary between the two regions could be correctly adjusted, this might combine the advantages of both algorithms.

To compare with the results of both these authors, it will be seen in section 6.5.3 in chapter 6 below that when the queue of requests is saturated, best fit always performs as well as or better than first fit for both these request size distributions, at least for the memory sizes which have been computed using the model described in that chapter. Knuth used a model in which the request queue was not saturated, that is, whenever a request arrived it could be immediately satisfied, and not to have the space available for it (overflow) was considered as a failure. Storage utilisation in such a system is certain to be lower, and less relevant anyway without saturation; his comparison of methods was apparently based more upon how long each algorithm would last before an overflow failure occurred. Shore's model did use a saturated (essentially infinite) queue of requests.

2.2.2 Fenton and Payne's half fit algorithm

Fenton and Payne (1974) also compared first and best fit by performing a number of simulation experiments, and found that best fit invariably performed better for various request size distributions, including uniform and exponential. Their results which they reported in a different way to previous authors, also appear to show that the difference in utilisation between these two algorithms is usually not more than two per cent. They also reported results on the modified first fit algorithm, and the half fit allocation algorithm (see below). Modified first fit consistently performed worse than first fit, generally up to 5 per cent lower utilisation, supporting Shore's remarks on the separation of small and large gaps which is not expected to occur in modified first fit. That modified first fit performs worse than first fit is also supported by the simulations reported in a short note by Bays (1977) who compared first fit, best fit and modified first fit finding a result consistent with Shore, and Fenton and Payne.

Fenton and Payne's half fit algorithm allocates a request into a gap which is approximately twice its size, and falls back upon using best fit if this is not possible. The idea is to deal more successfully with a distribution dominated by one size of request. If the next request is likely to be for the same size as the current one, then it might be a good idea to find and use a gap of twice the needed size if possible rather than, say, one which is too big by only half as much, (and to use best fit otherwise, to use up the already created gaps of the right size). They found that this idea was only moderately successful in practice, "half fit performed well in some cases but the results were too erratic to draw any conclusion", and gave the same figure as that for first fit for its average performance over a number of simulations.

2.2.3 Campbell's optimal fit algorithm; the optimal stopping problem

Besides Shore's "best then first" fit, another algorithm which combines first and best fit is the optimal fit method proposed by Campbell (1971). This is based on an analysis of the problem of optimal choice, or optimal stopping problem, which has been analysed by Dynkin and Yushkevich (1969) using Markov theory. Campbell quotes an explanation of the problem in terms of a cyclist wishing to stop overnight at one of a known number G of hotels, to determine the optimal strategy for finding the best hotel in the absence of any other advance information if a strong following wind prevents him from going back to any hotel once he has left it for the next. (Although it is less practical, Dynkin and Yushkevich's charming alternative example of an aspiring but astute bride-to-be wishing to make an unerring choice of the best of all suitors proposing marriage to her is more amusing.) To allocate any request, the optimal fit algorithm first computes an integer k(G) as a function of the number G of free gaps currently in the memory. The gaps are then scanned from left to right. The best possible fit in the first k(G)-1 is noted, but allocation is not performed. The algorithm then chooses the first gap numbered from k(G) onwards which has a better fit than all the previous gaps so far encountered. If there is no such gap, the best fitting gap is used (the one that the best fit algorithm would have chosen, and which must therefore be in the first k(G)-1 gaps; the cyclist and the bride-to-be do not have this option). For large G, a good approximation to k(G) is to choose the rounded up value of G/e = G/2.718 approximately. Campbell implemented the optimal and first fit algorithms in a particular application program and compared their performance as measured by the number of garbage collections required, finding that optimal fit was generally better, keeping the free storage list of gaps in the memory noticeably more compact.

2.2.4 Random and worst fit

Two more possible ways of deciding how to choose where to place a new request in storage which have received less attention in the literature than best or worst fit, are the so-called random fit and worst fit allocation algorithms. Given a request to be allocated, random fit chooses at random from all the available gaps sufficiently large to receive it. Reeves (1979, 1980) has used the random fit algorithm to successfully demonstrate how storage allocation systems may be analysed algebraically by means of generating functions, see section 2.6 below. Randell (1969) included it as one of the algorithms he compared in his simulations, as he thought it likely to be near the opposite extreme in performance to best fit. In fact, it is possible to be quite a lot more perverse than this, for after all a random choice will sometimes be made "correctly" as well as sometimes badly. With this idea in mind, the present author has devised and used a "worst fit" algorithm for the computations of chapter 6 in an attempt to discover how much the storage utilisation can be affected by "bad" allocation. One feature of this particular algorithm does not seem to have been described elsewhere. Given a request to be allocated in a particular configuration of memory, the worst fit algorithm allocates the request into the middle of the largest available gap, leaving an amount of space on either side of the newly formed block in two new gaps which are either equal in size or else different by only one word. The predicted results of using this algorithm appear and are discussed in section 6.5.

This version of the worst fit algorithm thus introduces an idea which, while it is not new, has not been discussed or analysed in this context elsewhere, that a request need not be allocated at the left or right hand end of the chosen gap but somewhere in the middle of it. This does not guarantee to do worse than putting the request at either end, although intuition again suggests that it usually should. To see why worse performance is not automatically guaranteed, consider figure 2.1. This shows a 5-word memory in which the first and last words are initially each allocated to blocks A and B each of length 1 word, and words 2, 3, 4 are empty. The queue requests which will arrive to be allocated are shown as C (1 word), D and E (each 2 words). If request C of 1 word is allocated, and then blocks A and B are deallocated, worst fit leaves the available space in two equal halves of 2 words each so that blocks D and E can subsequently be fitted in, whereas best fit divides it into 1 word and 3 words so that only D can be allocated. In this example, best fit would of course do better if D was a request for 3 words and E for 1 word. It should be pointed out that this idea of leaving a space on either side is inherent in an alternative meaning which can be attached to the word "random" to that intended by either Reeves or Randell. This is that instead of choosing equally one gap from each of the gaps sufficiently large for a

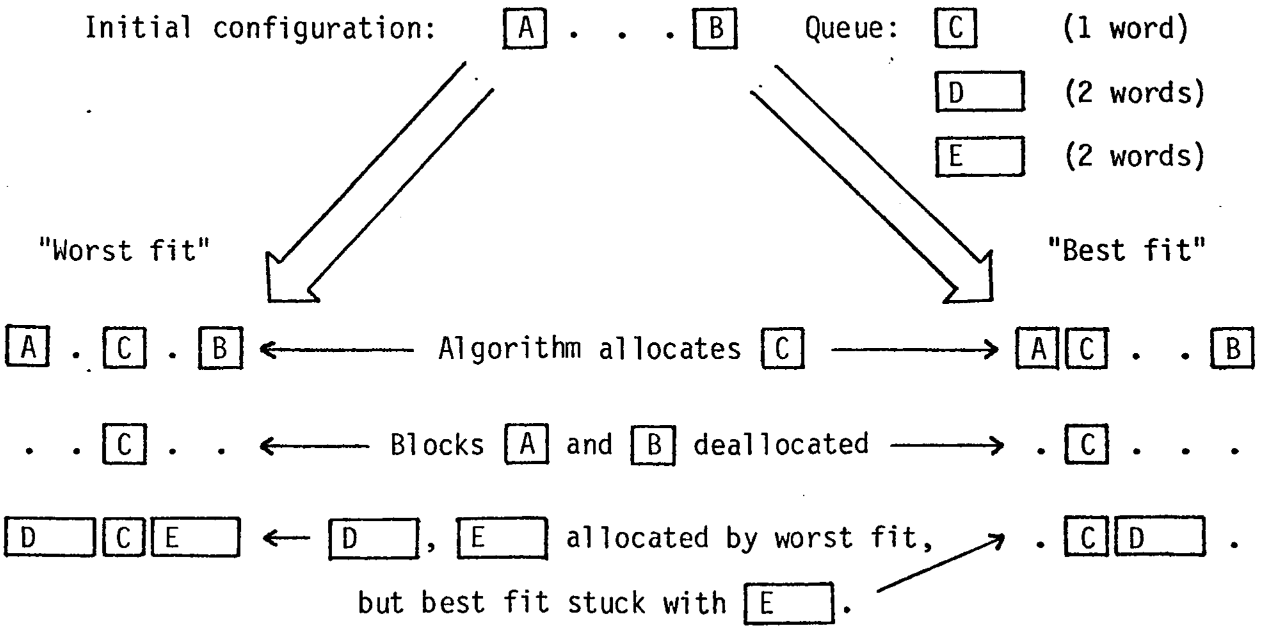

Initial configuration:
A . . . B
Queue: C (1 word)
D (2 words)
E (2 words)
"Worst fit"
"Best fit"
A . C . B ← Algorithm allocates C → A C . . B
. . C . . ← Blocks A and B deallocated → . C . . .
D C E ← D, E allocated by worst fit,
but best fit stuck with E.
. C D .

Figure 2.1 An example to show how "worst fit" can be better than "best fit"

given request, the choice can be equally at random from any of the possible positions in the memory in which the new block will not overlap any existing blocks. This therefore allows requests to be allocated in the middle of gaps as well as at their ends, and weights the choice towards the larger gaps. It is like trying to lay a new brick at random on a partially completed course of bricks whilst blindfolded; attempts which hit already existing bricks are discarded, and there is no guarantee that the successful try will place the brick exactly end up against one that is already laid. This is the idea of randomness which was used by Renyi (1958) in his consideration of space filling problems, see section 2.7 below.

2.2.5 Good and bad allocation; performance bounds, worst case behaviour

Whilst on the subject of inefficient as opposed to efficient performance, it should be mentioned that section 5.3 in chapter 5 below determines the algebraic conditions which must be satisfied by any allocation algorithm, good, bad or otherwise inbetween, in the model which is considered in this thesis from chapters 3 to 6. These conditions form a set of constraints within which the algorithms may cause the utilisation for example to be varied, so that by studying the variation within these constraints it may be possible to determine upper or lower bounds on performance. A different point of view on bad performance is also provided by Robson (section 2.5 below), who is concerned not with inefficient allocation algorithms but with worst possible behaviour in terms of the request sizes and sequences of allocations and deallocations which can occur.

2.2.6 Buddy systems

For no very clear reason these allocation algorithms have generated more interest and had more papers written about them than any other kind of algorithm, if not more than all the others put together. Their distinguishing feature (apart from taking longer to explain than most other algorithms, as can be seen here) is to make explicit and direct use of a tree structure to organise the configurations of allocated blocks and intervening gaps in memory. Consequently they tend to be quicker in operation than most other algorithms, that is they take less time on average to allocate or deallocate a block, but unless a particular request distribution is such that a particular buddy system can be tailored to suit it this is paid for in a higher wastage of space, especially internal fragmentation due to rounding. Most of the practical interest that has been shown is probably because of their speed of operation, in systems where execution time is more critical than storage space, but undoubtedly the opportunity to exercise a little ingenuity in tinkering with the mechanism of the algorithm has also been an attraction to some authors.

A description of the simplest scheme, the binary buddy system, was first published by Knowlton (1965a) although Knuth (1968) says that H. Markowitz independently invented and used it in 1963. Interestingly, Knowlton does not use the word "buddy" in his description. Initially the memory which has size a power of 2, $N = 2^M$ say, is empty. To allocate a request of size n, the requested size is first rounded up to the nearest power of 2, so that $2^{m-1} < n \leq 2^m$. If m=M the request is allocated, otherwise the empty memory of size 2^M is considered to be split into two equal halves, or buddies, of 2^{M-1} words each. One of these, it does not matter which, remains for subsequent requests and the other is again split into two equal halves, size 2^{M-2} and 2^{M-2}; and so on, until the equal halves have size 2^m. One of them is used to satisfy the request and the other remains for subsequent requests. The action of an allocation is similar for subsequent requests when the memory is initially not empty. If a free block (gap) of the correct sized power of 2 is available, it is used to allocate the (rounded up) request, but if not then a larger free block of the smallest size for which there is a free block available is recursively split, as before. The action on a deallocation merely reverses this procedure. If the buddy of a freed block also happens to be free, then the two are recombined into a single free block of twice the size, and its buddy is likewise examined in turn. Recombination stops either when a buddy is reached which is not free or else the originally empty memory of 2^M words in one free block is reached.

The speed of the algorithm lies in the complete absence of any list searching and the relatively small number $M = \log_2 N$ of possible block sizes. Free blocks all of the same size are remembered by means of a last in first out list, or stack, usually implemented by address pointers kept in the free blocks themselves. Thus there is one stack for each possible size 2^m. When a free block of a given size is required, the first one on the list is taken (and the list updated of course). When a block of the same size becomes free it is simply put onto the front of the list. To find the buddy of a given block, if this is not maintained also by a pointer then the very quick method given by Knowlton can be used, "... simply to complement the m-th order bit of the 2^m-block's address. Thus if the address of the beginning of a 2-word block ends in binary ...0110$\underline{0}$0, then the address of its mate ends in ...0110$\underline{1}$0. Similarly if the address of a 4-word block ends in ...010$\underline{1}$00 then the address of its mate ends in ...010$\underline{0}$00". One bit is required in each block to mark it as allocated or free.

Knowlton (1965) also described an implementation of a language compiler using the binary buddy algorithm. The description of the binary buddy system as a tree structure was made more explicit by Knuth (1968), and the word "buddy" was introduced. Since then other variations on the number and sizes of the buddies into which a block can be split have been proposed, more elaborate tricks for finding buddies of blocks and deciding whether or not they are free have accompanied these, and (in keeping with other storage allocation schemes) the results of a number of simulations and just a few preliminary analyses have been published. One of the first of these developments was a technique by Isoda, Goto and Kimura (1971) for reducing the amount of information needing to be kept to describe whether a block is free and what size it is, if this information is kept in a separate table rather than in the first few bits of each block. These authors also avoid the name "buddy", not referencing Knuth. The first proper extension away from powers of 2 to be published was by Hirschberg (1973), who did reference Knuth because he developed an idea which Knuth proposed as an exercise, to have blocks of sizes which are terms from the Fibonacci sequence and to split them according to the fundamental definition of that sequence (equation 4.2 in chapter 4 below). A year later, Shen and Peterson (1974) suggested a scheme which they called the weighted buddy system, providing blocks of sizes 2^m and 3×2^m upon splitting a block of size 2^{m+2}; blocks such as 3×2^m which are multiples of 3 are themselves split to form blocks of size 2^{m+1} and 2^m. The greater range of block sizes which are therefore closer to each other in the Fibonacci and weighted buddy systems tend to reduce the internal fragmentation since in general, and on average, there is less far to go to reach the next rounded up block size.

Cranston and Thomas (1975) removed a very awkward restriction of the Fibonacci buddy scheme, that to avoid the possibly large amount of storage required for pointers to find the buddy of any block, a time-consuming calculation was required to discover whether a block had a buddy on the left or the right instead of Knowlton's simple bit-inverting technique which does this automatically. Their neat solution requires only two extra bits per block, which record the information not only about whether each block is left- or right-handed but also the same information for the resulting recombined block so that upon recombination of the buddies the same information is available to be able to repeat the process. Hinds (1975) also noticed that this could be achieved by keeping a count for each block. For left hand buddy blocks this would be the number of splits that were required to reach the block starting from an initially empty memory, and the count would be zero for right hand buddies. Peterson and Norman (1977) have produced algorithms for implementing buddy systems in which the block sizes can be chosen freely so long as they satisfy recurrence relations of a certain general form used to perform the splitting. These relations allow the possibility of splitting into more than two buddies, and of tailoring the block sizes in this general buddy system to suit a known range of request sizes by an appropriate choice of the recurrence relations. They give an example accommodating sizes 12, 80 and 132 which might be used for control blocks and card and printer record buffers. Another example is that of Burton (1976) who also further developed the generalised Fibonacci relations suggested by Hirschberg in a similar way to Peterson and Norman, and applied it in an implementation of a disc storage allocation scheme in which the sizes 50, 128, 150, 1024 and 10240 occurred.

Concerning the performance and behaviour of the various buddy systems, apart from the descriptions of Knuth's experiences in trying out the binary buddy system the first performance analyses were by Purdom and Stigler (1970), and Purdom, Stigler and Cheam (1971). They were concerned with external fragmentation and running speed, and their analysis concentrated on the interaction between one level of block size and the next. They found that compared with the first fit method, although the buddy system was much faster at allocating and deallocating requests its total fragmentation was higher, quoting about 15 per cent more for the distributions they used, and that of this total the more important element was the internal space wasted due to rounding. (It has already been mentioned in the introduction above that in order to carry out their analysis at all, they assumed Poisson arrival times and exponential service times.) Their comparison agrees with that obtained subsequently by Nielsen (1977). Shen and Peterson found that the request size distribution made quite a difference to the amount of wasted space when comparing the weighted buddy and binary buddy systems, binary buddy being better for a uniform distribution and weighted buddy better for an exponential distribution. Hirschberg compared Fibonacci and binary buddy for a size distribution with an approximately exponential shape, and found that while the Fibonacci buddy system caused more external fragmentation than binary buddy it also had less internal fragmentation. He concluded that the Fibonacci buddy system was better as running times were generally about the same. As for estimating and predicting the amounts of external and internal fragmentation in advance, Bromley (1977) has given a block-counting technique for estimating practically useful upper and lower bounds on external fragmentation given the request distribution and knowing the set of possible block sizes, while Russell (1977) has shown how to investigate the expected internal fragmentation in a generalised buddy system when the largest possible request size is not simply equal to a convenient blocksize of the system.

2.2.7 Summary: comparison of the wide range of possible allocation algorithms

The above list of possibilities form a surprisingly large collection of different rules that have actually been used for deciding where to place a new block in a possibly fragmented memory. In chapter 5 below it will be seen how a complete family of placement rules or allocation algorithms, including all those above as well as many others, can be obtained by choosing different values for those particular elements which may be varied in a set of "allocation matrices" which are defined in that chapter, and which characterise all the possible placement rules. Each resulting set of these matrices obtained by a particular choice of values can be considered as a representation of an allocation algorithm, and conversely each algorithm determines a set of values to be used in the allocation matrices.

Mention should be made of the extensive simulation comparisons performed by Nielsen (1977), and the simulations of Weinstock (1976). Both of these authors parameterised storage allocation schemes not only by the above choice of placement rule, but also in terms of the choice of free list order, search strategy, compaction strategy and rounding up rule used by each algorithm as well as some other variables, and simulated some of the large number of resulting different schemes. Both were concerned with execution speed at least as much as the efficient use of storage space however, if not more so, and while this is a worthwhile concern it is outside the scope of the problems addressed by the present work. Nielsen characterised his input request distributions mainly by duration and inter-arrival time, and measured the maximum rather than the average space used by any scheme during a simulation run. Weinstock also measured the probability of failure (overflow caused by fragmentation) as a fraction of storage utilisation. Both were concerned with storage allocation models in which the queue of requests is not saturated, and indeed for which it is considered as a failure if it becomes so, that is, if a request arrives which cannot be immediately serviced. Consequently of course, their measures of the efficiency of storage space use cannot be applied to a saturated queue model. They do not give figures for the average storage utilisation, which should in any case be predictably lower than for saturation.

2.3 Empirically observed distributions of request size and duration

It is a remarkable fact that there are apparently very few existing published measurements concerning the distributions of request size and duration which have actually been observed to occur for storage allocation schemes in real computing systems. Consequently the analyses and simulations of such schemes which have been made have either had to rely on the few measurements that have appeared, or else more usually on some assumption about the shape of the distributions, which one may suspect are often chosen either just according to the author's intuition or for their tractability, or perhaps both. The only other possibilities open to would-be modellers in this respect are either to ignore the distribution altogether, as with the fifty per cent rule (below, section 2.4) or else to study a wide range of possibilities whether by simulation or analytically, perhaps by considering the general case.

Batson, Ju and Wood (1970) measured the distribution of segment sizes requested by Algol programs running on a Burroughs B5500 system in a university environment. They found some variation between requests generated by system activities such as the operating system and compilers which favoured a fewer number of larger segments without much variation of different sizes, and requests from user programs which were generally smaller and ranged widely in size. The distributions they found were for large numbers of relatively small segments, with a long tail of a few requests for much larger segments. These matched an exponential distribution only fairly roughly, although since the mechanism for generating storage demands was hidden they could not propose any other sensible model which would serve to parameterise the results. Later, by a different method Batson and Brundage (1977) again measured the segment sizes and lifetimes of working Algol programs in the same system. They again (not surprisingly as it was presumably a later version of the same system) found quite similar results, and that almost all of their empirical data did not fit very well to the exponential or other distributions commonly used in stochastic modelling.

Bryan (1967) reported measurements made over a long period on the JOSS computing system of the Rand corporation. The limited amount of data which he collected and reported on program sizes indicated that they could be approximated reasonably well by a negative exponential distribution. The data presented on program duration are similar to that of Batson and his colleagues (1970, 1977) with a long tail of a few requests for long storage duration or residency times. Although the first part of the curve can be fitted quite reasonably, the upper part of the tail, although small at any point, is less well approximated to by a negative exponential distribution.

Totschek (1965) collected a variety of data on the use made of the SDC time sharing system, including program sizes and computation and elapsed times. The observed frequency distribution of program sizes for the jobs which were run during the collection periods was W-shaped; programs tended to be under 10000, or about 20000, or over 40000 words long. Nearly half of these jobs involved utility functions such as compilation, interpretation, editing and filing. He made a number of statistical analyses of the collected data samples, and one of his findings was that program size, computation time and elapsed time were not significantly correlated for the jobs being run in the system. He also noticed that the observed distributions of the computation, elapsed and inter-arrival time for each job all had a similar shape. These times ranged over several orders of magnitude, there was a long slowly decreasing tail, and the variance was larger than but proportional to the mean. Totschek reported that these distributions could be fitted reasonably well by a 3-parameter lognormal curve.

Margolin, Parmelee and Schatzoff (1971) measured request size and duration statistics for the IBM CP-67/CMS computing system, and used the data to simulate and measure the performance of alternative allocation strategies which were candidates to replace and improve upon the existing algorithm. They were mainly concerned to find ways of reducing time inefficiencies while maintaining roughly the same space requirements as the original algorithm.

Their paper appears to be the only reported work in the literature which both presents such data for a real system and also the effects of using it to improve performance. Although Fenton and Payne's (1974) work was for the same general purpose of guidance in designing an efficient storage allocation scheme, they did not present any existing data on request distributions and so they could not "tailor" the algorithm in the same way; they used a variety of different distributions instead. Margolin, Parmelee and Schatzoff found that in CP/CMS, requests were not independently distributed, request size and duration were not independent, and the distribution did not have the simple profile (such as uniform or negative exponential) which is often assumed for tractability. This published data on request durations does not differ very much in general shape from negative exponential if the dependence of duration on request size is ignored. The distribution of sizes is quite irregular however, and they comment that the dependence between the two is a significant complication which can easily produce strong effects on performance. After quite extensive data collection and correlation and subsequent simulation, they found that an algorithm which gave special treatment by keeping separate lists of free gaps for the thirteen most frequently requested block sizes, reduced the processing time spent in the allocation algorithm by a factor of seven or eight to one. They also reported a slight gain in storage utilisation, but this was much less significant than the time saving.

Another example of a practical investigation of the effects of storage fragmentation on a real computing system is the work of Scherr (1966) subsequently extended by Lehman and Rosenfeld (1968). The operating system which they simulated and measured the performance of was IBM's OS/MVT, (Multiprogramming with a Variable number of Tasks), and the fragmentation that was studied was of the allocation of storage to each step of a job. The simulations modelled the behaviour of the real system closely and were therefore correspondingly complicated by a number of features of the way that MVT controls the selection and progress of jobs to be executed. For example, jobs in storage are not treated equally when sharing processing time, but instead the job-step that has been in storage longest is given the highest priority for execution, and thus runs as if it were alone. Run times for other jobs have to be multiplied by expansion factors in the simulation to allow for the fact that each job can run only when all jobs of longer residency in storage are in the wait state. The input data to the simulation was also an attempt at realism, a "standard" job stream containing Fortran, Cobol and Sorting jobs. The memory requirement or requested size varied according to the type of job-step, and was either chosen from a range of small sizes or else one of three possible large sizes. A figure was given for the average execution or run time but the distribution used was not specified. The conclusions of this work included the observations that the throughput rate of jobs was basically unaffected so long as the multiprogramming level and storage space were sufficient, although fragmentation did affect the order in which jobs were completed and hence the delay characteristics. Some jobs could be kept in the system for as much as 8 hours as a result of fragmentation, while other classes of jobs with less storage requirement suffered almost no delay. This suggested that modifying the first fit algorithm used to allocate storage to job steps would be a good idea, to take account of the jobs expected (by their job class) to stay a long time in the memory, and not divide the memory in two by allocating them in the middle but put them at one end or other of memory if at all possible.

These few results indicate that modelling real-life situations should be performed with as much generality as possible in the allowed distributions of request size and duration, although to choose the negative exponential distribution for request duration times is less unrealistic in most cases than for the size distribution. Thus the choice in the present work (chapter 3, section 3.2) of this exponentially decreasing distribution of request times, which has been dictated by the requirement of simplicity at least to begin with and the desirability (as will be seen) of being able to use Markov theory, is not necessarily an unfortunate one. In any case it has been found possible to allow a general distribution of request size to be assumed so that there can be no objection to this, although the two requested quantities size and time are assumed to be chosen independently of each other for the same request, and of other requests.

2.4 Fifty per cent, two-thirds and unused memory rules

The fifty per cent rule of Knuth (1968) has become so well known that no survey of storage fragmentation would be complete without mentioning it. Stated rather briefly, this rule says that on the average there will be only half, or fifty per cent, as many free gaps in memory as allocated blocks. In the same style as this result and based directly upon it are the unused memory rule of Denning (1970), and the two-thirds rule of Gelenbe (1971). All three are concerned with the equilibrium behaviour of an unsaturated storage allocation scheme and rely on intuitively simple arguments about average quantities. That they are approximate indicators of average behaviour cannot be denied, but their predictions should not be taken as exact except in special cases, as the following closer examination of their statement and derivation shows.

The quantity p is defined as the probability that an arbitrarily chosen allocation of a block will be into a gap of larger size than the block, leaving a smaller resulting gap which remains unallocated. Then if the average number of allocated blocks in equilibrium is N and the average number of free gaps is M, the fifty per cent rule actually states that

$$M = 0.5pN \qquad \text{.... 2.3}$$

Denning's unused memory rule is just a trivial corollary; if the average gap size in equilibrium is no less than k times the average block size, then the average fraction of the memory occupied by free blocks is no less than k/(k+2). The two-thirds rule is also derived from the fifty per cent rule and states that if the probability (1-p) that an arbitrary block allocation is into a gap of exactly the right size is very small (this is intuitively likely for large memories where the range of possible block sizes is large compared with the average number of allocated blocks, especially if the best fit algorithm is not being used) then the average memory utilisation U is nearly equal to 2/3. If p=1, then U = 2/3 exactly according to this rule. The two-thirds rule includes the unused memory rule, as during its derivation it is "shown" that the average size in equilibrium of a gap is equal to the average block size, so that both these rules say the same thing. Interestingly, although Gelenbe references Denning, it is for Denning's derivation of the fifty per cent rule, and he fails to even mention the unused memory rule.

These rules take little or no account of the allocation algorithm or the distribution of request sizes; all are supposed to give the same average utilisation in equilibrium. That this is not so, although for many cases it may not be far wrong, is indicated by almost any of the simulation studies of models with unsaturated request queues which have been carried out. Shore (1977) has noted deviations from the fifty per cent rule and investigated these by simulation. He advances an explanation that the systematic allocation of blocks always to the same end of gaps will reduce the free-to-allocated ratio below 0.5 if the coefficient of variation (standard deviation/average) of the distribution of request duration times is less than about 0.8. He discusses this in terms of the relative ages of the blocks in contiguously allocated groups. The groups tend to migrate along the memory because of the systematic placement bias which will always join a new block at the same end of an already existing group. The oldest block in such a contiguous group is usually at or near the trailing end (unless the group behind it catches up), which upsets the assumption of uniform randomness about which category (see below) a deallocated block is likely to belong to.

The derivation given by Knuth of the fifty per cent rule is in fact as follows. "Consider the following memory map, figure 2.2. This shows the

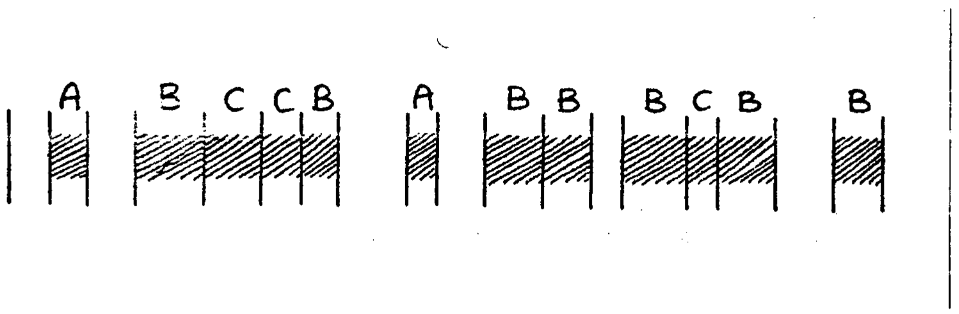


Figure 2.2 Example of a fragmented (or "checkerboarded") memory

reserved blocks divided into three categories, A (surrounded by two gaps), B (surrounded by one gap and another block), and C (surrounded by two other blocks). Let N be the number of reserved blocks, and M be the number of available gaps", [at this point some vagueness creeps in as N has by this time already been defined as the average number of allocated blocks], "and let A, B, C be the number of blocks of the above categories. Then

$$N = A + B + C$$
$$M = 0.5(2A + B + e) \qquad \text{.... 2.4}$$

where e = 0, 1, or 2 depending on conditions at the boundaries." [In a circular memory, e (for end) = 0; this variable is probably unimportant as far as this rule is concerned, for large memories.] "To derive the fifty per cent rule, set

$$\text{Probability[M increases by one]} = \text{Probability[M decreases by one]} \qquad \text{.... 2.5}$$

which leads to

$$C = A + (1-p)N \text{ "} \qquad \text{.... 2.6}$$

Equation 2.5 is an imprecise statement, although Knuth does admit this and qualify it, "...more precisely, the average change in M per unit time is set to zero during equilibrium". If 2.6 is accepted, then the fifty per cent rule 2.3 certainly follows by the use of 2.4, when the end-effect variable e is neglected.

The objection which may be raised to the above argument is as follows. If M is an average, then it is a fixed quantity and it is nonsense to discuss "M increases or decreases by one". If on the other hand M is a random variable, then it must be dependent on the particular configuration of blocks and gaps in the memory, and no justification for the statement of 2.5 or its derivation as 2.6 is given for any arbitrary configuration in general. The same remarks apply to the quantity p, originally defined as an average, but representing a (random variable) probability which is obviously dependent on the particular configuration being considered. It is a little surprising that apparently no-one has bothered to point this out before.

2.5 The allocator-defender games of Robson

Robson (1971, 1974, 1977) has studied those storage allocation schemes which can be considered in terms of a game played between an attacker and a defender. The game is played with a fixed maximum number M of tokens, each representing one word, which can be formed into strings to represent blocks to be allocated, on a board representing the memory. The length of a string (or size of a block) may never exceed n words, and M and n are given parameters of the game. The attacker chooses and removes strings of tokens (allocated blocks of words) from the board (memory), and forms new strings which the defender must then decide where to place on the board. The number $N^*(M,n)$ is defined as the size of the smallest board on which the defender, with correct play, can always manage to place the strings whatever the attacker may do. Robson's work, and subsequently that of Krogdahl (1973) and Ting (1975), is concerned with finding and improving upon bounds for this quantity $N^*(M,n)$. Upper bounds are discovered by inventing and improving on successful strategies for the defender if enough memory is provided, and lower bounds by conversely discovering a strategy for the attacker (in a smaller memory of course) which must lead to embarrassment for the defender by forcing a situation in which he is presented with a string which will not fit on the board. The strategies which have been described become cleverer and more complicated as the bounds are improved. In most real-life situations, storage allocation is not played as a game in this way, and Robson's work is concerned with the worst behaviour possible by the attacker and how to cope with it. Although such "pathological" sequences of requests and releases are not impossible, they are generally extremely unlikely in practice (unless of course, this game is explicitly being played). There is quite a difference between the amount of memory $N^*(M,n)$ required for the defender to be sure of winning against all comers, and the smaller amount which is sufficient in more general situations to make the probability of overflow very small.

Typical results are the later ones published by Robson. In his 1974 paper, he showed that the optimal strategy for the defender requires between $0.5M\log_2 n$ and about $0.84M\log_2 n$ words asymptotically, as M and n increase and where n is much less that M, although practical allocation strategies may require more space than this. The 1977 paper shows that the first fit algorithm is after all quite close to the optimal strategy and requires not much more memory, about $M\log_2 n$ words altogether. On the other hand the best fit algorithm needs about Mn words. Robson points out that "these results should be contrasted with the simulations of Shore (1975) which suggest that even when an allocation system is run on the brink of breakdown due to fragmentation, storage utilisation averages about 70 to 95 per cent for a fairly wide range of distributions of allocated block size. Clearly the sort of catastrophic fragmentation which is possible occurs only very rarely". Robson's work has stimulated three other authors, Knuth (1973), Krogdahl (1973) and Ting (1975) to publish further results. Knuth showed that in the buddy system the defender requires $2M\log_2 n$ words to always succeed. Krogdahl considers extending the game by specifying the allowable set of string (block) sizes as a set of integers $(b_1,\ldots,b_n)$. If attention is restricted to the cases where n divides M and is small compared with it, Robson showed (1974) that the ratio $N^*(M,n)/M$ tends to a limit of 3/2 as M increases, if n=2. Krogdahl showed that for block sizes $b_1 < b_2 < \ldots < b_n$ this limit becomes

$$1 + (1 - b_1/b_2) + \ldots + (1 - b_{n-1}/b_n) .$$

Ting removes the restriction that n should be much less than M, and shows that in this case the optimal defence strategy needs no more than $M\log_2 n - M/2$ words for all $n = 1,\ldots,M$.

2.6 The generating function technique of Reeves

Recently Reeves (1979, 1980) has analysed a random fit storage allocation model algebraically by a generating function technique, with encouraging success. In his model, the memory is assumed to be circular to avoid end effects, the queue of requests is not saturated so that requests are allocated the moment they appear (and so the probability distribution of the relative frequencies of their sizes in the memory is therefore known), and the random fit allocation method is used. The choice of these three characteristics of the model represents good judgement as each one has the effect of bypassing difficulties which would otherwise occur in the analysis. The size distribution in equilibrium of free gaps in the memory is represented by a generating function

$$\phi(a) = \sum_r \phi_r a^r$$

where ϕ_r is the equilibrium probability that a randomly selected free gap has length r. A similar generating function b is defined for the size distribution of generated requests, and ϕ is shown to be related to b by studying the actions occurring at block allocation and release. The analysis contains an argument, based on an assumption that certain configurations are equiprobable, which relates the probability (p_2) that a randomly selected allocated block is surrounded by gaps on either side, to the "block utilisation" ratio k = B/F, the ratio of the number B of allocated blocks to the number F of free gaps. The subsequent paper by Reeves (1980) continues the analysis and concentrates on the numbers of allocated blocks occurring in contiguous sequences irrespective of their sizes, and defines σ_r to be the probability that a randomly selected such sequence contains r blocks. Then as before a generating function

$$\sigma(a) = \sum_r \sigma_r a^r$$

is defined for this distribution. By again considering the equilibrium probabilities at block allocation and release, a condition on $\sigma(a)$ is found, and is immediately used to obtain a revised expression for p_2 in terms of k which is more satisfactory because it no longer depends on the former assumption. The analysis treats in detail the case when all requests are for just one word, the relation between ϕ and b and the condition on σ simplifying for this particular distribution. An interesting prediction is that for stable solutions ϕ to exist the storage utilisation has to be above a certain threshold found to be about 0.48, and this is confirmed by simulating the model for a range of values of the storage utilisation. These simulations also confirm the predictions of the model for higher values of storage utilisation. Reeves also observed from this simulation that at low utilisations the free memory tends to form into one large gap and a number of smaller gaps, so that ϕ is unstable because the model is not a suitable description of this situation. He developed a revised version of the model containing an infinitely large gap, and this makes predictions for low utilisation which are in good agreement with the simulation results.

The future work with this model should extend the analysis to variable request sizes, perhaps possibly as far as being able to investigate for example the anomalies noted by Shore (1977) in the behaviour predicted by the fifty per cent rule. It is clear that continuing to extend this use of generating functions as developed so far by Reeves will be a most useful way of coping with the combinatorial complexity of storage fragmentation problems.

2.7 Space filling : the "car parking" and other problems. A note of caution

Loosely related to storage allocation and fragmentation are a number of different problems which can be described under this heading. Although the published work on space filling problems is not directly relevant it can be helpful by suggesting various points of view which can be used for thinking about the present problem as well as ideas for ways to try and tackle it. Further, this work forms a useful background of knowledge of what may be called the static properties of storage fragmentation as opposed to the dynamic behaviour which starts when blocks are moved about by allocating and deallocating them. This section concludes this chapter by presenting a brief summary; it is by no means a complete survey.

Renyi (1958) considered the one-dimensional model which repeatedly allocates unit sized non-intersecting intervals uniformly at random in the initially empty real interval (0,x) until no further allocations are possible. Allocating an interval "uniformly at random" in (0,x) means, for each succeeding interval to be allocated, choosing its midpoint uniformly at random from the interval (0.5,x-0.5); if this chosen interval would overlap one already allocated, the choice is discarded and another is tried. Thus there is no commitment to placing each interval at one end of a gap, and indeed such an occurrence has vanishingly small probability (compare the discussion of the random allocation algorithm in section 2.2 above). He obtained a closed form expression for the expected number M(x) of allocated intervals, and showed that as x increases M(x)/x approaches a constant limit c = 0.748 approximately. Mannion (1964), and Dvoretzky and Robbins (1964) studied the distribution of M(x) as opposed to just its mean, and showed it to be asymptotically normal for increasing x. Palasti (1960) extended Renyi's work by considering the equivalent problem in two and three dimensions, conjecturing that for the expected number $M(x,y)$ of non-intersecting unit squares which can be placed at random in a rectangle of sides x and y, $M(x,y)/xy$ similarly approaches a limit which is $c^2 = (0.748)^2$ as x and y increase. Ney (1962) has considered the important generalisation of this interval filling problem on a continuous line in which the lengths of the allocated intervals are also allowed to vary randomly, and has studied the mean and moments of the number $N(b,x)$ of such intervals of length at least b in the interval $(0,x)$. He showed that the mean of $N(b,x)$ approaches a linear function of x as x increases, and for example determined this function to be: $C(b)x$, when the positions of the intervals are uniformly distributed, and where $C(b)$ is a certain specified integral dependent on b and on the length distribution of the allocated intervals. The origin of this work is in the theory of "cascades" of colliding (atomic) particles and their energy levels. As far as the present author can discover, Ney was the first to describe this, in the literature anyway, as the "car parking" problem. In this very convenient and descriptive analogy, a street is represented by the interval $(0,x)$ and the cars which are to be parked on it (without overlapping!) by the allocated intervals. Solomon (1964) has surveyed some of the above work on this problem, and repeated and extended the Monte Carlo experiments performed by Palasti to support her conjecture.

As for the relevance to storage allocation schemes, $c = 0.748$ is the value of the average storage utilisation which should be expected from the corresponding random storage allocation model in which all the requests are for the same unit size. In such a model, the queue would be saturated, the request distribution would specify blocks of a single constant size much less than the memory size, and the allocation algorithm would be random in the same (not left or right justified) sense that Renyi intended it.

Stevens (1939), Robbins (1944), Votaw (1946) and Domb (1947) have considered aspects of the related problem in which a finite number of intervals are placed independently of each other. This lack of dependence is a fundamental difference since the intervals may then overlap. Stevens obtained frequency distributions for the number of gaps when n intervals of length x are placed in this way at random on the circumference of a circle of unit length. For instance, he found that the probability f(k) of obtaining the maximum of n gaps (no overlapping) where k is the greatest integer less than 1/x, is

$$f(k) = \binom{n}{k}(1-kx)^{n-1} \qquad [\ \binom{n}{k} = \text{binomial coefficient}\ ]$$

The expressions for the other f's are similar but more lengthy. Robbins found expressions as integrals for writing down the expected value and higher moments of the amount covered by at least one interval without having to first determine its probability distribution. These expressions generalise quite naturally to the case where the overlapping intervals become overlapping sets in Euclidean n-space, and the random placement is not uniform but specified by a given probability distribution function. Votaw considered the latter difficult problem of determining the frequency distribution of the amount covered, for intervals in one dimension again, and succeeded when the intervals are randomly placed either uniformly or distributed negative exponentially. Domb considered all these problems and arrived at the same results by different analytical arguments which lead naturally to the use of continuous functions, making great use of Laplace transforms to solve the integral equations which arise.

Page (1959) studied a similar discrete problem suggested by physical experiments in which hydrogen atoms are first deposited in pairs as molecules on a rectangular lattice surface, and then subsequently removed again in possibly different adjacent pairs. He considered the analogue in one dimension, allocating pairs of points at random on a line of N points until only single vacancies remain, and found the mean and higher moments of the distribution of the number of remaining vacancies, the mean for example approaching approximately 13 per cent of N as N increases.

Although it is a digression, it may be worth bearing in mind that there are other kinds of physical experiments which lead to similar problems concerned for example with the number of regular solids, usually spheres or regular polyhedra, which can be closely packed together either regularly or at random into a given volume. This can be applied for example to some aspects of the study of the growth of cells in undifferentiated living tissue, and the physical properties of liquids. The papers of Bernal and Mason (1959, 1960, 1960a), Scott (1960, 1962) and Coxeter (1958) introduce the latter as a new and relevant aspect of this much older problem. As with simulation for computer storage fragmentation, they relied heavily to begin with on actual experiments in which the volume of lead shot, ball bearings, marbles and even plasticine pellets were measured inside such varied containers as steel cylinders under compression and uninflated balloons immersed in water. Matzke (1950) has surveyed the history of this problem in an amusing address in which he warned of the dangers of relying too much on intuition by, for example, assuming that regular patterns are formed. Experiment and observation (by him and others on lead shot, living cells, bubbles in a foam and even by repeating an apparently famous two-centuries old experiment of compressing peas, known erroneously it seems as "Buffon's peas") show that this does not in fact happen, whether for peas or for bubbles, although for many years even leading authorities of the scientific community believed that it did. Perhaps

Matzke's cautionary remarks are the most relevant contributions that this work can make to storage fragmentation studies; Randell's surprise on observing the effects of rounding on fragmentation and utilisation is a case in point.

Chapter 3 : Definition of a model of storage allocation

One quantity of great interest in any storage allocation scheme is the average or expected utilisation $U = U(T)$, defined as the expected value of the fraction of memory occupied at time T. In many present day systems this is obviously important for main memory (which in, say, the Burroughs B5500 and B6700 series of computers is allocated in a way which gives rise to fragmentation, see Organick (1973) for just one example). Decreases in main memory cost are unlikely to make utilisation of no importance whatsoever, and in any case the allocation of secondary memory is likely to be of considerable importance for some time to come. The average utilisation is usually the most important quantity whenever the queue of requests is saturated or nearly so, since it determines the throughput or rate of servicing requests, which in most cases is desired to be as high as possible. In the stochastic model of storage allocation which is about to be defined, U may be calculated as a scalar product:

$$U(T) = \underline{\pi}(T) \cdot \underline{u}' \qquad \text{(scalar product)} \qquad \ldots.\ 3.1$$

where $\underline{u}'$ is the transpose of a constant row vector $\underline{u} = (ui)$:

$$ui = \text{fraction of memory occupied by allocated requests when the model is in state } i, \qquad i = 1,\ldots,S' \qquad \ldots.\ 3.2$$

and where the model has a certain number S' of states, and $\underline{\pi}(T)$ is a vector of state probabilities at time T. Under certain general conditions, as T increases $\underline{\pi}(T)$ will converge to an equilibrium vector $\underline{\pi}$, so that U(T) will converge to the steady state utilisation U :

$$U = \underline{\pi} \cdot \underline{u}' \qquad \text{(scalar product)} \qquad \ldots.\ 3.3$$

The model provides the means of calculating and investigating U, principally by studying or using properties of the matrix P of transition probabilities between states. The steady state equation, 3.11 below:

$$\underline{\pi} = \underline{\pi} P$$

is the starting point for the algebraic analysis in chapter 5. Values of $\underline{\pi}(T)$ and hence U(T) for increasing T may be calculated by the well known power method, equation 3.13 below:

$$\underline{\pi}(T+1) = \underline{\pi}(T) P$$

by starting from some arbitrary initial distribution $\underline{\pi}(0)$ of state probabilities at T = 0. The utilisation U can be obtained this way as U(T) converges, and some work and results from this method are reported in chapter 6. Unless the transient behaviour is of particular interest, only the limit $\underline{\pi} = \underline{\pi}(T=\infty)$ is needed. Other quantities of interest besides U such as the expected amount of internal fragmentation can also be calculated knowing the request distribution and hence the average rounding up for each size of block, and the expected number of allocated blocks of each size. These last numbers can be obtained from $\underline{\pi}$ or $\underline{\pi}(T)$, the computation of the state probability vectors giving a complete description of the probabilistic behaviour of the model.

3.1 Definition of the storage allocation model to be investigated

Suppose that a computing system has a fixed size memory of N words and its control program contains an allocation algorithm servicing a queue of requests, each for a single block of contiguous words. Figure 3.1 is intended to illustrate this. The unit of allocation in a practical application might

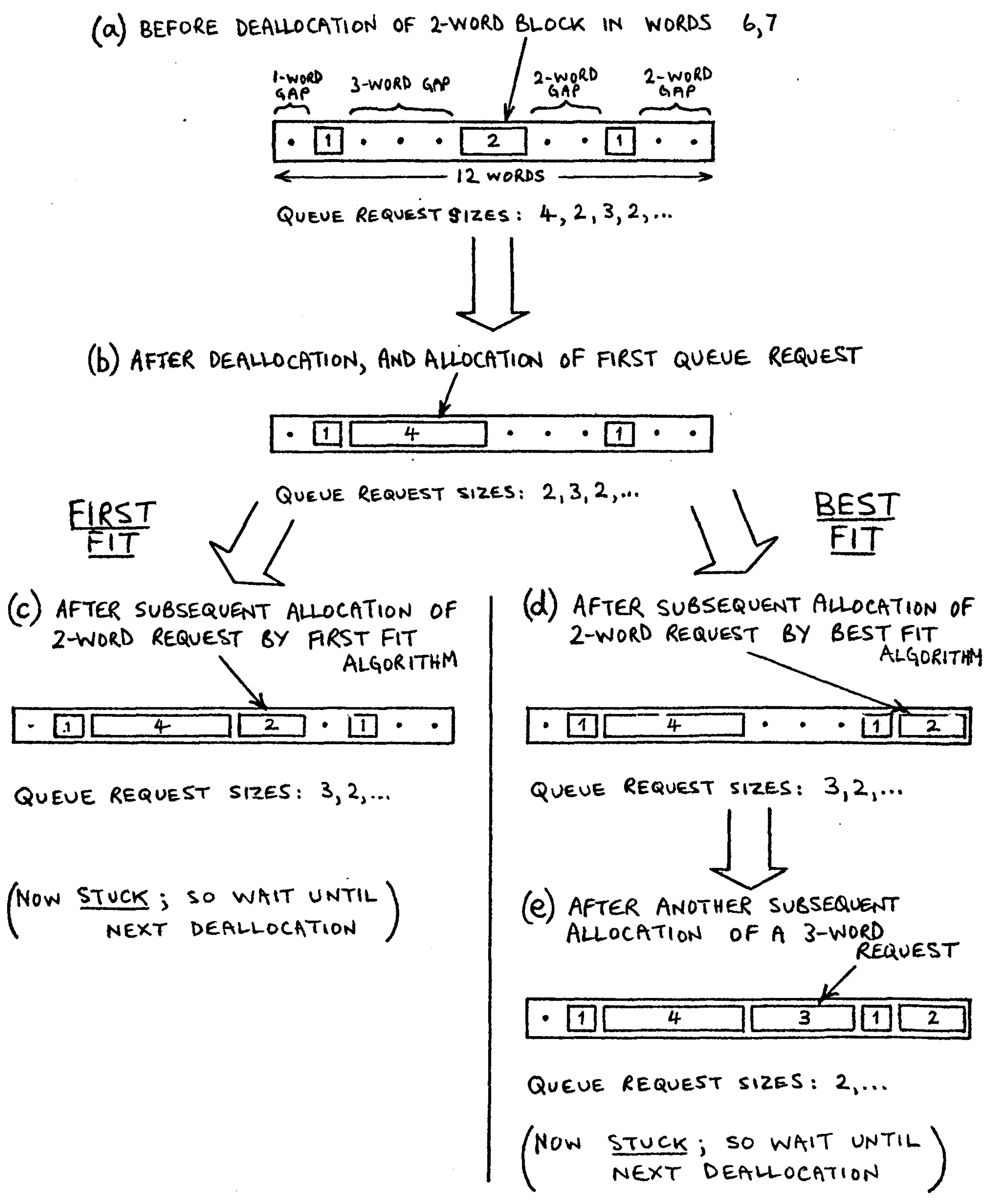

(a) BEFORE DEALLOCATION OF 2-WORD BLOCK IN WORDS 6,7
1-WORD GAP
3-WORD GAP
2-WORD GAP
2-WORD GAP
• 1 • • • 2 • • 1 • •
12 WORDS
QUEUE REQUEST SIZES: 4, 2, 3, 2, ...
(b) AFTER DEALLOCATION, AND ALLOCATION OF FIRST QUEUE REQUEST
• 1 4 • • • 1 • •
QUEUE REQUEST SIZES: 2, 3, 2, ...
FIRST FIT
BEST FIT
(c) AFTER SUBSEQUENT ALLOCATION OF 2-WORD REQUEST BY FIRST FIT ALGORITHM
• 1 4 2 • 1 • •
QUEUE REQUEST SIZES: 3, 2, ...
(NOW STUCK; SO WAIT UNTIL NEXT DEALLOCATION)
(d) AFTER SUBSEQUENT ALLOCATION OF 2-WORD REQUEST BY BEST FIT ALGORITHM
• 1 4 • • • 1 2
QUEUE REQUEST SIZES: 3, 2, ...
(e) AFTER ANOTHER SUBSEQUENT ALLOCATION OF A 3-WORD REQUEST
• 1 4 3 1 2
QUEUE REQUEST SIZES: 2, ...
(NOW STUCK; SO WAIT UNTIL NEXT DEALLOCATION)

Figure 3.1 Illustration of storage allocation

be a word, a page, or some other fixed size unit of memory, and "word" is used here to include all these terms. Each request specifies a random variable r (the size of the request) which may have any integer value from 1 to N words. The queue is infinite or saturated, i.e. it never becomes empty, and requests are serviced in a first-come-first-served order as soon as sufficient free space becomes available for the request at the head of the queue. A request of size r is allocated by choosing r contiguous free words from the memory, and an allocation algorithm is used to make a choice if this is possible in more than one way. Figure 3.1 illustrates the different actions of two possible example allocation algorithms. When blocks are deallocated any adjacent gaps of free words are merged together, but the blocks which remain allocated are not moved. Queued requests are then serviced in order until one is reached for which insufficient contiguous free storage is available. The new configuration of memory and queue thus arrived at then remains unchanged until the next deallocation occurs. Deallocations and allocations are considered to be performed instantaneously, and the probability of two or more blocks being simultaneously deallocated is zero since the timescale is chosen to be real.

3.2 Assumptions leading to Markovian nature

Despite the assumptions already made about the model's operation, (saturated first come first served queue, requests serviced as soon as possible), it is still too general to be considered as a finite discrete-time Markov chain and some more assumptions mainly concerned with the passage of time are necessary before this is possible. Before stating them precisely, some discussion and justification of their validity is in order. They are of course not necessarily the only ones which achieve the requirements for the model to be a Markov chain.

Computing systems which share processor time in equal amounts in a "round-robin" are common, and "processor sharing", the term given to the limit of this behaviour as the quantum of allocation time tends to zero, has been modelled quite extensively; see Kleinrock (1976) for example who lists some of the work on processor sharing, or Coffman, Muntz and Trotter (1970), who incidentally also make the assumption below on exponentiality. Thus it is quite reasonable to assume that the time of some processor imagined for the purpose is being shared equally among all the blocks present in storage at any moment, each of them having an equal claim for attention. It is also not particularly unreasonable to assume that each request requires a randomly variable amount of this processor's unshared time or undivided attention which is distributed negative exponentially, so that most requests are for short time durations with a progressively smaller minority needing increasingly longer times. This assumption has also been made in the past in the absence of any better conclusive information than the few studies discussed in section 2.3 of the last chapter. Whilst this little evidence suggests that negative exponential is not necessarily a very close fit to real life distributions, nevertheless it indicates that it has the right general shape and is a lot better than the uniform distribution, say. Of course, the negative exponential distribution is attractive as it does lead to some convenient mathematics, and this also brings one to suspect that it is found more often in the models of the literature than it occurs in real life. However, for the purpose of the analysis presented here, these two assumptions (processor sharing and negative exponential request durations) are actually only needed to justify the statement that at any moment each of the remaining blocks allocated in storage is equally likely to be the next to be deallocated, and if this latter is taken as a starting assumption then the first two are not required. In general, equal likelihood of deallocation is sometimes the only reasonable assumption that can be made in the absence of other information, and of course it has been used previously in analytical studies of storage allocation models, by Reeves for example (1979,1980). In particular, because it follows as a consequence from the preceding two assumptions, and as it also may be true or nearly so to a greater or lesser extent in other cases where they do not hold, it is not particularly restrictive judged by the standards of previous work. In practice therefore, this assumption or consequence of equally shared deallocation probability among the remaining blocks in storage should be quite widely applicable to a fair proportion of real life situations.

The corresponding necessary assumptions made about the distribution of request sizes can hardly be considered restrictive as the distribution may in general have any "shape", the only requirement being that it be unvarying, or constant. This is likely to be the case in very many computing systems for relatively long periods of time compared with the average lifetime of a block, the more so where many independent but similar activities are going on. Two more related assumptions are that successive requests are independent and that each request asks for independently distributed storage size and duration time. As pointed out in section 2.3 although Totschek (1965) found no significant correlation in the SDC system, Margolin, Parmelee and Schatzoff's (1971) observations of the CP-67 system do not confirm this. This is unfortunate although perhaps not surprising, but to take account of such dependencies would so greatly complicate matters that it is better to assume independence in the analysis at least on a first attempt. At two places in the present analysis, it has been necessary to assume that the request size distribution is not quite general but that (in the ergodicity proof of section 3.3.3) the probability of a request for a block of unit size is non-zero, which is reasonable, and further on (in the definition of the inverse of the basic allocation termination matrix in section 5.1.3), that the probability of a request for a block equal to the size of the memory is non-zero, which is much less reasonable. The probabilities can of course be very small. The consequences of these restrictions are discussed as the need for them arises, and there are good indications that both can be relaxed.

3.2.1 Some precise assumptions allowing Markov theory to be used

1) For each request, an independent random variable t called the "processing time" is defined which may have any positive real value. t is the amount of time spent "processing" for which the block allocated to the request needs to remain in storage. The probability distribution of t is assumed to be constant, negative exponential with mean $1/\lambda$ for some $\lambda > 0$. Successive new requests joining the queue have independently distributed values of t, and also t and the request size r are independent of each other.

2) (Processor sharing.) Imagine the existence of a single "processor" which works at a uniform constant rate in real time. If at any instant there are $n > 0$ allocated blocks in memory, each of them receives $1/n$ of the "attention" or time of this processor. That is, the time spent in "processing" any given block accumulates at a rate which is $1/n$ that of real time. When the accumulated processing time for any given block matches the amount requested, t, it is deallocated.

Section 3.2.3 below shows that (1) and (2) imply (not very surprisingly) the following two statements:

3) The distribution of real time t_e which elapses between successive deallocations is known and constant (in this case, the same as the distribution of t, deallocations occurring at a rate λ).

4) At any moment, each of the blocks present in memory has equal probability of being the next to be deallocated, independent of its size and the time it has been resident.

As mentioned above, these assumptions (3) and (4) may be true in practice for various reasons, but they follow automatically if assumptions (1) and (2) are made.

5) Definition of request size probability distribution:

The probability distribution (r_n) of the random variable r, the size of requests joining the queue, is assumed to be constant (and known):

$$r_n = \text{Probability}[r = n] , \quad n = 1,\ldots,N . \qquad \ldots . \; 3.4$$

The sizes of successive new requests are taken independently from this distribution, which at least to begin with is otherwise unrestricted,

$$\sum_{n=1}^{N} r_n = 1 \quad ; \quad r_n \geq 0 , \quad n = 1,\ldots,N . \qquad \ldots . \; 3.5$$

3.2.2 States and transitions between states

For the model to be considered as a Markov chain the states of the chain, and transitions between states, have to be defined. The dependence of transitions only on the current state must of course also be shown, and this is done in the next section. A state of the chain is specified by giving the sizes and locations of all the allocated blocks in memory. Configurations of the memory and queue which differ in these respects, and these only, define different states. In figure 3.1, for example, (a), (c) and (d) represent different states, although it happens that (c) and (d) have the same total utilisation and (a) and (d) the same maximum gap size. Assumption (4) above makes it unnecessary to include time information when specifying a state. The number of different states obtained by this definition is obviously finite for a given size of memory, N, although it is large; in chapter 4 it is shown that the total number of states increases exponentially with N, so that for example (see table 4.1, section 4.4.5) there are 89 states in a five word memory, and 10946 if N = 10. A transition occurs whenever a deallocation takes place, and it is defined to be a deallocation followed by (possibly zero) subsequent allocations from the queue which continue until the request currently at the head of the queue is too large to fit in even the largest free gap in the memory. This instantaneous operation is considered to be indivisible for the purpose of defining transitions between states, and although the intermediate stages of memory encountered during the allocation process are (except for the empty configuration) valid states of the model, and would be reached on another occasion with a different set of requests in the queue, they are discounted for the analysis in this chapter. The separation of complete transitions into sequences of simpler transitions between the intermediate stages is studied in chapter 5.

3.2.3 Proof that the model is Markov with the above assumptions

This section presents a proof that assumptions (4) and (5) above are sufficient for the model to be Markovian with the above definitions of state and transition, and it is similar to that given for the storage allocation model described by Betteridge (1974). As a preliminary to this proof, the next two paragraphs show that assumptions (3) and (4) are indeed consequences of (1) and (2).

Consider the distribution of the real time interval t_e which elapses between successive transitions. Suppose a transition has just occurred, and that as a result there are n blocks in memory. Each of these has a certain amount of processing time to complete before being deallocated, by assumptions (1) and (2). Because requests arriving in the memory have independent t-values distributed negative exponentially by assumption (1), and at this instant these blocks are still present, then the remaining processing times of each block also have this same distribution of t, regardless of however much processing time may have already elapsed for each or at whatever rates (Feller (1968), section 17.6). Therefore, each of the blocks present in memory is equally likely to be the first to be deallocated causing the next event to occur, which is assumption (4).

Since by assumption (2) processing time for each block is elapsing at a rate n times slower than real time, each block currently has a remaining real time t_{real} in memory distributed negative exponentially with mean n/λ , i.e.

$$\text{Probability}[t_{real} \leq x] = 1 - e^{-\lambda x/n}, \quad x \geq 0 \qquad \text{.... 3.6}$$

Therefore the distribution of t_e is that of the minimum of n independent random variables (t_i) each with the distribution of t_{real}. This is easily shown to be the same as the distribution of t:

$$\text{Probability}[t_e > x] = \text{Probability}[t_1 > x \text{ and } t_2 > x \text{ and } \dots \text{ and } t_n > x]$$
$$= \text{Prob}[t_1 > x] \times \text{Prob}[t_2 > x] \times \dots \times \text{Prob}[t_n > x], \quad \text{by independence,}$$
$$\text{so} \quad \text{Probability}[t_e > x] = (e^{-\lambda x/n})^n = e^{-\lambda x} = \text{Probability}[t > x] \qquad \text{.... 3.7}$$

Thus the distribution of t_e is known and constant, and events occur at a rate determined only by λ (from the distribution of t) independently of whatever states the model may enter. This is assumption (3).

Using only assumptions (4) and (5) now, consider the factors which influence state transitions. Suppose the model is in a particular state at some arbitrary time. Then the positions and sizes of blocks and gaps in the memory are known. By assumption (5) and because the queue is first come first served, the probability distributions of size r of all the requests in the queue which may take part in the transition to the next state are known, and are independent and constant, except for the size distribution of the first request in the queue. This has to be dependent at least on the given state, for allocations are performed if at all possible, so the size of this request must be larger than the maximum gap size in memory. In the next section 3.2.4 it is shown that this particular size distribution does in fact depend only on the (given) maximum gap size as well as the distribution of r, and so is also known. Knowing how any given particular allocation algorithm works, everything that is needed to calculate the transitions and their probabilities from the given state is available. These must therefore be independent of the model's past history and of the time at which the model enters the given state, and so with the states and transitions as defined the model is a finite discrete-time Markov chain.

3.2.4 Distribution of request size for the first request in the queue

Because the allocation process continues until the request which is currently first in the queue will not fit in memory, the probability distribution of the size of the first request is different when viewed at times inbetween transitions, because it is affected at least by the memory configuration. The distribution of subsequent requests which have not yet reached the front of the queue is unaffected because the allocation algorithm restricts its attention to this first request. In fact, in any state of the Markov model (i.e. one which is waiting for a deallocation, not an intermediate stage of an allocation process) with configuration C having a maximum gap size $g < N$, the probability distribution of q, the size of the request at the head of the queue, is:

$$q_n = \text{Probability}[q=n] = 0 \qquad \text{if } n \leq g$$
$$= r_n/c_{g+1} \qquad \text{if } n > g \qquad \ldots .\ 3.8$$

where by definition,

$$c_k = r_k + r_{k+1} + \ldots + r_N\,, \qquad k = 1,\ldots,N \qquad \ldots .\ 3.9$$

(The symbol "c" has been chosen, to stand for "cumulative".) The statement of equation 3.8 may seem almost obvious, but in any case it can be justified with a formal proof, to which the rest of this section is devoted.

Consider the last block movement which took place during the state transition which resulted in the present state. It must have been either an allocation or a deallocation.

If the last block movement was an allocation, then immediately before it the request now at the head of the queue was second in the queue. Because of the first come first served queue discipline, up to this point the probability distribution of its size q was independent of the model's operation and so was (r_n), $n = 1,\ldots,N$ (definition 3.4). Applying the theorem of conditional (or total) probability:

$$\begin{aligned} r_n &= \text{Prob}[q=n] \\ &= \text{Prob}[q=n \mid n>g] \times \text{Prob}[n>g] \\ &\qquad + \text{Prob}[q=n \mid n\leq g] \times \text{Prob}[n\leq g]\,, \qquad n = 1,\ldots,N \end{aligned}$$

When the allocation occurs resulting in configuration C and q rises to the head of the queue, it is given that the allocation process stops and therefore that $q > g$, the size of the largest gap in C. It follows that in this case

$$\text{Prob}[q=n \mid n \le g] = 0 \ ,$$

and so $\text{Prob}[q=n \mid n>g] = r_n/\text{Prob}[n>g] = r_n/c_{g+1}$

as required.

If the last block movement was a deallocation, then immediately before it occurred the memory configuration was C', say, with maximum gap size $g' \le g$, and with m' words of memory allocated out of the total N. Suppose that there are m words of allocated memory in C, then $m' > m$. Consider the (descending) inductive hypothesis, for $x = N, N-1, \ldots, 1$:

H(x): in any state with $w \ge x$ words allocated, equation 3.8 above holds.

H(N) is true from the argument above, since any block movement which results in a state with all N words allocated must be an allocation. Assume H(x) is true for all $x > m$. Then it is true for C', and so the request at the head of the queue in state C' has size q' with probability distribution

$$q'_n = \text{Probability}[q'=n] = 0 \qquad \text{if } n \le g'$$
$$= r_n/c_{g'+1} \qquad \text{if } n > g'$$

Again applying the theorem of conditional probability,

$$\text{Prob}[q'=n] = \text{Prob}[q'=n \mid n>g] \times \text{Prob}[n>g] + \text{Prob}[q'=n \mid n \le g] \times \text{Prob}[n \le g]$$

For n satisfying $n > g \ge g'$,

$$\text{Probability}[q'=n] = r_n/c_{g'+1}$$

and $\text{Probability}[n>g] = \sum_{n>g} r_n/c_{g'+1} = c_{g+1}/c_{g'+1}$

When the deallocation occurs from C' to C, it is given that in this case the request q' at the head of the queue remains there, and so $q' = q > g$. Therefore, in this case,

$$\text{Prob}[q'=n \mid n \le g] = 0$$

and so, for $n > g$,

$$q_n = \text{Prob}[q=q'=n \mid n>g] = \text{Prob}[q'=n]/\text{Prob}[n>g]$$
$$= r_n/c_{g'+1} \div c_{g+1}/c_{g'+1} = r_n/c_{g+1}$$

as required. Thus for any state C with m words allocated, if C was arrived at by a deallocation, equation 3.8 holds inductively, and if C was arrived at by an allocation, equation 3.8 has already been shown to be true. Hence H(m) is true, and so by induction it is true for all m, m = N,N-1,...,1 .

3.3 Time measurement, ergodicity

The previous two sections of this chapter introduced a model of storage allocation, and showed that it is a finite state discrete time Markov chain if certain further defined conditions are made. This section examines how the model behaves as time passes. The assumptions or conditions of the previous section guarantee that events occur randomly at a constant rate independently of the states entered. Event time T, the vector $\underline{\pi}(T)$ of state probabilities, the i-th component of which is the probability of being in state i at time T, and the transition probability matrix P are all introduced. Section 3.3.3 shows that under normal circumstances the Markov chain is ergodic. An important consequence of this is that the steady state (or equilibrium, or limiting) probabilities are uniquely determined by the steady state equation, 3.11 below, so that whatever state the model starts in this state probability vector $\underline{\pi}(T)$ will always converge to a unique "steady state" or equilibrium vector $\underline{\pi}$. A counter example of unusual conditions in which the chain is not irreducible is presented in section 3.3.5, to show that this possibility exists and can exceptionally occur.

3.3.1 Time measurement

The assumption that at any moment any of the remaining blocks in memory is equally likely to be the next to be deallocated, (4) of section 3.2.1, and the discrete time formulation which merely numbers successive events, avoid the use of any explicit timing information in the model. This significant saving in the total amount of information required to be kept, and the added simplification of not needing to know about time to compute state transitions, follow conveniently from this assumption. As a substitute for real time, the integer variable T is used to count "event time", i.e. for any non-negative integer T it is possible to determine the state probabilities after T transitions have occurred from a given initial state at T = 0. It is worth repeating assumption (3) of section 3.2.1 that the distribution of real time t_e which elapses between transitions is assumed known, and events occur at a constant rate λ independently of whatever states the model may enter.

3.3.2 A brief summary of the basic theory of Markov chains; ergodicity

Discrete-time Markov chains are classified into various types according to their properties; see for example Feller (1968), chapter 15, or Seneta (1973), chapters 1 and 4. Basic Markov theory is well known and can be found in many other reference texts besides these. A very brief summary is included here merely to explain a few terms.

The property of being able to reach any state from any other after sufficiently many transitions as are necessary, is called irreducibility. The states of an irreducible chain can not be separated into groups such that the states in one group can not eventually be reached from those in another group. Reducible Markov chains can be split into such groups however, and the groups can then be studied separately as chains distinct from each other. Periodicity is the property of only being able to reach one state from another after numbers of transitions which, apart from a constant of addition, are multiples of an integer (called the period, or cycle index) greater than unity; for example if state j can only ever possibly be reached from state i after 1, 4, 7, 10, 13, 16,... transitions. If one state of an irreducible chain is periodic, then so must all the others be, and with the same period. Periodic Markov chains can also be split into groups by equal periodic position, so that all the states in each group can only occur at the same place in the period. Markov chains with a finite number of states which are irreducible and non-periodic are called ergodic. (Ergodicity is usually defined in terms of non-periodicity, irreducibility and persistence; finite chains which are non-periodic and irreducible are automatically persistent.) The basic Markov theory shows that ergodic chains are "well behaved" in the sense that, irrespective of the starting state, the probabilities of being in any of the states must eventually converge after sufficiently many transitions to a set of uniquely determined values called the steady state (or equilibrium) probabilities.

Thus it is enough to know in the present case that the Markov chain representing the storage allocation model is finite, non-periodic and irreducible, to know that its behaviour as time passes must so converge. It is interesting to know this because useful quantities such as the expected storage utilisation and expected fragmentation must also then converge to values which can be determined from the steady state probabilities.

3.3.3 Proof that the Markov chain storage allocation model is ergodic

A sufficient condition for this to be so is that the probability of a request for a block of one word is non-zero, i.e. $r_1 > 0$, and the present proof is based on this assumption. An example which the reader might like to see first and which may make the details clearer is given in the next section after this proof (which does not depend on the example).

Consider the particular state in which the memory of N words is completely filled by N 1-word blocks, and call it F (for full). By section 3.2.4, the size q of the request at the head of the queue in state F retains the (r_n) distribution, (equation 3.4), the same as all the other subsequent queued requests.

1) F can be reached from any other state in a finite number of steps. This is because in any state, the N requests after the first at the head of the queue can all be for 1-word blocks with non-zero probability, and because also with non-zero probability any allocated blocks of more than one word in any state can be successively deallocated in preference to any of the 1-word blocks which may be in memory. Since requests must always be satisfied as long as at least one sufficiently large gap exists, any allocation algorithm will in these circumstances have no choice but to continue allocating the N 1-word blocks once the first request has been allocated, while at various time intervals any larger blocks initially present are removed, and this process must eventually reach state F after no more than N+1 blocks have been allocated.

2) Any "possible" state (configuration of memory) can be reached from F in a finite number of steps, whatever allocation algorithm is used. A "possible" state is any containing no allocated block of a size i with zero

request probability $r_i = 0$, (that is it only has blocks of sizes which can occur in the queue), and containing no gap as large as the size, (max) say, of the largest possible request for which $r_{(max)} > 0$ (this is so that there can be a request, (for example for (max) words), waiting at the head of the queue for which no gap is sufficiently large). For any "possible" state, the requests at the front of the queue in state F can be for just the number and sizes of blocks which occur in it, in some order, followed by a request for a block of size (max), the largest which can occur. Then in succession and with non-zero probability, the 1-word blocks present in state F in the memory positions where the first queue request is allocated in the given "possible" state can be deallocated; when the last such 1-word block is removed, the allocation algorithm has no choice but to allocate the first request at this position. It is possible for this sequence to then be repeated for the succeeding requests until all the blocks which are in memory in the given state have been allocated. Finally, if there are any, it is possible for the 1-word blocks at the remaining memory locations where there are gaps in the given state to be deallocated. All this can occur with non-zero probability. The result when the last unwanted 1-word block is removed will be the given "possible" state. The request for (max) words will not be allocated immediately when the last unwanted block is removed, since by the above definition of "possible" there is no gap large enough for it. This choice of queue sizes and sequence of deallocations and allocations starting from F has a non-zero probability and is certain to take no more than 2N transitions since each word of the memory is involved in at most one deallocation and one allocation.

3) If F is reachable from any state in k transitions, then since subsequent requests can also be for 1 word, F is reachable in (k+1), (k+2), ... transitions.

It follows from the definition of a "possible" state that the set of "possible" states is closed and absorbing, since they can only make transitions to other "possible" states. It is not hard to prove that all other states can only be transient, since they can only be not "possible" because they contain blocks of sizes with zero request probabilities, all of which must eventually be deallocated with certain probability.

(1) and (2) together prove that the set of "possible" states is irreducible, so that there is just this one irreducible absorbing subchain, and (3) proves that they are aperiodic. Since the subchain is finite, it must therefore be ergodic; see for example Feller (1968), chapter 15, or Seneta (1973), section 4.2.

3.3.4 An example to illustrate the foregoing proof of ergodicity

An example of the argument to show that F can be reached from any other state is illustrated by figure 3.2. Part (a) of this figure represents one such other state, and the succeeding parts (b) - (f) show how it is possible to reach F after, in this case, five complete transitions.

In part (a), a memory of 20 words contains one 4-word block, two 3-word blocks, a 2-word and a 1-word block. The largest gap is 4 words long, and the request at the head of the queue is for 7 words. Subsequent requests in the queue are for 1-word blocks.

In part (b), the 4-word request has been deallocated (with probability 1/5). This action begins a transition. The largest gap is now 8 words long, so the allocation algorithm can choose from two places to put the 7-word request. It does not matter which it chooses.

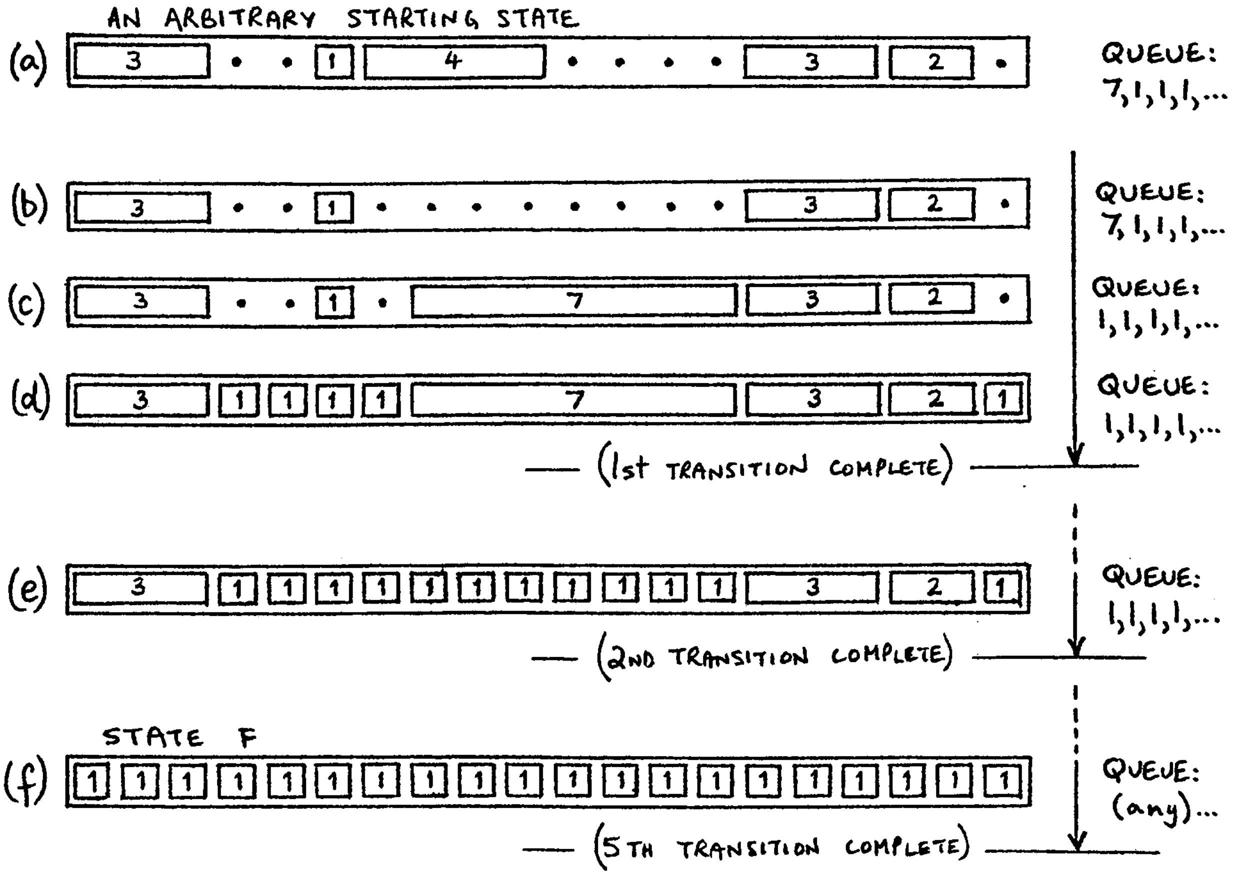


Figure 3.2 Illustration of ergodicity proof: From any state, to F

In part (c), the 7-word request has been allocated. The first request in the queue is now for a 1-word block, so the allocation process will continue by allocating it.

In part (d), all the available gaps have been filled with 1-word blocks and there are no remaining gaps. The first transition is complete.

In part (e), the 7-word block has been deallocated (with probability 1/9) and the resulting gap immediately filled with 1-word blocks from the queue, completing the second transition. This process of choosing successively to deallocate the 2-word and two 3-word blocks then continues, causing three more transitions to occur.

In part (f), all the larger blocks have been deallocated and replaced by

1-word blocks. State F has been reached. The first request in the queue can be for any of the possible sizes.

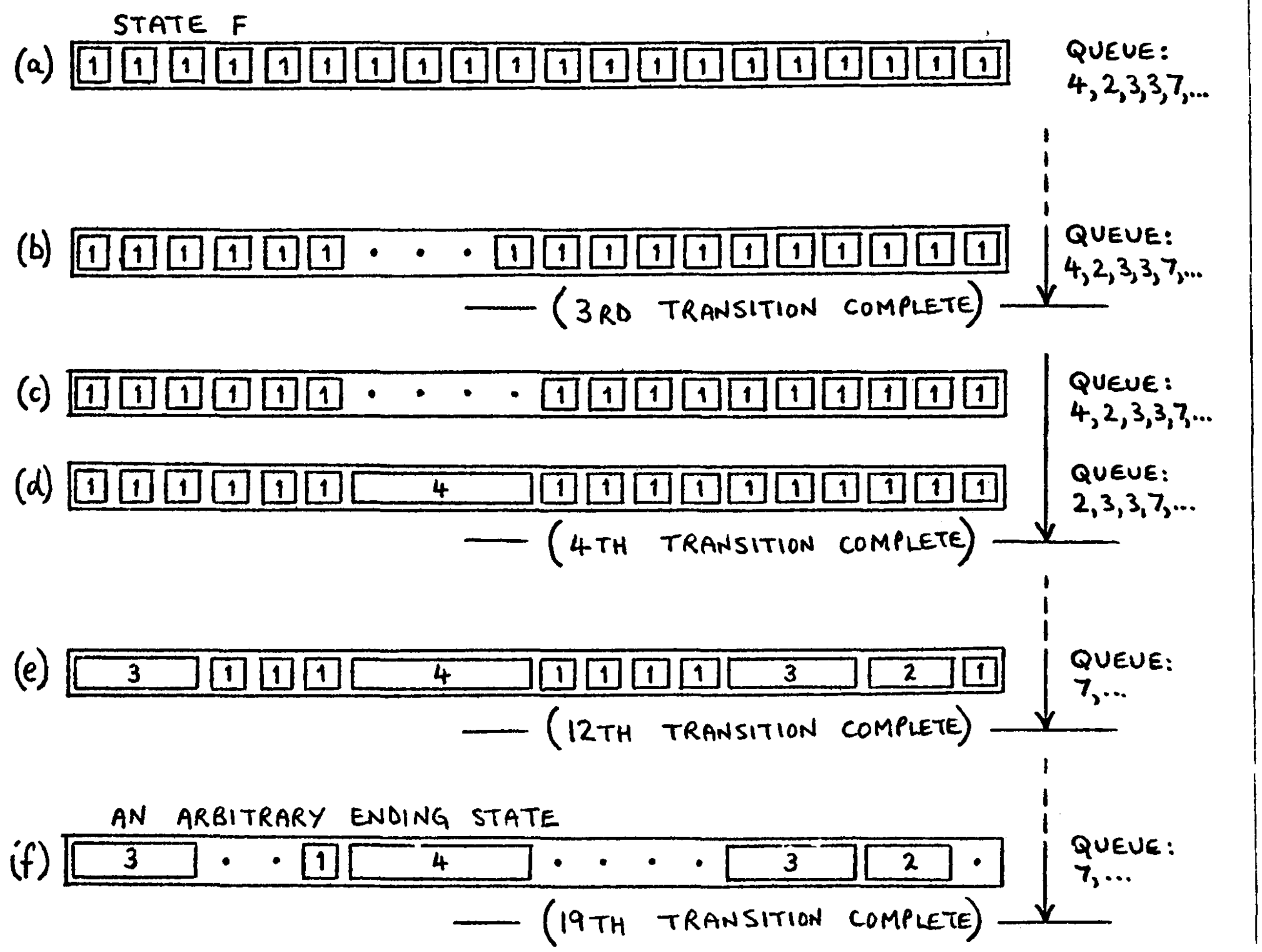

STATE F
(a)
QUEUE: 4,2,3,3,7,...
(b)
QUEUE: 4,2,3,3,7,...
(3RD TRANSITION COMPLETE)
(c)
QUEUE: 4,2,3,3,7,...
(d)
QUEUE: 2,3,3,7,...
(4TH TRANSITION COMPLETE)
(e)
QUEUE: 7,...
(12TH TRANSITION COMPLETE)
AN ARBITRARY ENDING STATE
(f)
QUEUE: 7,...
(19TH TRANSITION COMPLETE)

Figure 3.3 Illustration of ergodicity proof: From F to any state

An example to show how any "possible" state can be reached from state F, is shown in figure 3.3. The state to be reached is the original starting state shown in figure 3.2(a).

In figure 3.3(a), the memory is filled with 1-word blocks, and the queue contains successive requests for 4, 2, 3, 3, 7 words.

In part (b), three of the 1-word blocks have been deallocated one transition at a time, with successive probabilities 1/20, 1/19, 1/18. The resulting gap is still too small to allocate the first queue request for 4 words.

In part (c), a fourth 1-word block has been deallocated. The four words form a gap in exactly the position that a 4-word block occupies in the final state.

In part (d), the allocation algorithm has had no choice but to put the first request in this gap, and a 2-word request is now first in the queue. This process of deallocating the 1-word blocks one at a time from the positions where the larger blocks are to be, is continued. At each stage, the allocation algorithm is allowed no choice.

In part (e), all of the larger blocks are in place, and the first queue request is for 7 words.

In part (f), seven of the remaining eight 1-word blocks have been deallocated in some order in seven successive transitions. No allocations have taken place, and the final state has been reached.

3.3.5 A reducible example of the Markov storage allocation model

The Markov model is not always irreducible, if there are no requests for 1-word blocks and if the allocation algorithm is chosen rather unusually. Perhaps the simplest example is shown in figure 3.4, of a five word memory and a queue of requests all of which are for 2-word blocks. That is, $N = 5$ and $r_2 = 1$, $r_1 = r_3 = r_4 = r_5 = 0$. It is not hard to see that between transitions there must always be exactly two blocks in memory, in one of three possible configurations. Since blocks are deallocated one at a time, the memory can

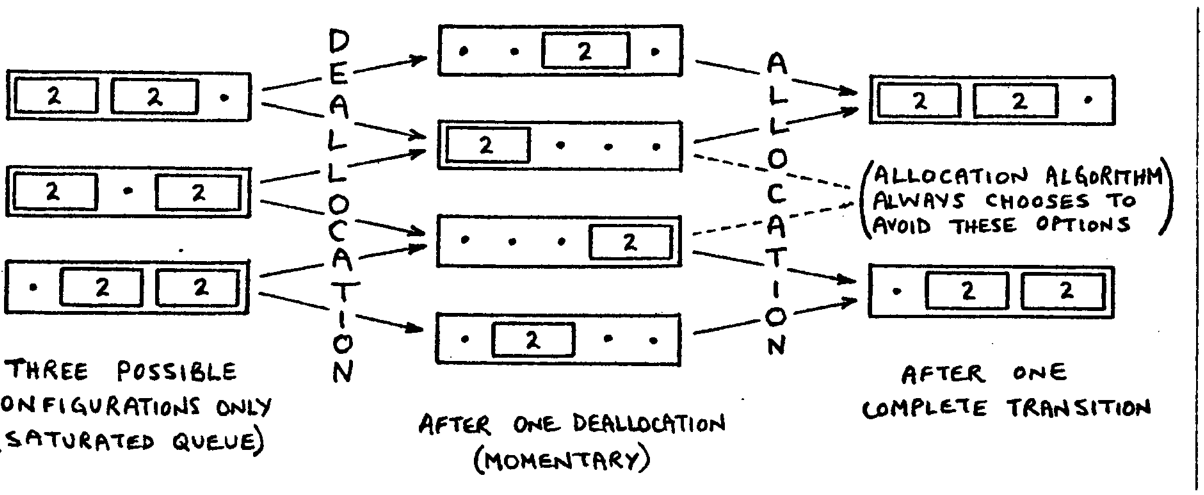

DEALLOCATION
ALLOCATION
(ALLOCATION ALGORITHM ALWAYS CHOOSES TO AVOID THESE OPTIONS)
THREE POSSIBLE CONFIGURATIONS ONLY (SATURATED QUEUE)
AFTER ONE DEALLOCATION (MOMENTARY)
AFTER ONE COMPLETE TRANSITION

Figure 3.4 Transitions in an example of a reducible model

never be empty after the start, not even instantaneously. In this example, the allocation algorithm is chosen so that it always allocates a new request starting on a word of the same even-odd parity as the existing allocated block. Of the three possible configurations, one is transient with this algorithm, and the other two form closed groups of one state each. Once one of these two states has been reached, as it must be at the latest after just one transition, the allocation algorithm ensures that the Markov chain never leaves it.

This example can easily be extended for any N-word memory, where $N \geq 5$. The request distribution is chosen to allow only even-sized blocks of sizes up to but always less than $(N+2)/3$ words. The memory can consequently never be completely emptied, and so there must always be at least one block present to serve as an indicator of the even-odd parity of the previous configuration, which the allocation algorithm is chosen to preserve.

The existence of this possibility of non-ergodicity is something of a surprise, but it is surely most unlikely to occur by accident in practice.

The style of this proof and the counter-example is reminiscent of the perverseness of the attacker and cunning of the defender which Robson finds it necessary to assume in his studies, referred to in chapter 2 section 2.5. These roles are reversed here as now it is the allocation algorithm (defender) which has to be forced into sufficiently reasonable or ergodic behaviour. The author has the intuitive feeling that, even without requests for 1 word, the existence of enough relatively prime request sizes should be sufficient ammunition for the attacker. For example, all the integers greater than 1 can be produced by adding 2 and 3 (2,3,2+2,2+3,3+3,2+2+3, etc.). However, remembering Matzke's warning from the end of chapter 2, this intuition has yet to be proved and it may be wrong.

3.4 Transition matrix P, state probabilities π

This chapter has established the Markov nature of the storage allocation model being studied, and ends by introducing the most important object which is the key to the subsequent analysis. Let the transition probability matrix $P = (p_{ij})$ of the Markov model be defined:

$$p_{ij} = \begin{array}{c}\text{(conditional)}\\ \text{Probability}\end{array}\left[\begin{array}{c}\text{model will make}\\ \text{a transition}\\ \text{to state } j\end{array}\middle|\begin{array}{c}\text{model}\\ \text{is in}\\ \text{state } i\end{array}\right], \text{ for } 1 \leq i,j \leq S' \quad .... 3.10$$

where $S' = S'(N)$ is the number of states (non-empty memory configurations) for a given memory size N words. The quantity $S'(N)$, and possible orderings of the states, are discussed in more detail in chapter 4. Each p_{ij} is independent of time (the Markov chain has stationary transition probabilities, or is homogeneous) because of assumptions (4) and (5) of section 3.2.1. P is finite, square, stochastic (row sums are all unity, all elements are real and non-negative) and primitive (irreducible and aperiodic). The Perron-Frobenius theorem (see for example Seneta (1973), chapters 1 and 4, or Gantmacher (1959) chapter 13) therefore applies to show that P has a real, non-repeated eigenvalue 1 which is dominant, that is, all other eigenvalues λ of P have modulus $|\lambda| < 1$. The left eigenvector $\underline{\pi} = (\pi_i)$ of P corresponding to this eigenvalue exists, has positive elements $\pi_i > 0$, $i = 1,\dots,S'$, and is unique if it is normalised so that its elements sum to 1. $\underline{\pi}$ is the unique steady state probability distribution of P, satisfying:

$$\underline{\pi} = \underline{\pi}\, P \qquad \dots\dots 3.11$$

If the normalised row vector $\underline{\pi}(T) = (\pi_i(T))$ of state probabilities is defined:

$$\pi_i(T) = \text{Probability[model is in state } i \text{ at time } T] , \quad \text{for } i = 1,\dots,S' \qquad \dots\dots 3.12$$

then

$$\underline{\pi}(T+1) = \underline{\pi}(T)\, P \qquad \dots\dots 3.13$$

and by the ergodic theorem for primitive Markov chains,

$$\lim_{T\to\infty} \underline{\pi}(T) = \underline{\pi} \qquad \dots\dots 3.14$$

Equation 3.13 is the basis of the well-known power method of von Mises, also known as the method of iterated vectors. For an extended exposition of the power method, see for example Bodewig (1956), page 231 onwards, and especially pages 250-254 which are particularly critical, pointing out that convergence using this method can on occasion be exceedingly slow, depending on the eigenvalues of the matrix. This topic will be returned to in chapter 6 sections 6.3 and 6.5 and in chapter 7 section 7.1, where happily it will appear that this pessimism is unfounded for the present family of transition matrices P being considered. The rate of convergence is geometrically rapid and depends on the largest absolute value magnitude $|\lambda|$ of the eigenvalues of P not equal to 1, the above limit being approached as $|\lambda|^T \to 0$. This convergence to $\underline{\pi}$ is independent of the starting state or states, i.e. $\underline{\pi}(0)$ may be any arbitrary (non-negative, normalised) probability row vector.

Chapter 4 : Preliminary analysis of the storage allocation model

Before proceeding with the algebraic analysis of the Markov model of storage allocation defined in the last chapter, it is useful to make a preliminary survey of the model and this is done in this chapter. The number of states in the model depends on the memory size N, and it is the number of different possible ways that there are of arranging any number of blocks and intervening gaps of arbitrary sizes greater than zero into an N-word memory. This number turns out to be just the 2N-th term from the Fibonacci sequence, which increases exponentially with N. Any straightforward attempt to calculate the transition matrix and steady state eigenvector for general N is therefore quickly limited, whether directly or iteratively by the power method for instance. If the definition of the model is changed to eliminate external fragmentation by allowing relocation (compaction) of the blocks in memory whenever necessary, it has been found possible to obtain direct expressions for the expected steady state storage utilisation with relocation, and an example is given. The performance of this model gives an upper bound for the expected storage utilisation in the corresponding non-relocating model.

4.1 Size of problem, and empty state

A major difficulty facing any simple analysis, such as the implementation of the power method described in chapter 6, is the rapidly increasing number of states in the Markov chain as the memory size N increases. It is proved quite easily below that the number S(N) of different configurations of an N-word memory into allocated blocks with possible intervening gaps is given by

$$S(N) = f_{2N} , \quad N = 1,2,... \qquad .... \ 4.1$$

where f_{2N} is the 2N-th Fibonacci number,

$$f_0 = f_1 = 1 \; ; \quad f_n = f_{n-1} + f_{n-2} \, , \quad n = 2,3,\ldots \, . \qquad \ldots. \; 4.2$$

S(N) includes the empty configuration (no blocks in memory). In section 3.3.3 it was shown that in the general case where the request probability distribution (r_n) is strictly positive, every non-empty configuration is a possible state, so that the number of possible states S'(N) is just one less than this:

$$S'(N) = S(N) - 1 \, . \qquad \ldots. \; 4.3$$

In chapter 5 the empty configuration will necessarily be included as it is a possible intermediate stage during a transition. S(N) is an exponentially increasing function of N:

$$\begin{aligned} S(N) &= f_{2N} \\ &= 0.72\ldots \times (2.6\ldots)^N \quad \text{approximately, as N increases} \qquad \ldots. \; 4.4 \end{aligned}$$

Values of S(N) for increasing N are shown in table 4.1 below.

4.1.1 Proof that the number of memory configurations is f_{2N}

This simple proof proceeds by induction on N. By inspection,

$$S(1) = f_2 = 2 \, ,$$

for a memory of one word is either empty or occupied by a single block. Assume inductively that

$$S(n) = f_{2n} \, , \quad n = 1,2,\ldots,N$$

and consider n = N+1, a memory of N+1 words. There are two contributions to the total, S(N+1):

1) The first word is unoccupied. There are then S(N) possible configurations of words (2,3,...,N+1) considered as an N-word memory. When the empty first word is added, N-memory configurations which start with a block at word 2 of the (N+1)-memory are unchanged, while those which start with a gap of length g produce (N+1)-memory configurations which start with a gap of length (g+1) at word 1.

2) The first word is occupied, by a block of length b, b = 1,2,...,N+1. For b < N+1 there are then S(N+1-b) possible configurations of the remaining words, and for b = N+1 the contribution to S(N+1) is a single possibility.

Thus altogether,

$$\begin{aligned} S(N+1) &= S(N) + S(N) + \ldots + S(2) + S(1) + 1 \\ &= 2f_{2N} + f_{2N-2} + \ldots + f_4 + f_2 + f_1 \\ &= f_{2N+2}\text{ , as required.} \end{aligned}$$

4.2 Illustration of increasing transition matrix size

Figures 4.1 to 4.7 display examples of the transition probability matrices for models of memory size up to N = 6 words. Figures 4.1 to 4.4 are the transition matrices for models with N = 1, 2, 3, 4 words respectively. The states are represented diagrammatically down the left hand side, ("from" states) and along the top ("to" states) of each of the matrices. Thus the first state shown in each case is that in which the memory is completely filled with N one-word blocks, the state which was called F in section 3.3.3 in fact. All the possible configurations are present in the diagram, and for

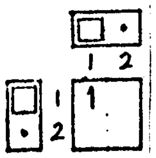


Figure 4.1 Transition matrix P for N=1

Uniform distribution, first fit

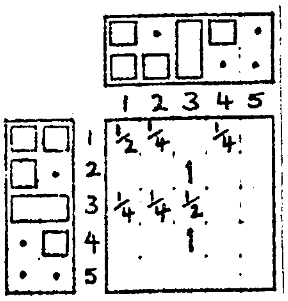


Figure 4.2 Transition matrix P for N=2

Uniform distribution, first fit

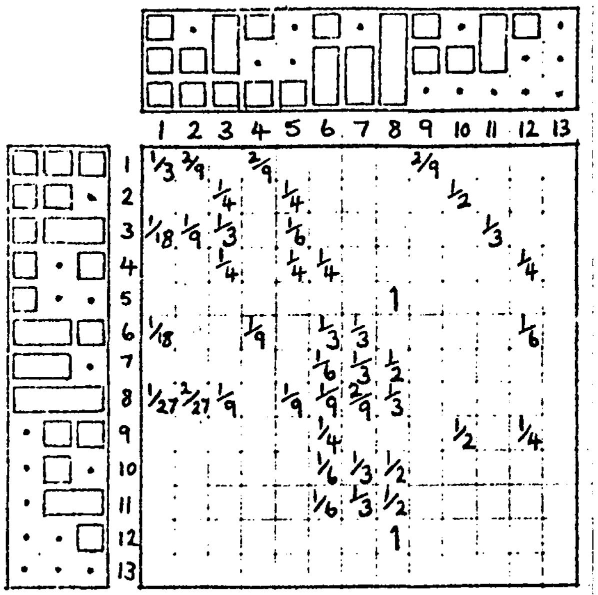


Figure 4.3 Transition matrix P for N=3 Uniform distribution, first fit

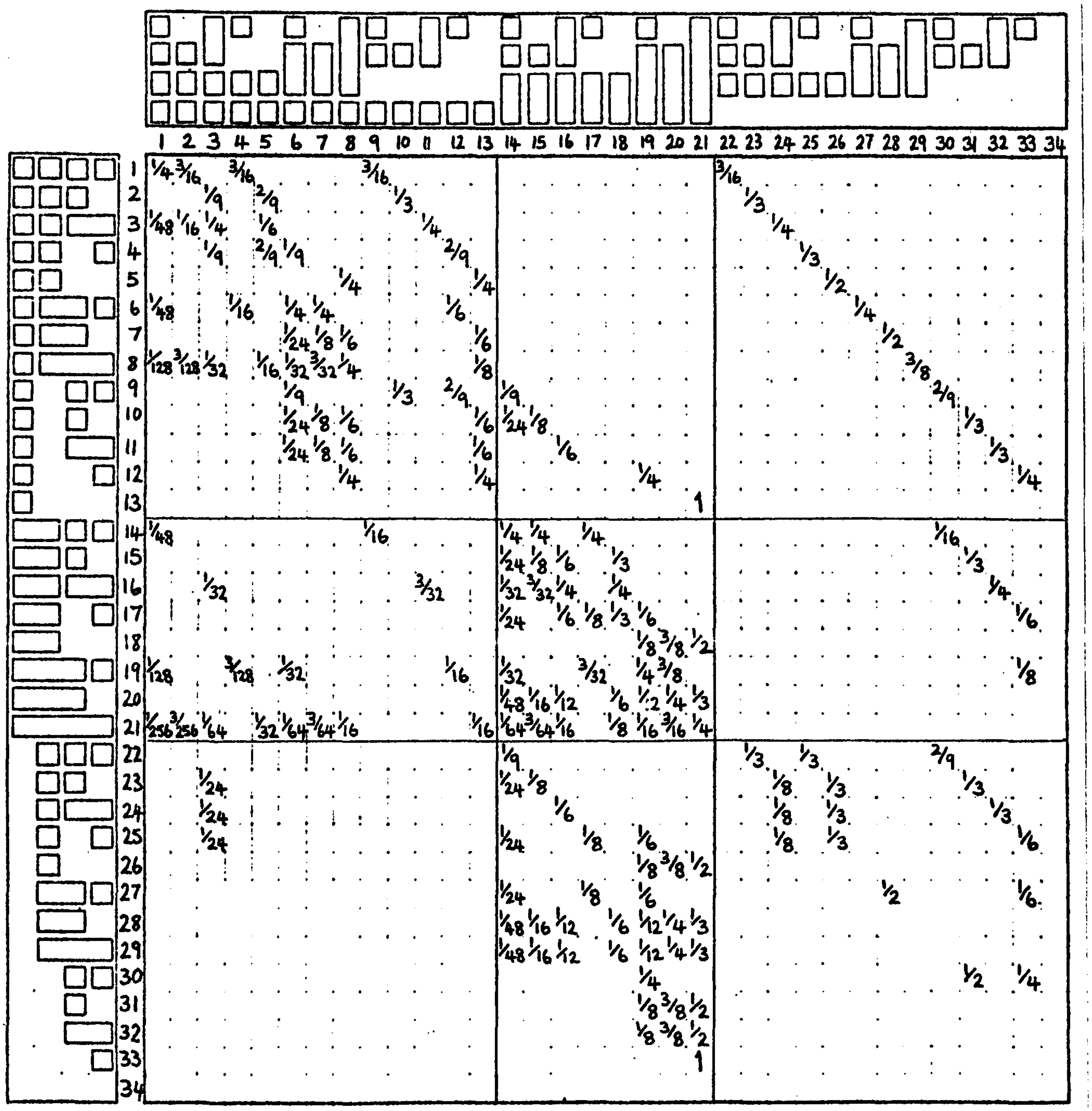


Figure 4.4 Transition matrix P for N=4 Uniform distribution, first fit

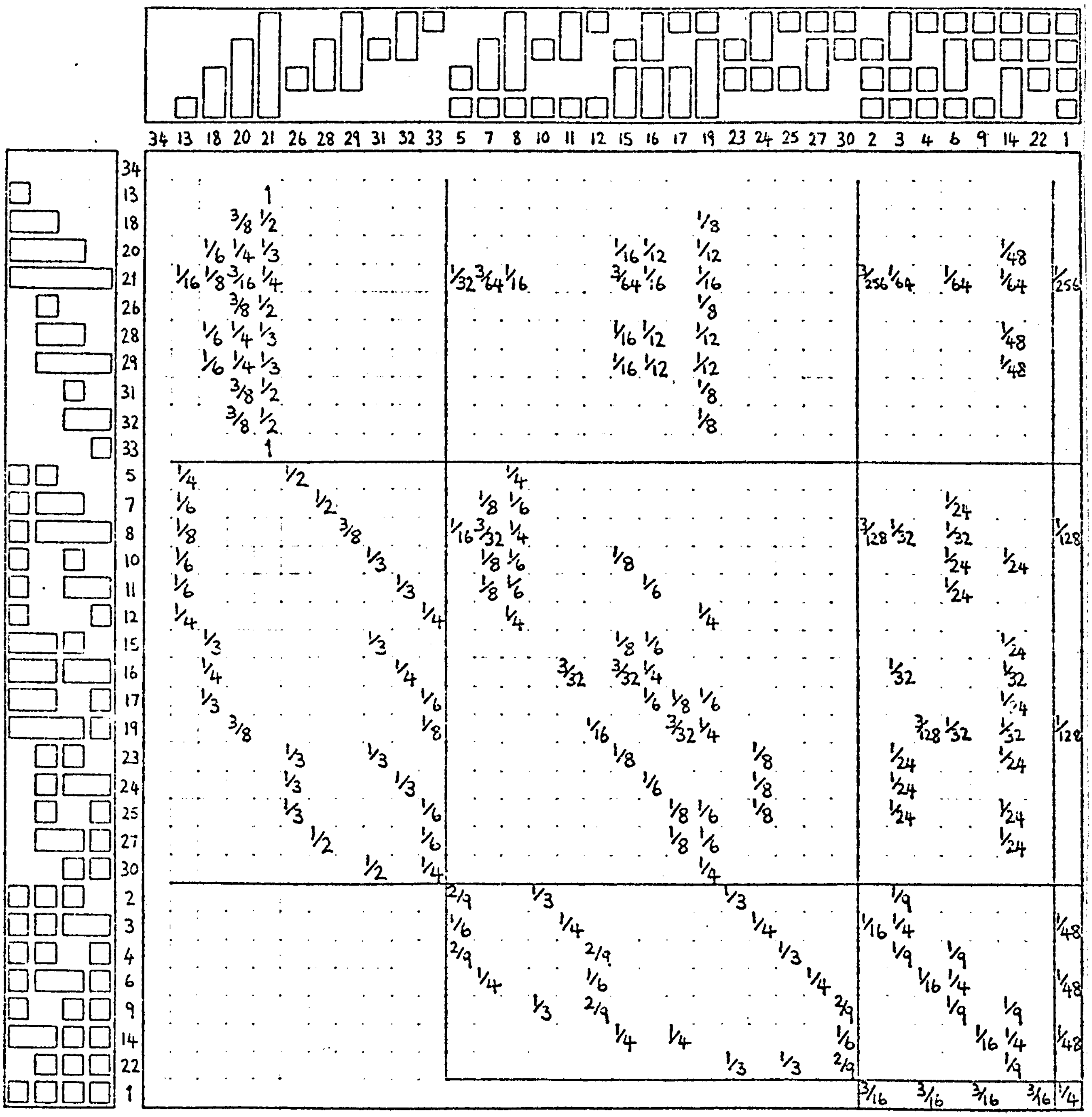


Figure 4.5 Transition matrix P for N=4 Uniform distribution, first fit

(A different state ordering to that in figure 4.4 has been used)

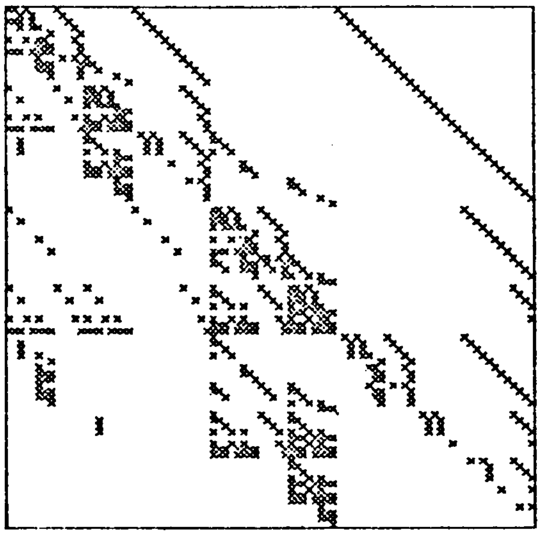


Figure 4.6 Silhouette, or incidence matrix, of transition probability matrix P (N=5, best fit)

completeness' sake the empty configuration has been included but of course with zero row and column as it can only occur momentarily as an intermediate stage of a transition. Except for the empty state, the elements of any row of these matrices are the probabilities that the state shown at the left of the row will make a transition to the respective state shown above the columns. Non-zero probabilities occur wherever a transition is possible, and blank spaces indicate zero probabilities elsewhere.

From section 3.2.2 a transition from any state consists of a deallocation of just one of the blocks in the state, followed by (possibly zero) allocations from the queue. The request at the head of the queue cannot be for a block of words which could have been allocated before the deallocation occurred. In these examples, the probability distribution of request size is the uniform distribution, for example in figure 4.4, N = 4 : $r_1 = r_2 = r_3 = r_4 = 1/4$. The "first fit" allocation algorithm is used, which allocates a request into the left hand end of the leftmost gap large enough to take it. In such small memories there are very few cases where a choice is

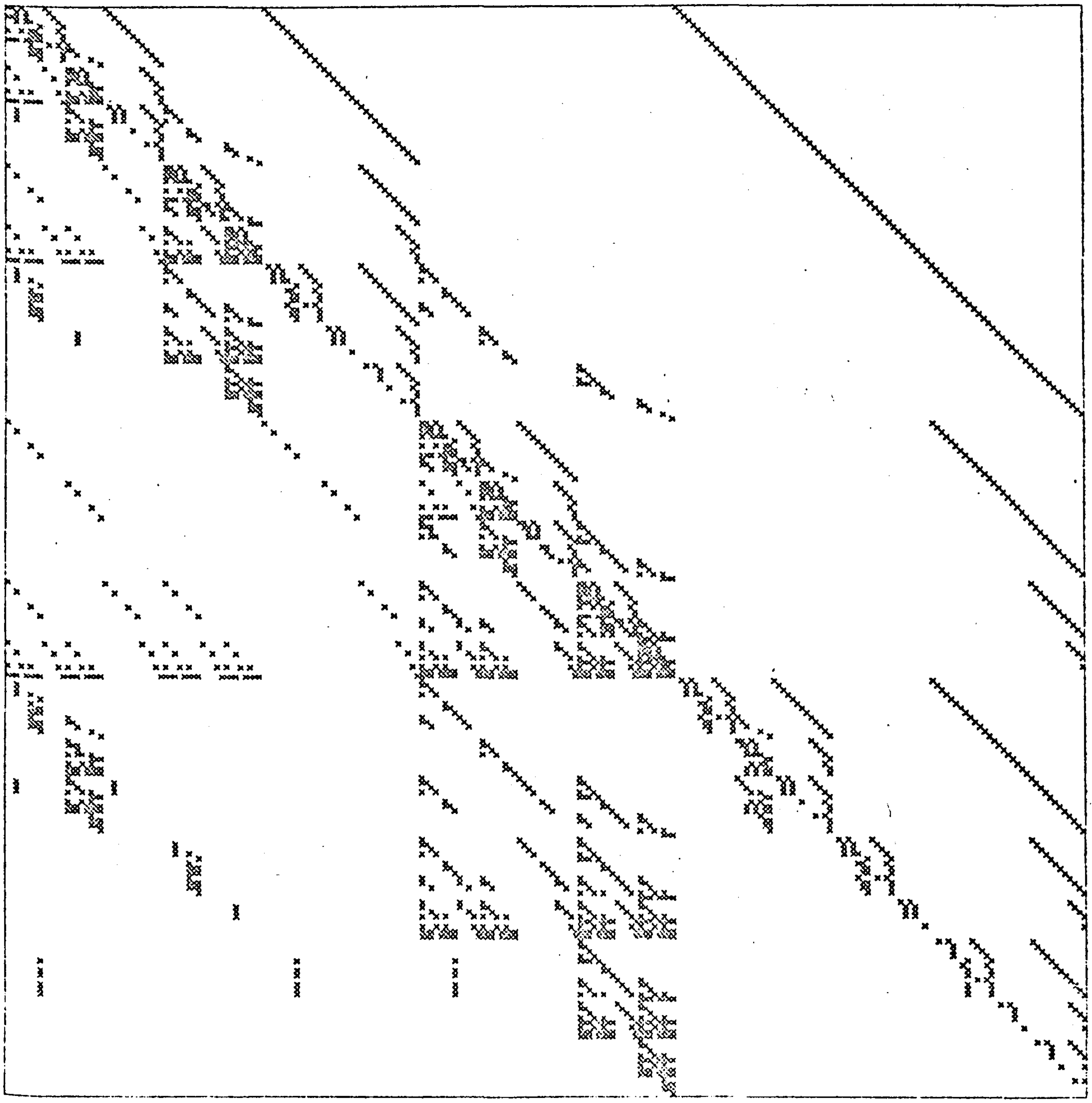


Figure 4.7 Silhouette, or incidence matrix, of transition probability matrix P

(N=6, best fit)

possible ("small" memories might in fact be large ones being allocated in large quanta, of course, and section 6.5.4 presents just such an example of this, showing the use to which this observation can be put).

Figure 4.5 shows the same transition matrix as in figure 4.4, for N = 4 words, but with a different ordering of the states. The state ordering which has been used in figures 4.1 to 4.4 and throughout most of the rest of this thesis is described below. The example of figure 4.5 is merely to show that other orderings are possible and may be no more or less arbitrary.

Figures 4.6 and 4.7 continue the sequence of figures 4.1 to 4.4 for N = 5 and 6 respectively, but now the matrices are too large to be able to show the individual non-zero elements, so their positions only have been marked. All models using the best fit algorithm and in which the request probabilities are non-zero have this same shape, or incidence matrix.

Two features are very noticeable from these examples. Firstly, the elements seem to fall into a marked repetitive pattern, both in position and (although it is less evident in the figures) in value, the pattern repeating within the same matrix and also from one matrix to the next in the sequence. This repetitive pattern is the key to the analysis in chapter 5. It is there because the transition matrix is a combination of much simpler matrices whose non-zero elements form some obvious and simple recursive patterns. Secondly, the transition matrix is sparse, and this is because any individual state can make a transition to relatively few of the other states, (although given enough transition steps, eventually to them all by ergodicity). This sparsity is used in the implementations of the model for small memory sizes described in chapter 6, in the attempt to make "small" become as large as possible with the computational resources available.

4.3 State ordering

In principle, the order in which states are indexed is irrelevant as the transition matrix is essentially unchanged whatever order its rows and columns are permuted into. However it has been found desirable to use an order which makes certain features of the component parts of the transition matrix easily apparent, such as upper or lower triangularity and the existence of diagonal submatrices. The ordering which so far has been found to do this as well as any other is illustrated by figures 4.1 to 4.4, and as the analysis of chapter 5 is described in terms of this ordering, it is defined here so that it is not in doubt. As pointed out in the last section, it is arbitrary in the sense that any other ordering which allows the same or other properties of the transition matrix to be studied would be equally as good.

It should be apparent from figures 4.1 to 4.4 that in the ordering which has been used, shorter blocks take precedence over long ones, gaps having the lowest precedence. Specifically, any configuration can be represented as a sequence of (gap block) pairs, for example in figure 4.4, N = 4, state 8 = (0 1)(0 3), state 17 = (0 2)(1 1), state 33 = (3 1). States with a final gap of any length can also be described this way, for example state 23 in figure 4.4 = (1 1)(0 1), state 31 = (2 1). The length of the final gap can be found by subtracting from the memory size N. The pairs are given the following order of precedence:

$$\begin{aligned}
&\text{if } g_1 < g_2 && \text{then } (g_1\ b_1) < (g_2\ b_2) \\
&\text{if } g_1 = g_2\ ,\ b_1 < b_2 && \text{then } (g_1\ b_1) < (g_2\ b_2) \qquad \ldots.\ 4.5 \\
&\text{for all } g\ ,\ b\ : && (g\ b) < \text{no pair (final gap)}
\end{aligned}$$

For N = 4, the order of precedence of the pairs is

(01),(02),(03),(04),(11),(12),(13),(21),(22),(31),(no pair)

The ordering of states can now be defined. To decide the relative order of two given states, their (gap block) pair representations are compared pair with pair from left to right until an inequality is reached, as it must be if the memory configurations are different. The first such pair inequality is used to decide the order of the states. For example, for N = 4,
(0 1)(0 1)(0 2) occurs before (0 1)(0 1)(1 1) because (0 2) < (1 1), and
(0 1)(0 1)(1 1) is before (0 1)(0 1) because (1 1) < (no pair).

This definition is trivially a well-ordering. If states a, b, c are related such that a < b and b < c by this ordering, then let the first non-equal pairs in the pair-representations of a and b be the i-th, and the j-th pairs the first unequal between b and c. Then if k = minimum(i,j) the k-th pairs must be the first unequal between a and c, and they define that a < c. The ordering is also strict, two states only comparing equal by this order if their memory configurations are equal, that is, if they are the same state.

Chapter 6, section 6.4 describes an algorithm for computing the (gap block) representation of a state given its index, and vice versa.

4.4 Analysis of a model in which relocation is allowed

If the model as described in section 3.1 is modified so that allocated blocks in memory are moved around to collect together the free space whenever necessary, then the problem of calculating how much memory is used on average becomes simpler, and closed form solutions are possible in individual cases.

The reason for wanting to study the relocating model, besides the practical fact that it is much easier than the fragmented non-relocating version, is for a comparison with the fragmented model. Randell (1969) used this comparison in his definition of external fragmentation as the difference in storage utilisation between two otherwise identical models, one with relocation and one without. It is obvious that the storage utilisation in the relocating model is an upper bound for that in the fragmented case, for anything that can be allocated in a fragmented memory must also fit if all the blocks can be pushed together as required to bring the free space into one gap. The idea of an allocation algorithm becomes irrelevant in this model; it makes no difference where a block is, and only its size matters. With a saturated queue the only external fragmentation possible will be any remaining words left over which are insufficient for the size of the request currently at the head of the queue. Thus its simplicity and the provision of an upper bound to any possible performance with fragmentation, make the relocating model interesting and worthwhile to study.

As an addition to the external fragmentation, following the simulation experiments of Randell, it is possible to include in the calculation the effect of rounding up requests to the nearest integral multiple of a fixed quantum size, and the internal fragmentation so introduced can be calculated. Two different distributions of request sizes, uniform and negative exponential, will be presented as examples. Since this relocation model was analysed in detail in Betteridge (1974) and the analysis is in fact very straightforward consisting mainly of several easy but lengthy algebraic reductions, the algebra will be slightly shortened and summarised here.

4.4.1 Steady state fragmentation in the relocating model

The first step to obtain information about the steady state behaviour when relocation is allowed is to show that, whatever the request size distribution is, if the memory is allowed to fill up from empty then the probability of any resulting configuration is the same as its probability of occurrence in equilibrium, and this was proved rigorously as a theorem in Betteridge (1974). With this established, the steady state behaviour becomes much easier to discover as the problem reduces to the simpler one of deciding what happens on the average when the memory is allowed to fill up from empty until a request is reached which will not fit.

It is convenient to identify the "filling up" process as being the set of possible transitions from one particular state E in which the memory is completely filled by a block of N words. The first action to occur during any transition from this state must result in a completely empty memory (hence the choice of the symbol E) and the size of the request waiting at the head of the queue retains the (r_n) probability distribution. Two simple observations are justified in the proof which is given in the 1974 paper. Firstly, filling the memory up from empty by discarding the first request and starting from the second request instead, must give all the same transition probabilities as starting with the first request. Secondly, in getting to some state i in two transitions, after the first transition from E the first allocated request in any intermediate state j may as well be deallocated as any other, since the probability distribution of the sizes of the remaining blocks in memory after the deallocation will be the same whichever of the blocks in state j is deallocated. But these two observations show that the probability of getting to any state, i say, from E in two transitions is the same as the probability of reaching it in one transition and this is also proved in the theorem. Then for any i,

$$\text{Prob[from E to i]} = \sum_{\text{all } j} \text{Prob[from E to j]} \times \text{Prob[from j to i]} \qquad \text{.... 4.6}$$

But this shows that the probabilities Prob[E to i] satisfy the steady state equation and so by uniqueness they are the steady state probabilities, since the probabilities Prob[j to i] are just the elements of the transition probability matrix of this relocating model.

4.4.2 Two example request distributions for the relocation analysis

The uniform request size distribution is easily defined:

$$r_n(\text{uniform}) = 1/N \quad , n = 1,\dots,N \qquad \text{.... 4.7}$$

That is, for any request arriving in the queue, all possible block sizes from the smallest (1) to the largest (N) are equally likely. The mean E_{uniform} of this distribution, or average requested block size, is evidently:

$$E_{\text{uniform}} = (N+1)/2 \qquad \text{.... 4.8}$$

The negative exponential request size distribution that has been used in the following example calculation, is defined:

$$r_n(\text{exponential}) = (r_N)^n \quad , n = 1,\dots,N \qquad \text{.... 4.9}$$

The value r_N, which depends on N, must of course be chosen so that

$$\Sigma r_n = 1 \quad \text{and} \quad 0 \leq r_n \leq 1 \quad , \quad n = 1,\dots,N$$

and this implies that r_N is the root of the equation

$$r^{N+1} - 2r + 1 = 0 \qquad \text{.... 4.10}$$

lying in the range $0.5 < r_N < 1$ (for $N > 1$). The average requested block size E_{exp} of this distribution may be evaluated:

$$E_{exp} = \sum_{n=1}^{N} n(r_N)^n = 2N - \left(\frac{N-1}{1-r_N}\right) \qquad \text{.... 4.11}$$

Values of r_N and E_{exp} are given in table 4.1 below; their respective limits as N increases are 0.5 and 2.

It is possible to perform all of the following analysis for a more general form of exponential request size distribution:

$$r_n(\text{general exponential}) = ar^{n-1} \qquad , n = 1,\ldots,N \qquad \text{.... 4.12}$$

Here, a and r must be chosen to satisfy

$$0 < a,r \leq 1 \quad \text{and} \quad \sum_{n=1}^{N} ar^{n-1} = 1 \qquad \text{.... 4.13}$$

which gives

$$ar^N - (a+r) + 1 = 0 \qquad \text{.... 4.14}$$

The algebra which follows this is then more complicated, but closed form expressions may still be obtained. By setting $a = r$, the simpler version presented above is obtained. The reason for using this is that the one degree of freedom allowed in the choice of a and r allows the mean of the distribution to be varied while retaining the exponential characteristic. By making the starting value a approach 1/N from above, and r approach 1, the distribution becomes close to uniform. Conversely if a approaches 1 and r becomes very small, then most requests will be for 1 word and large requests will become correspondingly unlikely. It did not seem worthwhile to use this more general exponential distribution for the sizes of memory in the models which can presently be computed, and the above simpler version 4.9, 4.10 of the exponential distribution is the one that has been used.

4.4.3 External fragmentation in the relocating model

This will be considered in terms of the average utilisation U, the average fraction of memory occupied by requests. Then by the definition 2.1 in chapter 2,

average external fragmentation = 1 - average utilisation = 1 - U .

Whatever distribution of request sizes is used, the expected utilisation U(reloc) in the relocating model satisfies:

$$U(\text{reloc}) = \frac{1}{N}\times\sum_{i=1}^{N} i\times \text{Probability}\left[\begin{array}{c|c}\text{exactly } i \text{ words} & \text{state } E\\ \text{allocated} & \text{at } T=0\\ \text{at } T=1 & \end{array}\right] \qquad \text{.... 4.15}$$

where Probability[i words at T=1 | E at T=0]

$$= \sum_{n=1}^{i} \text{Probability}[i \text{ words allocated to } n \text{ blocks at } T=1 \mid E \text{ at } T=0] \qquad \text{.... 4.16}$$

$$= \sum_{n=1}^{i} \text{Probability}[\underline{r}_1+\underline{r}_2+\ldots+\underline{r}_n=i]\times\text{Probability}[\underline{r}_{n+1} > N-i] \qquad \text{.... 4.17}$$

and where $\underline{r}_1$, $\underline{r}_2$, ... are the random variables representing the sizes of successive requests in the queue in state E.

For the uniform distribution, it is not hard to show by induction (Betteridge (1974)) that

$$\text{Probability}[\underline{r}_1+\ldots+\underline{r}_n=i] = \binom{i-1}{n-1}N^{-n} \qquad \text{.... 4.18}$$

and of course

$$\text{Probability}[\underline{r}_{n+1} > N-i] = i/N \qquad \text{.... 4.19}$$

so that the algebraic summation of 4.17 gives

$$= \sum_{n=1}^{i} \binom{i-1}{n-1} N^{-n}.(i/N) = \frac{i}{N^2}(1+\frac{1}{N})^{i-1}$$

and after the summation in equation 4.15

$$\text{Average utilisation } U(\text{reloc,uniform}) = (1+\frac{1}{N})^{N+1} - 2 - \frac{1}{N} \quad . \qquad \ldots.\ 4.20$$

Similarly, for the exponential distribution,

$$\text{Probability}[\underline{r}_1+\ldots+\underline{r}_n=i] = r^i\binom{i-1}{n-1} \quad , \quad \text{writing } r \text{ for } r_N \qquad \ldots.\ 4.21$$

$$\text{and Probability}[\underline{r}_{n+1} > N-i] = r^{N-i+1}.(\frac{r^i-1}{r-1}) \qquad \ldots.\ 4.22$$

so that the summations of 4.17 and 4.15 lead to

$$\text{Average utilisation } U(\text{reloc,exp}) = \frac{1}{N}(2^{N+1} - 2 - \frac{1}{2r_N-1}) \quad . \qquad \ldots.\ 4.23$$

4.4.4 Internal fragmentation in the relocating model

In order to discuss internal fragmentation, a rounding up process has to be introduced whereby requests are allocated blocks which are equal to or larger than the amount requested. This can be done in a number of different ways, for example the (binary) buddy scheme rounds request sizes up to the next higher power of two, but in the present calculation the rounding that will be considered is to the next higher multiple of a fixed quantity, the allocation quantum. Rather than introducing this as an integral number of words, w say, so that allocated blocks are always w, 2w, 3w, ... words long which would require the external fragmentation analysis of the last section to be (fairly trivially) modified, this will equivalently be done by supposing that the unit of 1 word is the allocation quantum and that requests can originally have been for some fraction of this, with a size probability distribution before the rounding which results in the distributions 4.7, 4.9 of section 4.4.2 after the requests have been rounded. The difference between rounding original requests in words to multiples of w words, and rounding requests in fractions of words to integral amounts, is only one of scale and possibly of convenience in the calculation.

There are obviously many possible distributions of request size which will result, after rounding up to the nearest integer number of words, in the uniform and exponential distributions being considered here. For simplicity it will be assumed in the following examples that the average loss or amount of rounding up per request is half of one word. The same assumption will be made in chapter 6 in order to compare the results of that chapter with the analysis of these present examples. Having modified the request distribution in this way so that the distribution of rounded up request sizes remains the same, it is necessary to consider again the meaning of the quantity U, introduced as the average utilisation. As explained in section 2.1, it is convenient to keep the definition of U as the average fraction of the memory occupied by the (possible rounded up) allocated blocks, so that the average unused space between them is given by

$$E[EF] = 1 - U .$$

The fraction of memory which is allocated at any moment to the original request sizes before rounding up, or the fraction of memory which is actually "being used" at any moment, the "proper utilisation" of chapter 2, is then equal to U minus the amount IF of rounding up or internal fragmentation which has occurred. This amount of rounding is considered now for the relocating model and again below for the non-relocating case in chapter 6 section 6.5.3.

Whatever original distribution of request sizes is used, if the random variable IF is used to denote the fraction of memory unused because of rounding up, then

$$E[IF] = (1/2N) \times (\text{average number of allocated requests}) \qquad \dots\ 4.24$$

But (average number of allocated requests)

$$= \sum_{n=1}^{N} n \times \text{Probability}[n \text{ requests allocated in } N \text{ words}] \qquad \dots\ 4.25$$

and as before,

Probability[n requests in N words]

$$= \sum_{i=1}^{N} \text{Probability}[\underline{r}_1+\underline{r}_2+\dots+\underline{r}_n = i] \times \text{Prob}[\underline{r}_{n+1} > N-i] \qquad \dots\ 4.26$$

where also as before from section 4.4.1 it is understood that these are the probabilities at T=1 given that the model is in state E at T=0. The component quantities in this equation are the same as those occurring and already evaluated in the above expressions for the external fragmentation, but now the order of summation is different.

For the uniform distribution, using 4.18 and 4.19 of the last section 4.4.3, 4.24, 4.25 and 4.26 above can be summed to give

$$\text{Average internal fragmentation } E[IF](\text{uniform}) = \frac{1}{2N}[(1+\frac{1}{N})^N-1] \qquad \dots\ 4.27$$

Similarly for the exponential distribution, using 4.21 and 4.22 of the last section the algebra can again be summed and eventually simplifies, if that is the right word, to give

$$E[IF](\text{exponential}) = \frac{r^{N+1}}{r-1}\left[\frac{(N+1)(2r)^{N+2}-(N+2)(2r)^{N+1}-4r^2+4r}{8N(2r-1)^2} - 2^{N-2}\right] \qquad \dots\ 4.28$$

where $r = r_N$ has the value just above 0.5 determined by equation 4.10.

4.4.5 Summary of relocation analysis

Figure 4.8 plots the expressions for the expected utilisation U(reloc) and internal fragmentation E[IF], for values of memory size N up to N = 12, for the uniform and negative exponential request size distributions. The values shown in this figure appear in table 4.1.

		Uniform distribution (equations 4.7, 4.8)			Exponential request size distribution (equations 4.9, 4.10, 4.11)				
N	S(N)	U	E[IF]	Both	r_N	E_{exp}	U	E[IF]	Both
1	2	1.0	0.5	0.5	1.0	1.0	1.0	0.5	0.5
2	5	0.8750	0.3125	0.4375	0.6180	1.3820	0.8820	0.3455	0.4635
3	13	0.8272	0.2284	0.4012	0.5437	1.6170	0.8518	0.2963	0.4445
4	34	0.8018	0.1802	0.3784	0.5188	1.7657	0.8475	0.2744	0.4269
5	89	0.7860	0.1488	0.3628	0.5087	1.8590	0.8532	0.2633	0.4101
6	233	0.7752	0.1268	0.3516	0.5041	1.9165	0.8627	0.2573	0.3946
7	610	0.7674	0.1105	0.3431	0.5020	1.9514	0.8734	0.2541	0.3807
8	1597	0.7615	0.0979	0.3364	0.5010	1.9721	0.8840	0.2522	0.3682
9	4181	0.7569	0.0878	0.3309	0.5005	1.9842	0.8938	0.2512	0.3574
10	10946	0.7531	0.0797	0.3266	0.5002	1.9912	0.9027	0.2507	0.3480
11	28657	0.7500	0.0729	0.3229	0.5001	1.9951	0.9106	0.2504	0.3398
12	75025	0.7475	0.0672	0.3197	0.5001	1.9973	0.9175	0.2502	0.3327

Table 4.1 Average utilisation and fragmentation in the relocating model

Notes: N = memory size.
S(N) = f_{2N} = total no. of configurations, including the empty state.
Average utilisation U and average internal fragmentation E[IF] are computed from equations 4.20, 4.23, 4.27 and 4.28 (see figure 4.8).
Both = average total fragmentation, external+internal = (1-U)+E[IF].

The exponential distribution, for increasing values of N gives almost constant weights to small request sizes, approaching the limits 1/2, 1/4, 1/8, ... as can be seen in table 4.1. The probability of a request for any given fraction of the total memory, say half of it, approaches zero as N increases, and this is reflected in the utilisation and internal fragmentation

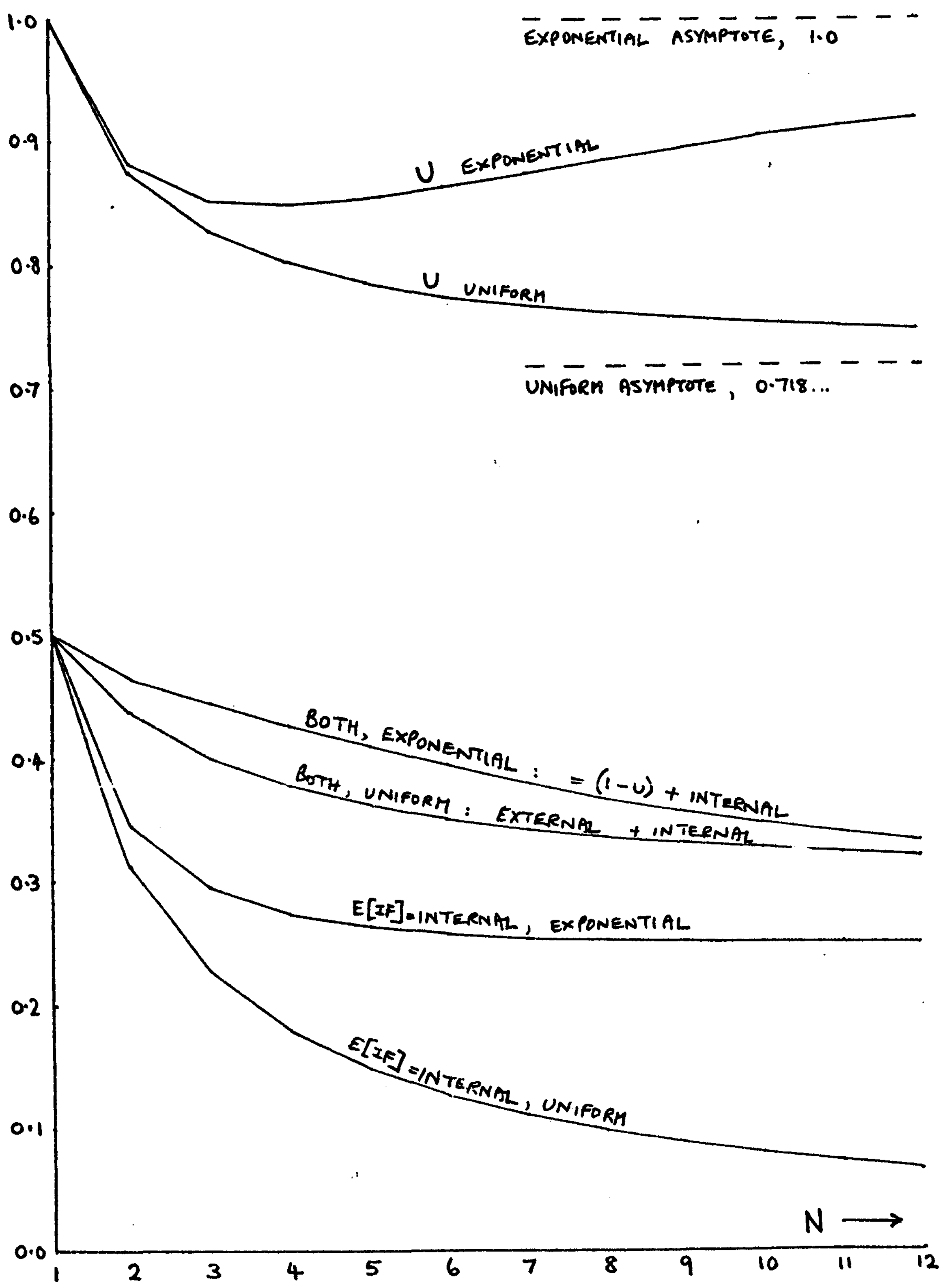


Figure 4.8 Utilisation and internal fragmentation in the relocating model

Average utilisation U and average internal fragmentation E[IF] are computed from equations 4.20, 4.23, 4.27 and 4.28. U and E[IF] are shown for both the uniform and exponential distributions. Compare this figure with tables 4.1 and 6.5, and figures 6.6 and 6.7.

curves shown in figure 4.8. The utilisation approaches 1.0 as the expected amount of wasted space left over at the end of memory (because the next request is too large for it) becomes insignificant compared to N, but the internal fragmentation approaches the limit 0.25 as the average number of allocated blocks approaches N/2.

Conversely, for the uniform distribution the average number of allocated blocks becomes insignificant compared with N, since the memory is just as likely to be filled by the next request in the queue being for N words as that it is for one word, and so the average internal fragmentation approaches zero as N increases. The average amount of space wasted at the end of memory because it is insufficient for the next request remains significant compared with N, and so the utilisation approaches a limit of 0.718 approximately.

All four of these curves provide upper limits for the corresponding quantities when relocation is not allowed, as will be seen in chapter 6, and the curves in figure 4.8 are consequently reproduced in figures 6.6 and 6.7 below so that they can be compared with the values computed from the various non-relocating models reported on in that chapter.

4.5 Storage fragmentation and the problem of size: ways to proceed

This chapter has shown that any straightforward application of the Markov theory established in chapter 3 to the present storage allocation model (and indeed to other similar exact models, as will be indicated in chapter 7) is likely to run into difficulties because of the very large number of states which naturally occur for any reasonable memory size N. One traditional way which is often used to overcome such a difficulty is to redefine the model and start again, or in other words to leave the difficult problem on one side and solve an easier one instead. This has been done here; by allowing blocks to be relocated the difficulties of fragmented space go away, the problem no longer grows as rapidly with memory size, and in any case closed form solutions for quantities of interest such as the expected utilisation and wasted space can be obtained in individual cases, although not always without more than a little labour. Of course this does not solve the original problem which still remains, but the relocating analysis is particularly useful anyway as its space utilisation performance provides an upper bound to that of the corresponding non-relocating model which does become fragmented.

The rest of this thesis is increasingly concerned either directly or indirectly with trying to answer the question, what can be done with the original problem in spite of the very large growth in the number of states with memory size, without the side-stepping trick such as changing the problem definition as was done to produce the relocation analysis. Chapter 5 concentrates with some success on the constant underlying algebraic structure which has already been noted at the beginning of this chapter as apparently being present in the transition matrix whatever its size. The hope in doing this is that useful properties can be deduced from the structure which in turn will either allow the problem to be collapsed to a more manageable size, or else permit the dependence of utilisation and fragmentation on parameters such as the distribution of requests and the choice of allocation algorithm to be studied regardless of the size of the matrix. Chapter 6 presents the results of directly computing the transition matrix P and its dominant eigenvector $\underline{\pi}$ from which the average utilisation and fragmentation can be extracted. It also explains the variety of ways that have been used to contain the exponential growth of the number of states for as long as possible, for the range of still comparatively small memory sizes up to the largest (N=12) that has been managed so far. Chapter 7 outlines the subsequent ideas and alternative approaches (some of which are concerned with the size of the problem) which have been generated as a result of this work, and suggests ways in which they can be followed up. From these three chapters it will be clear that making exact predictions about storage allocation models would not always be easy to do if the problems of size were not present, but it is certainly much harder to do because of them.

Chapter 5 : Algebraic analysis of the steady state equation

This chapter concentrates again on the non-relocating model of storage allocation described in chapter 3 section 3.1, which was shown in section 3.2 to be Markovian if certain assumptions about its behaviour were made. The transition matrix $P = (p_{ij})$ of this Markov chain was defined in section 3.4 in the natural way by equation 3.10, p_{ij} being the conditional probability of a transition at the next event to state j given that the model is in state i. In chapter 4 section 4.2, some examples of P were exhibited (figures 4.1 to 4.7), for models of memory size up to N = 6 words. These examples indicate that the elements of P might be arranged in some sort of complicated repeating pattern, suggesting that P might perhaps have a structure which could be analysed.

In section 5.1 below it is shown by considering the intermediate stages or steps of a complete transition that P does indeed have an algebraic expansion as sums and products of much simpler matrices which represent these individual steps. The elements of these simpler matrices are found to be arranged in simple repeating patterns, the complexity of the transition probability matrix P being produced almost entirely by the algebraic combination of these matrices. Section 5.2 reduces this complexity and justifies the usefulness of this expansion of P by using it to simplify the steady state equation 3.11 to a form more suitable for analysis, which is investigated in section 5.3.

5.1 Algebraic expansion of the transition matrix P

Any single transition between two states consists in general of a sequence of more basic transitions between intermediate memory configurations which each move just one block, either out of the memory (basic deallocation transition) or into memory from the top of the queue (basic allocation transition). Since the request initially at the head of the queue would have been allocated as part of the previous transition if it could have been, the first movement in each transition must be a deallocation. All of the rest, if any, are allocations because deallocations and allocations are performed instantaneously and the probability of more than one simultaneous deallocation is zero. If the request initially at the head of the queue before the transition occurs will still not fit after the deallocation then there are no subsequent allocations and the transition is completed, otherwise requests are allocated from the queue until one is reached which cannot be fitted into memory. If the memory contained just one allocated block, then after its deallocation the memory is momentarily empty and so the empty configuration is a possible intermediate state.

It follows that the transition probability p_{ij} from any state i to any state j can be expanded in the usual way as products of the individual probabilities of these more basic transitions summed over all the possible ways of combining them to get from i to j. In figure 3.1 for example, the probability of the occurrence of the illustrated transition using the best fit algorithm given that the initial state of memory is as shown, is the probability of the initial deallocation of the 2-word block multiplied by the probability of a 4-word, a 2-word and a 3-word request being allocated in the indicated positions, multiplied by the probability that the next request in the queue is for more than one word. The complete probability of getting from this particular initial to this equally particular final state is therefore the total of all such products of probabilities, summed for all the deallocation - allocation sequences which are possible between these two states. It therefore becomes necessary to know what are the probabilities of deallocating or allocating exactly one block from or into any given memory configuration, and what are all the ways in which these may be combined. These are studied in detail in the following sections. The basic deallocation transition probabilities are considered first because they occur first in any transition and are simpler because there is just one deallocation in each complete transition.

5.1.1 Deallocation matrix

This is the matrix $D = (d_{ij})$ of basic deallocation transition probabilities d_{ij}, $i,j = 1,\ldots,N$, which are obtained by deallocating one block:

$$d_{ij} = \begin{array}{r}\text{conditional}\\ \text{Probability}\end{array} \left[\begin{array}{c|c} \text{state } j \text{ will be reached} & \text{model} \\ \text{by deallocating at random} & \text{is in} \\ \text{one of the allocated} & \text{state } i \\ \text{blocks in state } i & \end{array} \right] \qquad \ldots.\ 5.1$$

Whatever state i the model is in, the block to be deallocated is chosen with equal probability from those present. The matrix has a nice simple recursive structure, the reasons for which are not hard to see but which are somewhat lengthy when set out rigorously below as a formal argument.

Informal explanation

As an example, consider the deallocation matrix of the N = 4 model, figure 5.1. The 34 states can be split into five groups, those which begin (i.e. at the left hand end) with a block of length k words, k = 1, 2, 3, 4,

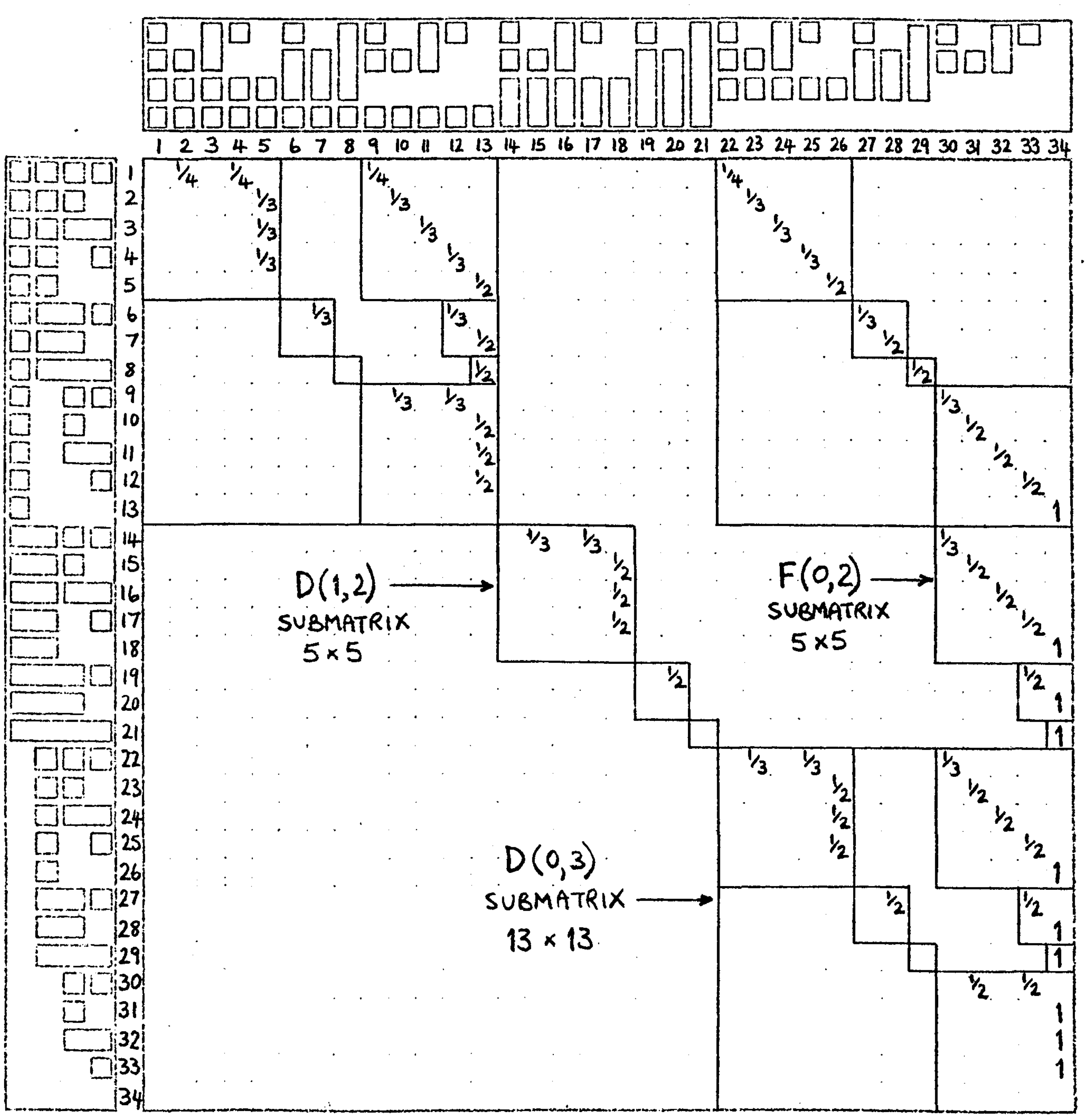


Figure 5.1 Deallocation matrix $D=(d_{ij})$ for N=4

$$d_{ij} = \text{conditional Probability}\left[\begin{array}{c}\text{state } j \text{ will be reached}\\ \text{by deallocating at random}\\ \text{one of the allocated}\\ \text{blocks in state } i\end{array}\middle|\begin{array}{c}\text{model}\\ \text{is in}\\ \text{state } i\end{array}\right] \quad \dots\dots 5.1$$

and those in which the first word is empty. The ordering used, defined in section 4.3, naturally groups these states together in that order. First, consider a group of states all of which begin with a block of length k, for example, k = 2. In this group, for any state i having exactly a_i allocated blocks there are two possibilities for deallocation. Either

1) the first block (of length k) is deallocated with probability $1/a_i$. These possibilities for all the states i in group k form a diagonal submatrix (for k = 2 this is the submatrix labelled F(0,2) in figure 5.1). This is because the relative ordering of states beginning with a k-word block is the same as those beginning with at least a k-word gap,

or 2) the first block stays put with probability $1 - 1/a_i$ and another is deallocated, as if this was the (N-k) problem applied to the last (N-k) words of memory but in which the row sums no longer add to 1. These possibilities naturally lie within the square submatrix on the main diagonal of D (for k = 2, labelled D(1,2) in figure 5.1) which includes all possible deallocation transitions from group k states to other group k states.

Now consider the last group of states all of which have an unallocated first word. Their deallocations can be considered exactly as if this was an (N-1)-word memory ignoring the first word, and so they also form a square submatrix on the main diagonal of D (labelled D(0,3) in figure 5.1), this time with row sums which are all equal to 1 (except of course for the empty state).

Formal argument

The recursion of submatrices of D should now be apparent. At any stage, a subset of states is chosen by dividing memory into a left and right region, see figure 5.2. The left region contains a single fixed configuration, L say,

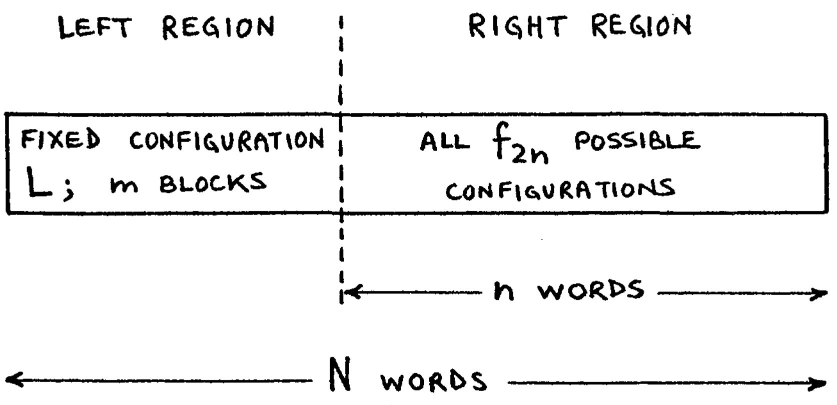

LEFT REGION
RIGHT REGION
FIXED CONFIGURATION L; m BLOCKS
ALL f_{2n} POSSIBLE CONFIGURATIONS
n WORDS
N WORDS

Figure 5.2 Subset of states Sub(L,m,n), f_{2n} in all.

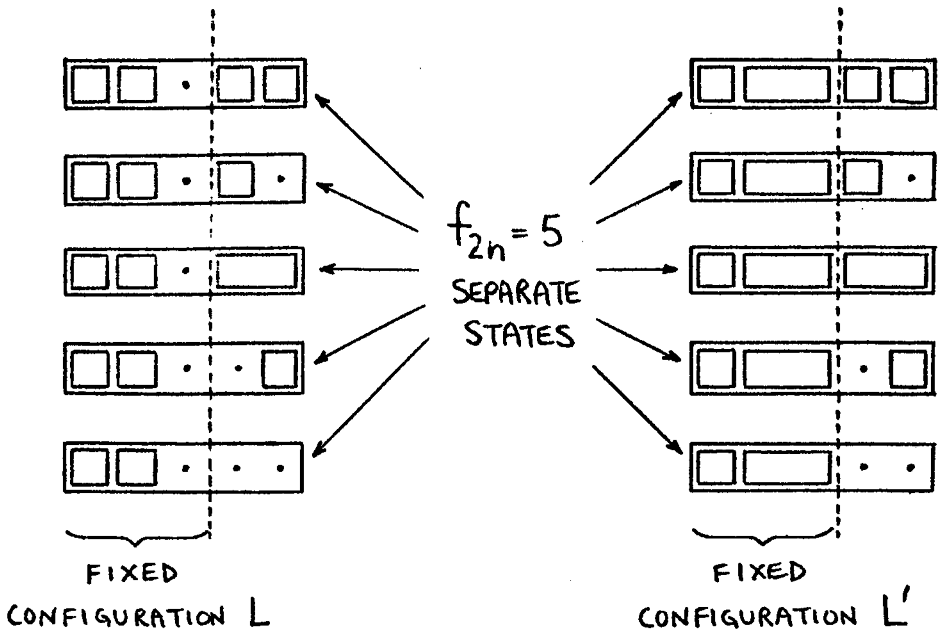


Figure 5.3(a) Subset Sub(L,m,n) N=5, m=n=2 Figure 5.3(b) Subset Sub(L',m,n) N=5, m=n=2

of exactly $m \geq 0$ allocated blocks and possibly some gaps, while the right region of length n words, $0 \leq n \leq N$, varies through all the $S(n) = f_{2n}$ possible configurations of blocks and gaps that it can contain, the m fixed blocks being completely contained in the first N-n words. Either region may have zero length, in which case the other will have all N words.

The juxtaposition of the left configuration with all f_{2n} right configurations gives a set of states: Sub(L,m,n) which will be grouped together by the state ordering defined in section 4.3. Figure 5.3(a) shows an example of such a subset Sub(L,m,n). In this example, N=5, i.e. it is a 5-word memory, and m=2, n=2; the left region of N-n = 5-2 = 3 words contains a configuration L with m=2 allocated blocks, which happen to be 1-word blocks in the first and second words in this example, and the right region of n=2 words can contain any of $f_{2n} = f_4 = 5$ possible configurations, so that there are five states in this subset. Figure 5.3(b) shows another subset Sub(L',m,n) in which again N=5, m=n=2, the only difference being that the left region contains a different fixed configuration L' say, still having m=2 allocated blocks in this example.

Strictly, the notation Sub(L,m,n) is redundant as L determines both m and n, but it is convenient to make them explicit. Two gaps may be adjacent at the boundary between left and right, as in the last two states of the example in figure 5.3(a); they will combine to form a single gap when the regions are viewed together as an N-word memory. Cases in which blocks are allowed to straddle the boundary will not have to be considered.

The notation D(m,n) is defined to be the submatrix of the whole deallocation matrix D = (d_{ij}) containing all the entries d_{ij} for which both i and j represent states in Sub(L,m,n), so that D(m,n) contains all the possible one-block basic deallocation transitions from states in Sub(L,m,n) for which the resulting states are also in Sub(L,m,n). Because the chosen state ordering groups all the states of Sub(L,m,n) together, D(m,n) is a square block on the diagonal of D. Dropping L from the definition of D(m,n) nevertheless leaves it well defined, as any other left-hand configuration L' having the same values for m and n will give rise to the same deallocation submatrix, except of course that it will be in a different place on the main diagonal of D.

The recursive composition of D begins with

D = D(0,N) = whole deallocation matrix. 5.2

Following the example in the above informal explanation, any square submatrix D(m,n) lying on the main diagonal of D can be recursively split by grouping the f_{2n} states of Sub(L,m,n) into (n+1) groups, those in which the right region begins with a block of length k, k = 1,...,n , and those in which the first word of the right region is empty, see figure 5.4. Consider each group in turn. Call the group in which the right region begins with a block of k words Sub(L,m,n):(k). Then by concatenating this k-block to the left-hand fixed region L leaving (n-k) words on the right,

Sub(L,m,n):(k) = Sub(L(k),m+1,n-k), ... 5.3

where L(k) is the new left-hand region formed by this concatenation. As in the informal example, there are two possible sets of deallocations in D(m,n) for this subset, the submatrix D(m+1,n-k) in which L and the k-block are preserved and a block in the (n-k) rightmost words is deallocated, and the diagonal submatrix F(m,n-k) (F for first) in which the block of length k is deallocated. The last subset of Sub(L,m,n) in which the first word of the right-hand region is empty can similarly be called Sub(L,m,n):(-1), where the notation :(-k) is meant to indicate all those states in which the right hand region begins with at least k contiguous empty words. Then

Sub(L,m,n):(-1) = Sub(L(-1),m,n-1) 5.4

where the notation L(-k) indicates the new left hand region formed by

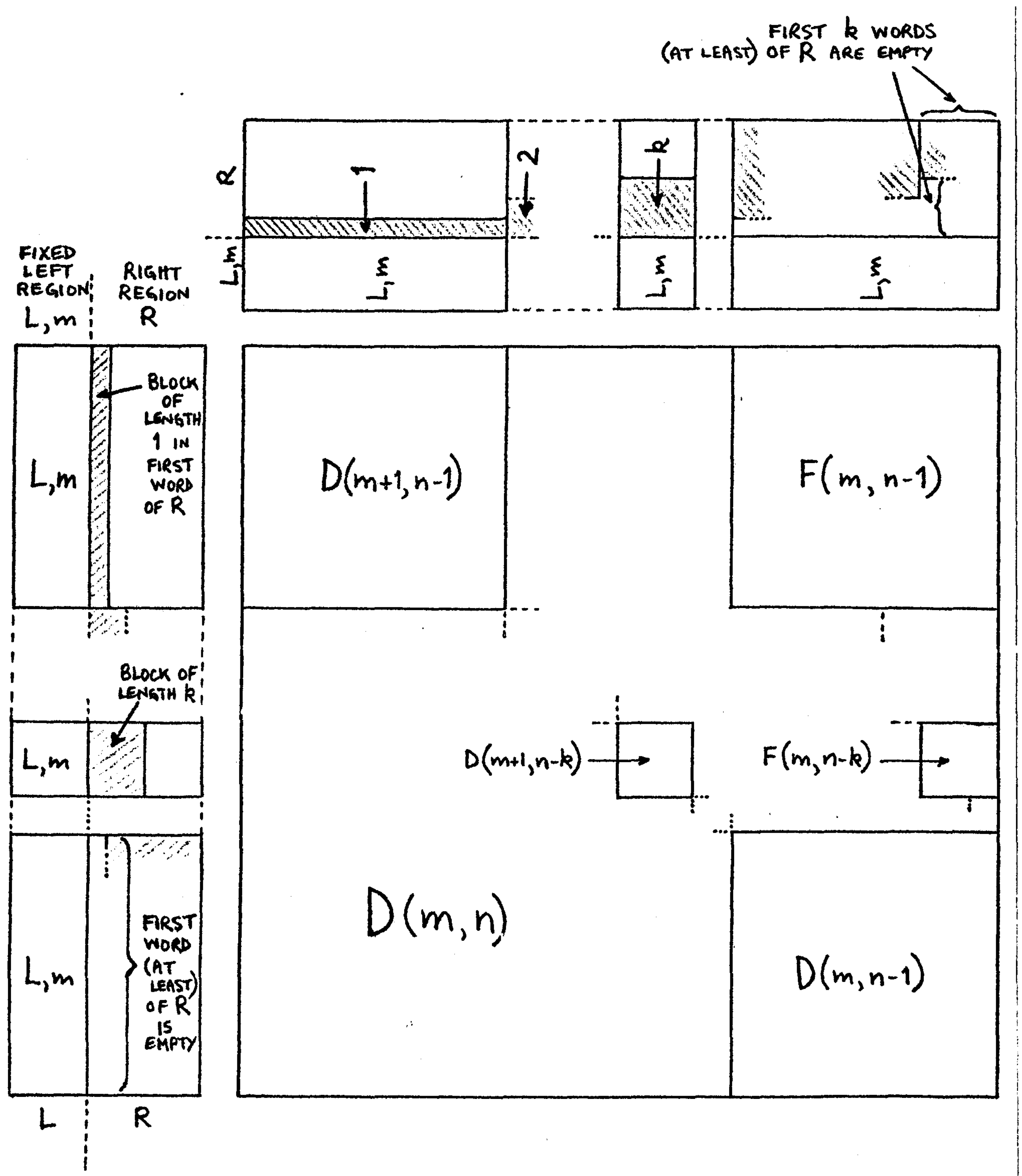


Figure 5.4 Recursive composition of D(m,n)

concatenating L with k contiguous empty words. Hence this last subset of states of Sub(L,m,n) gives the submatrix D(m,n-1).
Summarising so far,

$$D(m,n) = \sum_{i=0}^{n-1} D(m+1,i) + \sum_{i=0}^{n-1} F(m,i) + D(m,n-1) \qquad \ldots\ 5.5$$

where, in the use of "+" or addition, the position of each of these variously sized submatrices is understood.

It remains to see how to determine the values in the diagonal F matrices. A recursive formula can be found for these also, and figures 5.5 and 5.6 may help to illustrate the following slightly complicated argument, which identifies other smaller F matrices and shows that these contain the same values as any given F matrix.

From equation 5.5, each F(m,n) arises from the decomposition of a larger D matrix, D(m,k+n) for some positive k. Replacing n by k+n, equation 5.5 can be rewritten:

$$\begin{aligned} D(m,k+n) = {} & D(m+1,k+n-1) + \ldots + D(m+1,n) + \ldots + D(m+1,0) \\ & + F(m, k+n-1) + \ldots + F(m, n) + \ldots + F(m, 0) \\ & + D(m, k+n-1) \end{aligned} \qquad \ldots\ 5.6$$

F(m,n) represents the transitions made by deallocating the block of length k beginning in the first word of the right-hand half of length (k+n) for all those states

$$\mathrm{Sub}(L,m,k+n):(k) = \mathrm{Sub}(L(k),m+1,n)$$

of Sub(L,m,k+n) with such a block at the beginning of the (k+n)-word right half. That is, the states which give rise to F(m,n) have a left half L of (N-k-n) words with m blocks allocated, and then at the beginning of the right half of (k+n) words a block of length k, the deallocation of which gives F(m,n). Except for the very last of these states, the only one in which the

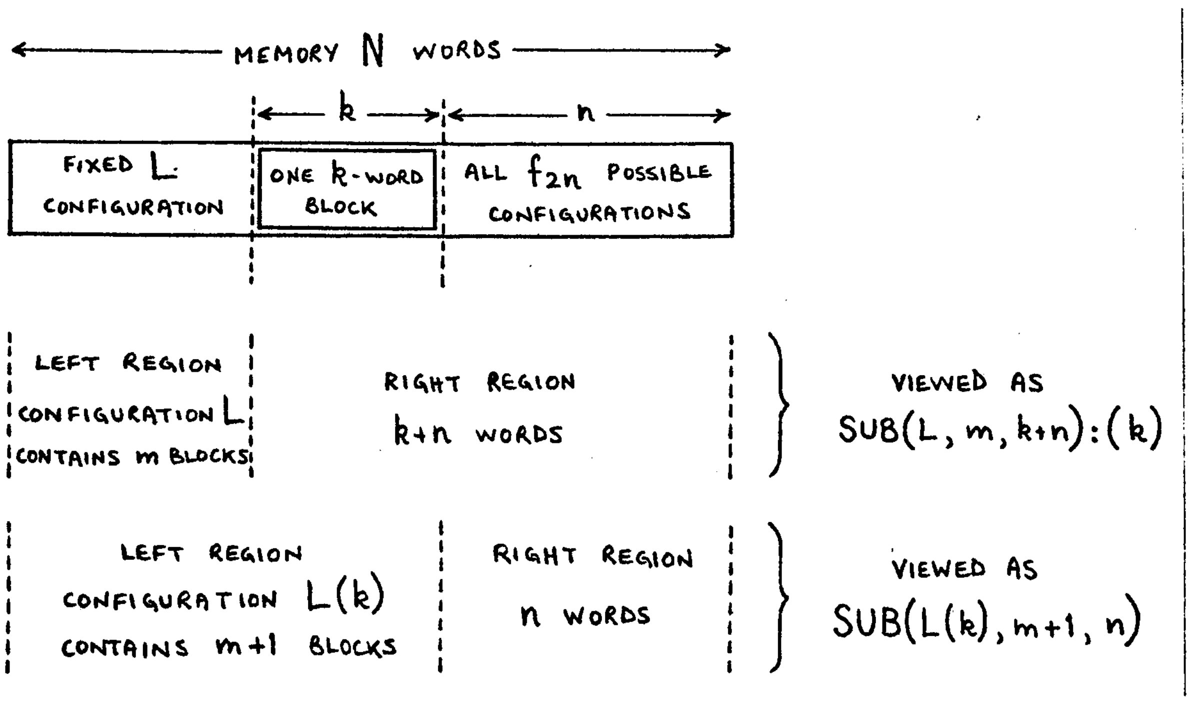

MEMORY N WORDS
k
n
FIXED L CONFIGURATION
ONE k-WORD BLOCK
ALL f_{2n} POSSIBLE CONFIGURATIONS
LEFT REGION CONFIGURATION L CONTAINS m BLOCKS
RIGHT REGION k+n WORDS
VIEWED AS SUB(L, m, k+n):(k)
LEFT REGION CONFIGURATION L(k) CONTAINS m+1 BLOCKS
RIGHT REGION n WORDS
VIEWED AS SUB(L(k), m+1, n)

Figure 5.5 The subset of states Sub(L,m,k+n):(k)=Sub(L(k),m+1,n)

rightmost n words are empty, each of them (there are f_{2n} altogether) has alternative deallocations in D(m,k+n) to the k-word block, i.e. D(m+1,n) from equation 5.6, and each alternative will have the same probability as the corresponding element of F(m,n) which is being sought, because in any state each allocated block is equally likely to be deallocated. In particular for the first f_{2n-1} states in Sub(L(k),m+1,n) in which the first word of the n-word right half is allocated to a block, there are the alternatives F(m+1,n-1), ..., F(m+1,0) which will arise in the decomposition of D(m+1,n), the alternative to F(m,n). Thus, the values of F(m,n) will be the same as those of F(m+1,n-1), ..., F(m+1,0) for these first f_{2n-1} states. For the remaining f_{2n-2} states, those in which the first word of the n-word right half is empty, there is a matching subset of states Sub(L(-k),m,n):(1) elsewhere in

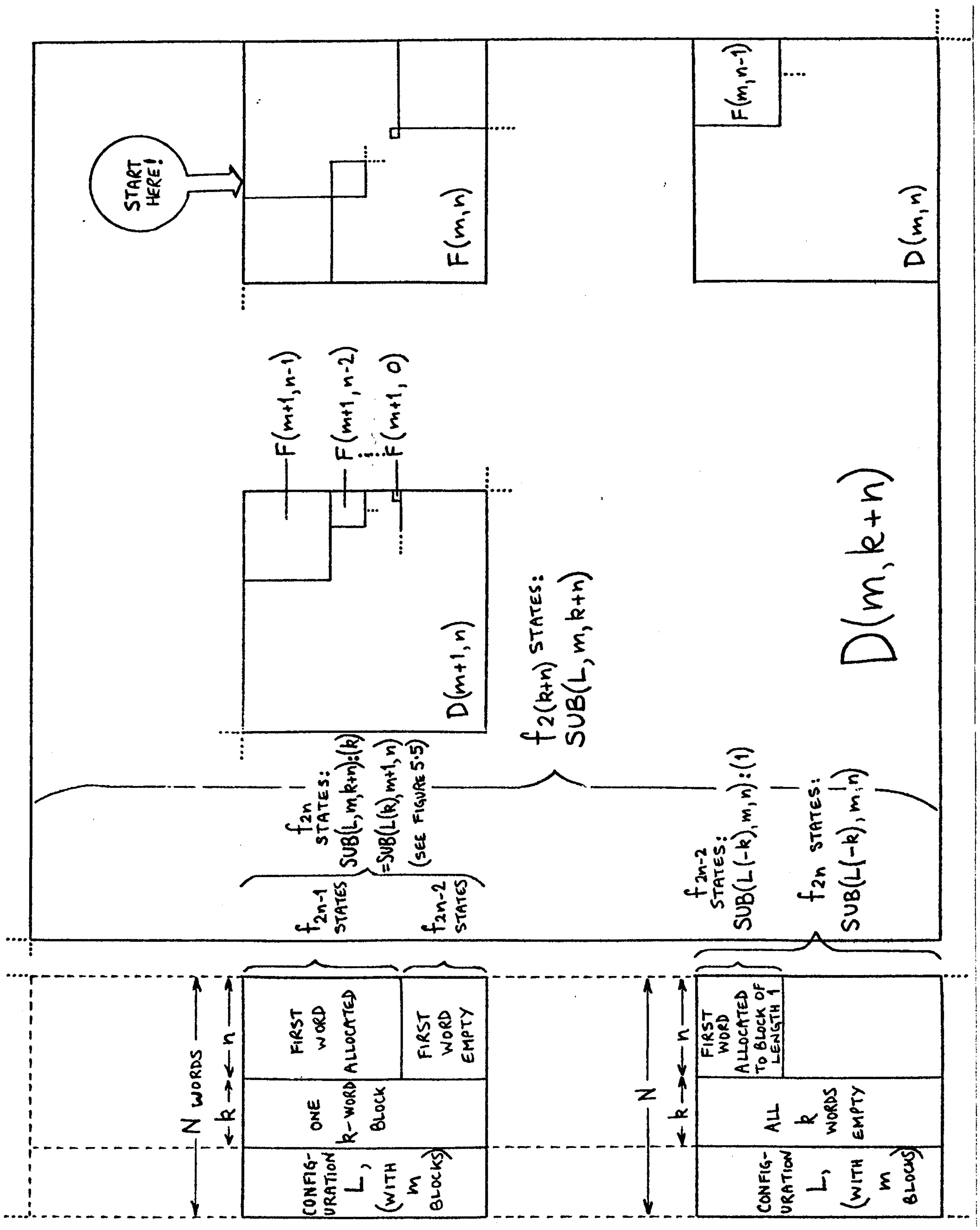

START HERE!
F(m,n-1)
F(m,n)
D(m,n)
F(m+1,n-1)
F(m+1,n-2)
F(m+1, 0)
D(m+1,n)
$f_{2(k+n)}$ STATES: SUB(L,m,k+n)
D(m,k+n)
f_{2n} STATES: SUB(L,m,k+n):(k) =SUB(L(k),m+1,n) (SEE FIGURE 5·5)
f_{2n-1} STATES
f_{2n-2} STATES
f_{2n-2} STATES: SUB(L(-k),m,n):(1)
f_{2n} STATES: SUB(L(-k),m,n)
N WORDS
k
n
FIRST WORD ALLOCATED
FIRST WORD EMPTY
ONE k-WORD BLOCK
CONFIGURATION L, (WITH m BLOCKS)
N
k
n
FIRST WORD ALLOCATED TO BLOCK OF LENGTH 1
ALL k WORDS EMPTY
CONFIGURATION L, (WITH m BLOCKS)

Figure 5.6 Recursive composition of F(m,n)

The argument matches the component submatrices of F(m,n) with others representing equally probable alternatives inside its parent D(m,k+n)

Sub(L,m,k+n), where the match is obtained by removing the k-block to leave a gap of length k and adding instead a block of length 1 in the (empty) first word of the n-word right half. Each of the states in Sub(L(-k),m,n):(1) will have deallocation transition alternatives with the same probabilities (but to different destination states of course) as the corresponding states in the remainder of Sub(L(k),m+1,n), including the last one, because the number of allocated blocks is pairwise state for state the same. Pick out one alternative for each state in Sub(L(-k),m,n):(1) by choosing to deallocate the 1-word block. These are exactly the values which will appear in a diagonal submatrix F(m,n-1) in the decomposition of D(m,n) for the set Sub(L(-k),m,n) of which the states Sub(L(-k),m,n):(1) are a subset.

Summary of section 5.1.1 (Deallocation matrix)

For integer $m \geq 0$, $n > 0$,

$$\begin{aligned} D(m,n) = {} & D(m+1,n-1) + \dots + D(m+1,0) \\ & + F(m, n-1) + \dots + F(m, 0) \\ & + D(m, n-1) \end{aligned} \qquad \text{(positional placing understood)} \qquad \dots.\ 5.5$$

$$\begin{aligned} F(m,n) = {} & F(m+1,n-1) + \dots + F(m+1,0) \\ & + F(m, n-1) \end{aligned} \qquad \dots.\ 5.7$$

By inspection,

$$D = \text{whole deallocation matrix} = D(0,N) \qquad \dots.\ 5.2$$

$$\left.\begin{aligned} D(m,0) &= 0 \\ F(m,0) &= 1/(m+1) \end{aligned}\right) \quad 1 \times 1 \text{ matrices} \qquad \dots.\ 5.8$$

5.1.2 Basic allocation transitions

The allocation part of the decomposition of the transition probability matrix P is more complicated than the deallocation matrix because

a) this is where the choice of allocation placement algorithm enters,

b) the size q of the first request waiting at the head of the queue varies not according to the unconditional distribution (r_n) (equation 3.4, section 3.2.1), but to the related conditional distribution (q_n) (equation 3.8, section 3.2.4), whereas the sizes of subsequent requests in the queue follow the unconditional distribution,

c) a variable number of requests will be allocated depending on how far the allocation process gets before a request is reached which will not fit.

Suppose the allocation process is proceeding and has reached a point at which the allocation of the request now at the head of the queue is about to be attempted, and that the size n of this request is known. The allocation placement algorithm which is being used, e.g. first fit, best fit, is also supposed known. Then given the configuration of allocated blocks in memory, everything is available to determine if the request will fit and if it does, where it will be allocated, or if there is a random element in the placement algorithm, the probability of its being allocated at any possible location. Allocation algorithms which require more information than this, for instance those which look down the queue to see what is coming next before deciding where to put the present request, will not be considered here. Taking all the memory configurations together, all the entries can be determined in a matrix A_n, the basic allocation probability matrix for request size n, of probabilities which represent these basic allocation transitions.

To understand the matrices A_n, $n = 1,\ldots,N$, it is helpful to consider a matrix which has as its non-zero entries the size of request that would cause the corresponding basic allocation transition to occur. This is referred to as the generalised allocation incidence matrix. Figure 5.7 is the generalised allocation incidence matrix for $N = 4$. It indicates where the non-zero

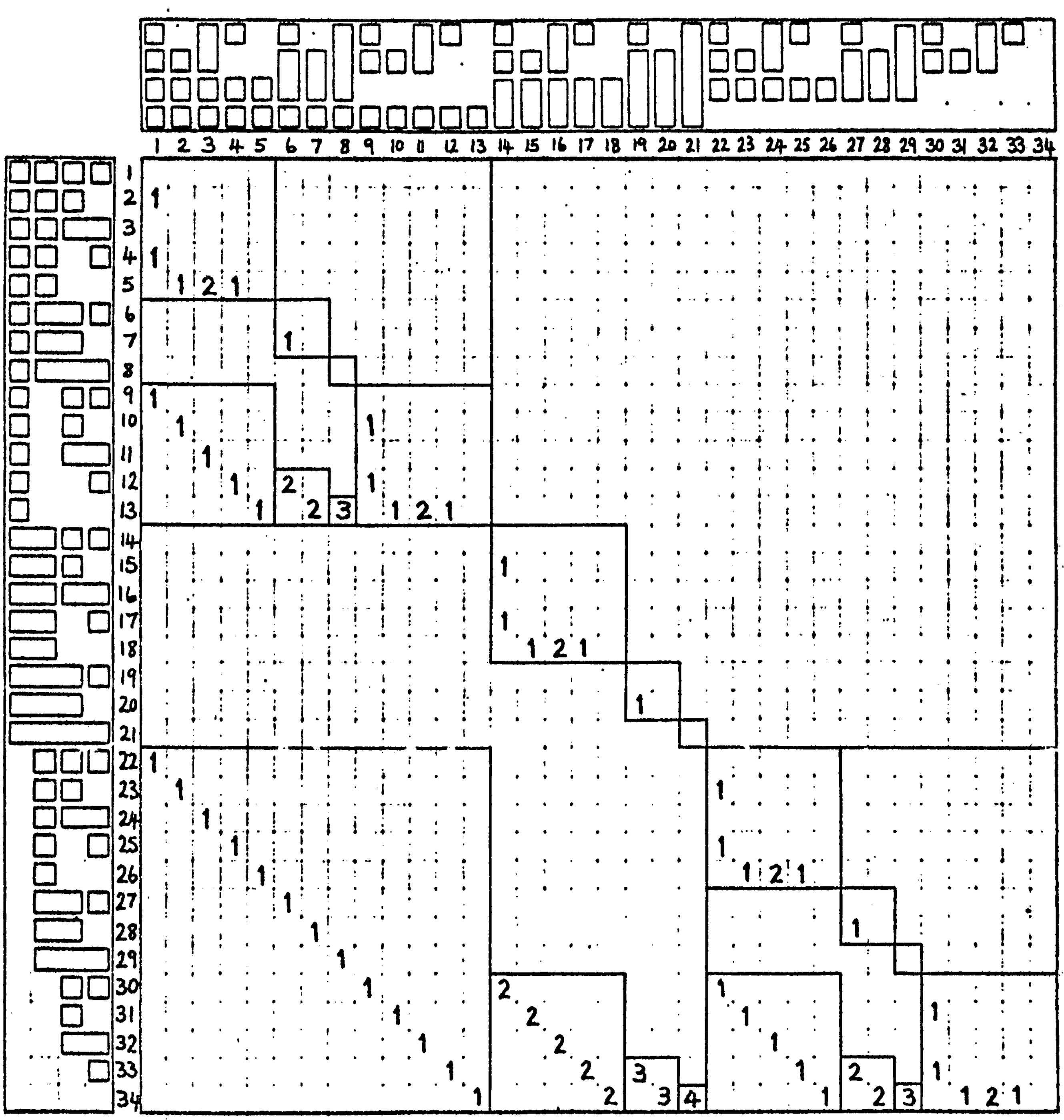


Figure 5.7 Generalised allocation incidence matrix for N=4

All possible allocations are shown. The entries are not probabilities, but indicate where non-zero allocation transition probabilities may occur. Each entry (1,2,3 or 4) is the size of request for the corresponding allocation.

entries can occur in such a basic allocation probability matrix A_n, for all possible request sizes n = 1, 2, 3, 4 in an N = 4-word memory. Not all of the entries that can be non-zero need necessarily be so in any particular A_n, depending on the allocation algorithm. For example, if n = 2 and the first fit allocation algorithm is used, then the basic allocation probability matrix A_2 for N=4 is as shown in figure 5.8. It has zero probabilities everywhere except for 1's which can only occur at those places where a 2 appears in figure 5.7. Of course, the choice of allocation algorithm is almost irrelevant for a memory size as small as N = 4.

Figures 5.9 to 5.12 show the four general basic allocation probability matrices A_1, A_2, A_3, A_4 for N=4. They have been displayed at length because they are referred to again in section 5.3. In these general matrices all the possible allocation choices have been retained by using variables in place of constants to mark the entries where a choice is possible. For example, in a four word memory there is just one state (the empty state) for which there is a choice of where to place a block of three words, and so in A_3 for N = 4, shown in figure 5.11, only the last row which describes the possible transitions from the empty state contains more than one non-zero element. The entry h is the probability that the three word block will be allocated in the first three words of the memory, and h' is the complementary probability of the only possible alternative that it will be allocated in the last three words. Of course, $h, h' \geq 0$ and $h + h' = 1$; and similarly for the choices in A_1 and A_2, figures 5.9, 5.10. The choice may be between several states, in A_2 there occurs $i + i' + i'' = 1$, and in A_1,

$$w + w' + w'' = s + s' + s'' = p + p' + p'' = m + m' + m'' = 1$$

and $l + l' + l'' + l''' = 1$. States which contain exactly one gap just sufficiently large enough to allocate the requested block have corresponding rows containing precisely one non-zero transition probability, value 1. For

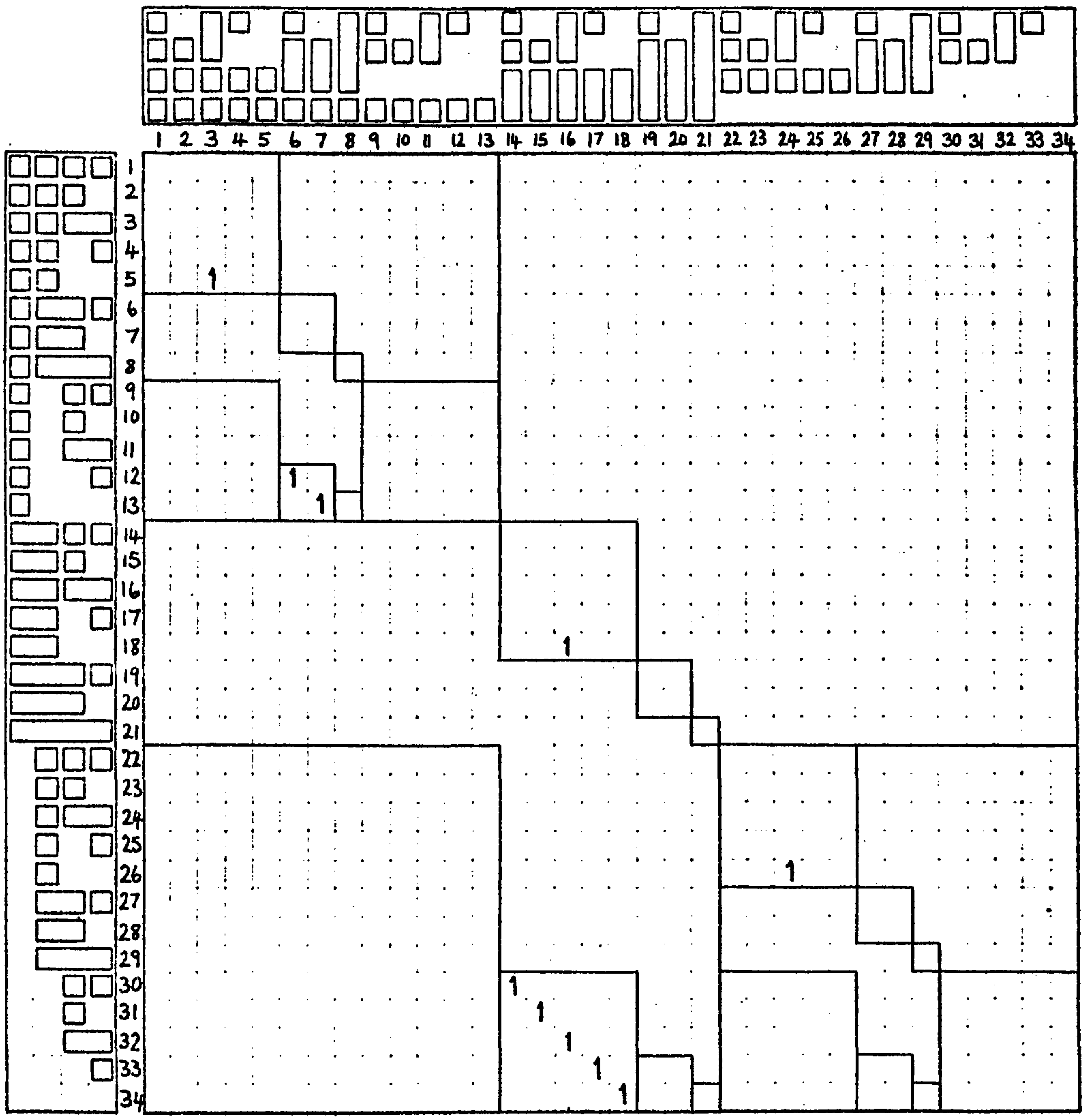


Figure 5.8 Basic allocation matrix A_2 N=4, first fit, request size n=2

A_2 is strictly lower triangular with the given state ordering.
The entries in this matrix are probabilities.

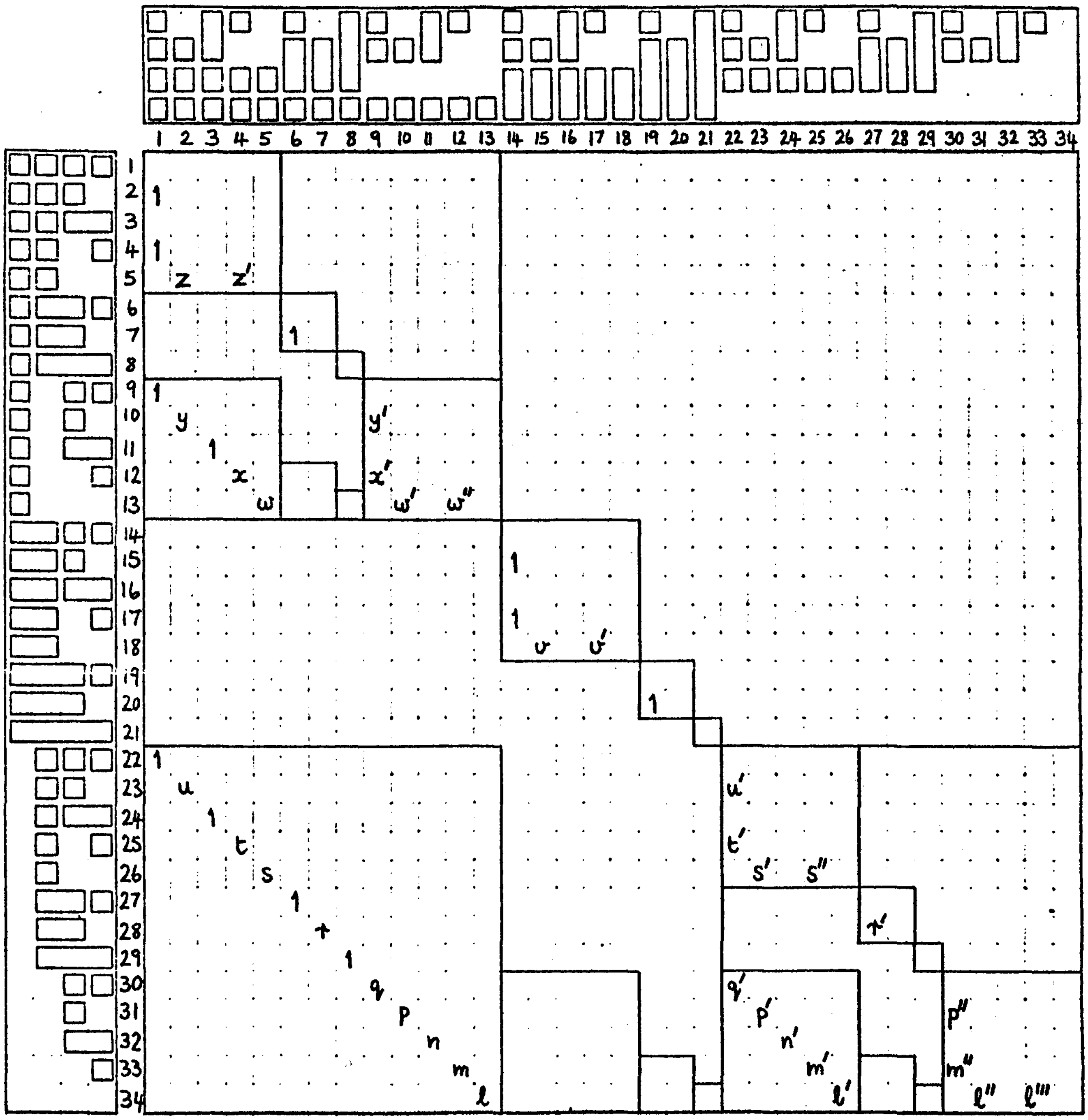


Figure 5.9 Basic allocation matrix A_1 N=4, general allocation algorithm.

The entries in this matrix are probabilities.

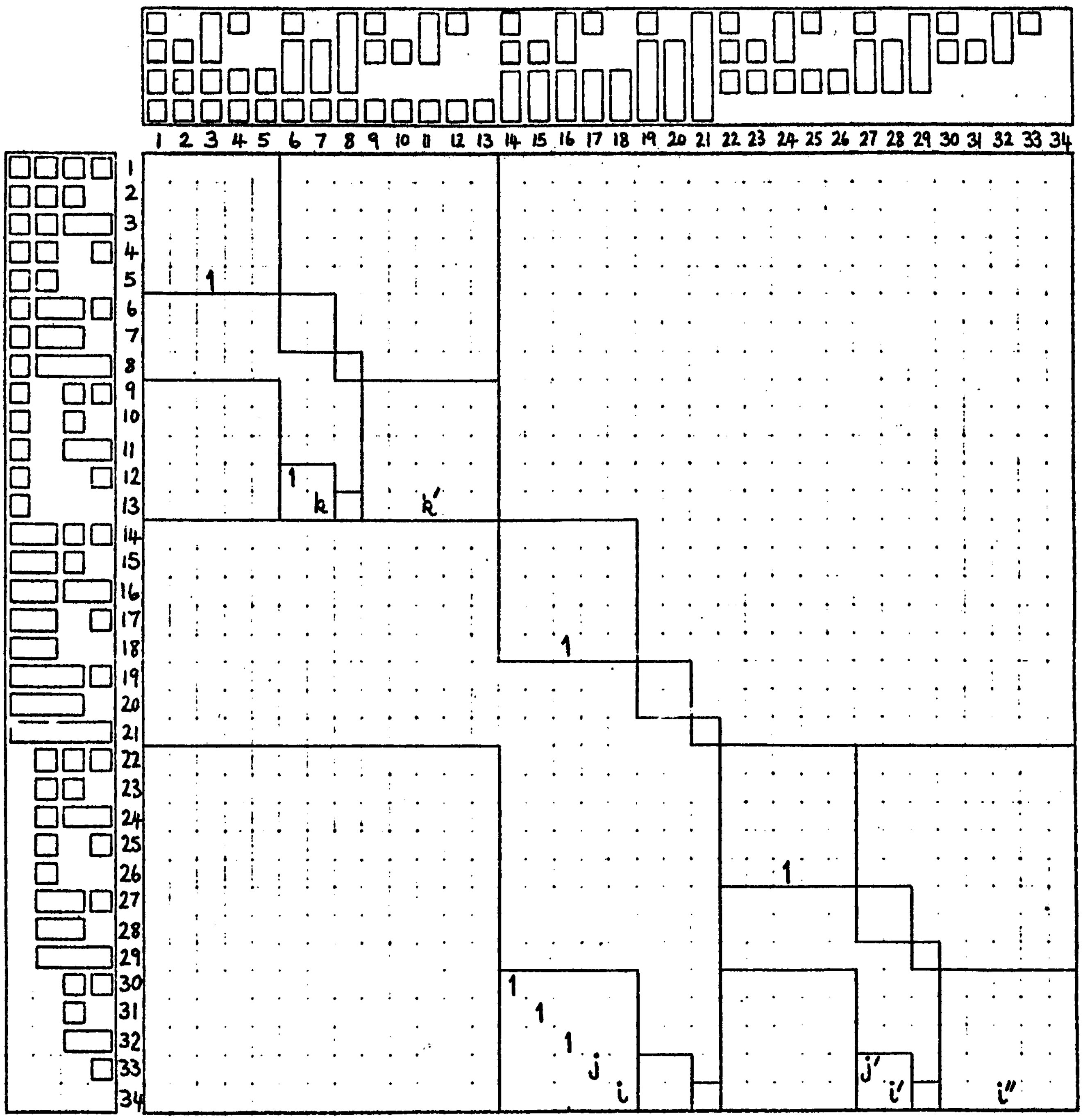


Figure 5.10 Basic allocation matrix A_2 N=4, general allocation algorithm.

The entries in this matrix are probabilities.

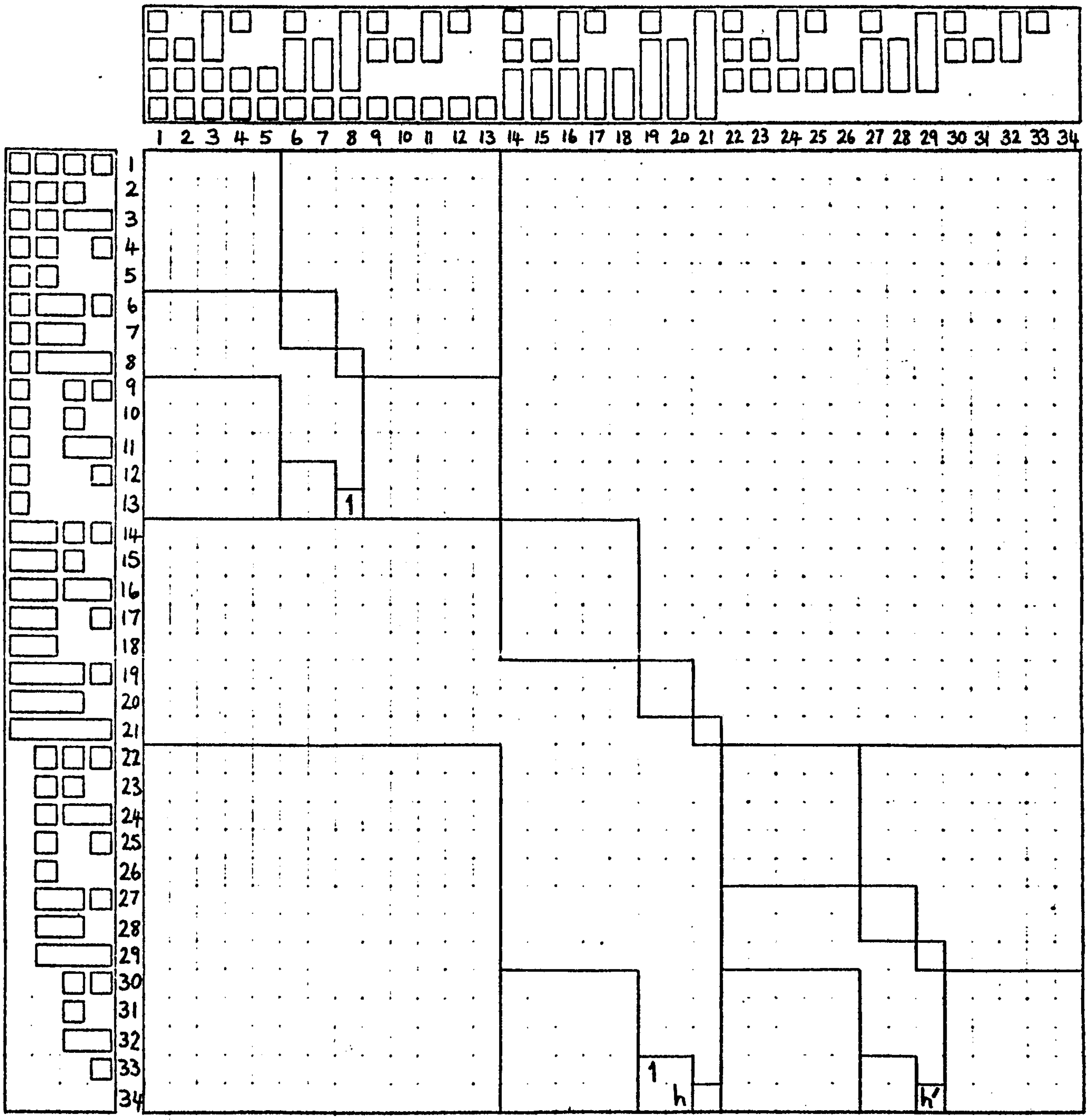


Figure 5.11 Basic allocation matrix A_3 N=4, general allocation algorithm.

The entries in this matrix are probabilities.

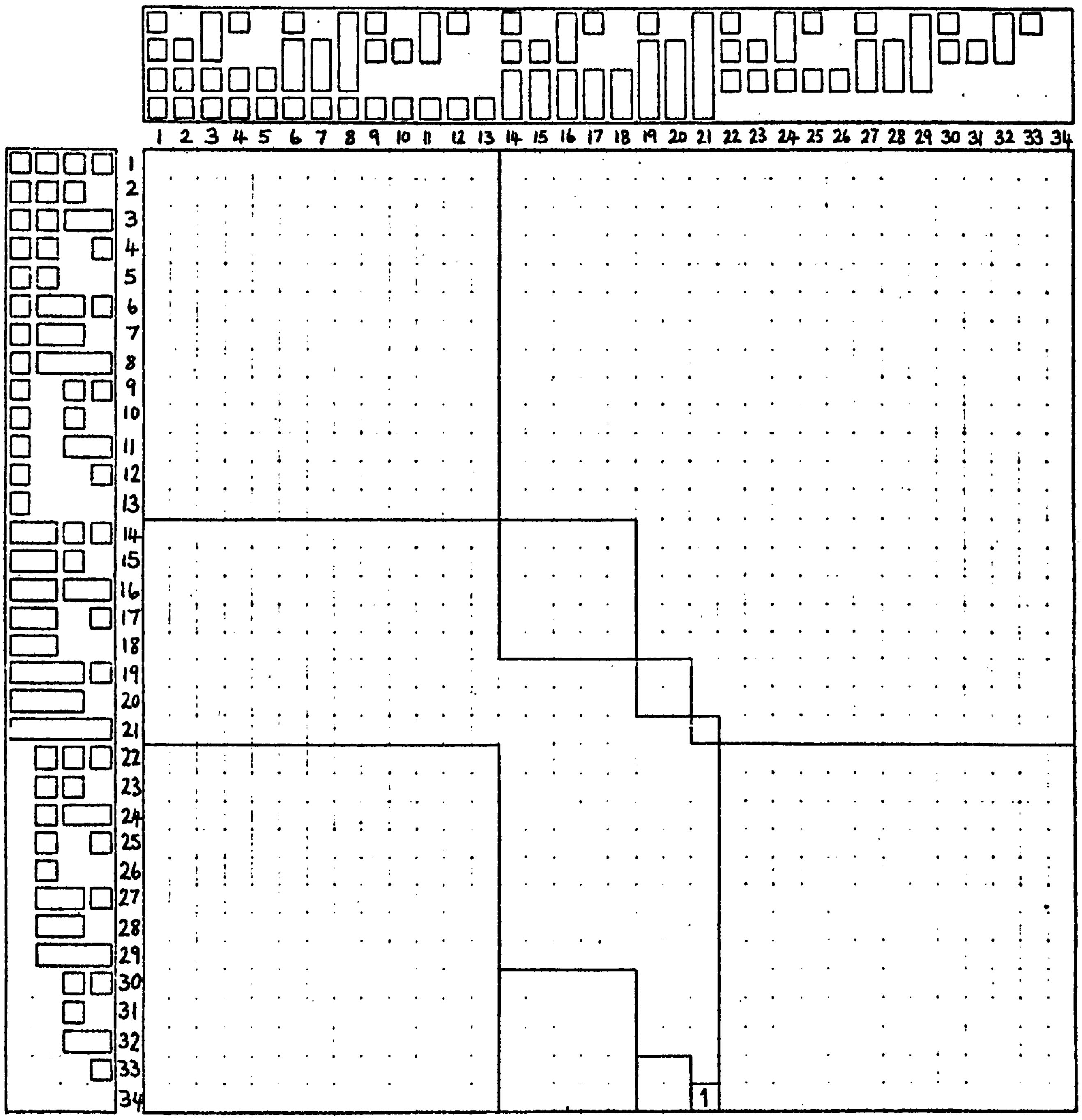


Figure 5.12 Basic allocation matrix A_4 N=4, general allocation algorithm.

The entry in this matrix is a probability.

example, there are two such states in A_3, figure 5.11, those which already contain one 1-word block at one end of memory. States which contain no gap sufficiently large to allocate the request give rise to completely zero rows. For general N, the basic allocation probability matrices $A_1,\ldots,A_N$ encompass all the possible allocation algorithms which are being considered here, the variables taking on particular values for any given algorithm. For example, the first fit algorithm determines that $h = 1$, $h' = 0$ in A_3, figure 5.11.

Figure 5.7 has a simple recursive structure which is set out in figure 5.13. Allocation transitions from states beginning with a k-word block obviously map onto the same subset of states, hence the square regions on the upper part of the main diagonal of figure 5.13 marked 3, 2, 1, 0 (for $N = 4$). For such a group of states the allocation transitions occur entirely within the last $(N-k)$ words so that each block is a smaller complete allocation transition matrix contained recursively within the original matrix. Transitions from states with an unallocated first word may or may not cause the first word to be allocated. Those that do not, again form a square submatrix on the diagonal which recursively has the same pattern as an $(N-1)$-word memory allocation matrix. Transitions which do allocate the first word to a block of 1, 2, ... words form the submatrices below the main diagonal marked 1, 2, ... The state ordering ensures that these submatrices are diagonal.

It should be clear from figure 5.13 how the rigorous argument justifying this recursive structure is constructed. The argument will not be explained in such detail as the corresponding argument for the deallocation matrix was in section 5.1.1, partly to shorten the description but mainly because it is in fact the same as for the deallocation case, but in reverse. Comparison of figures 5.1, 5.4 with 5.7, 5.13 reveals that as far as their pattern is

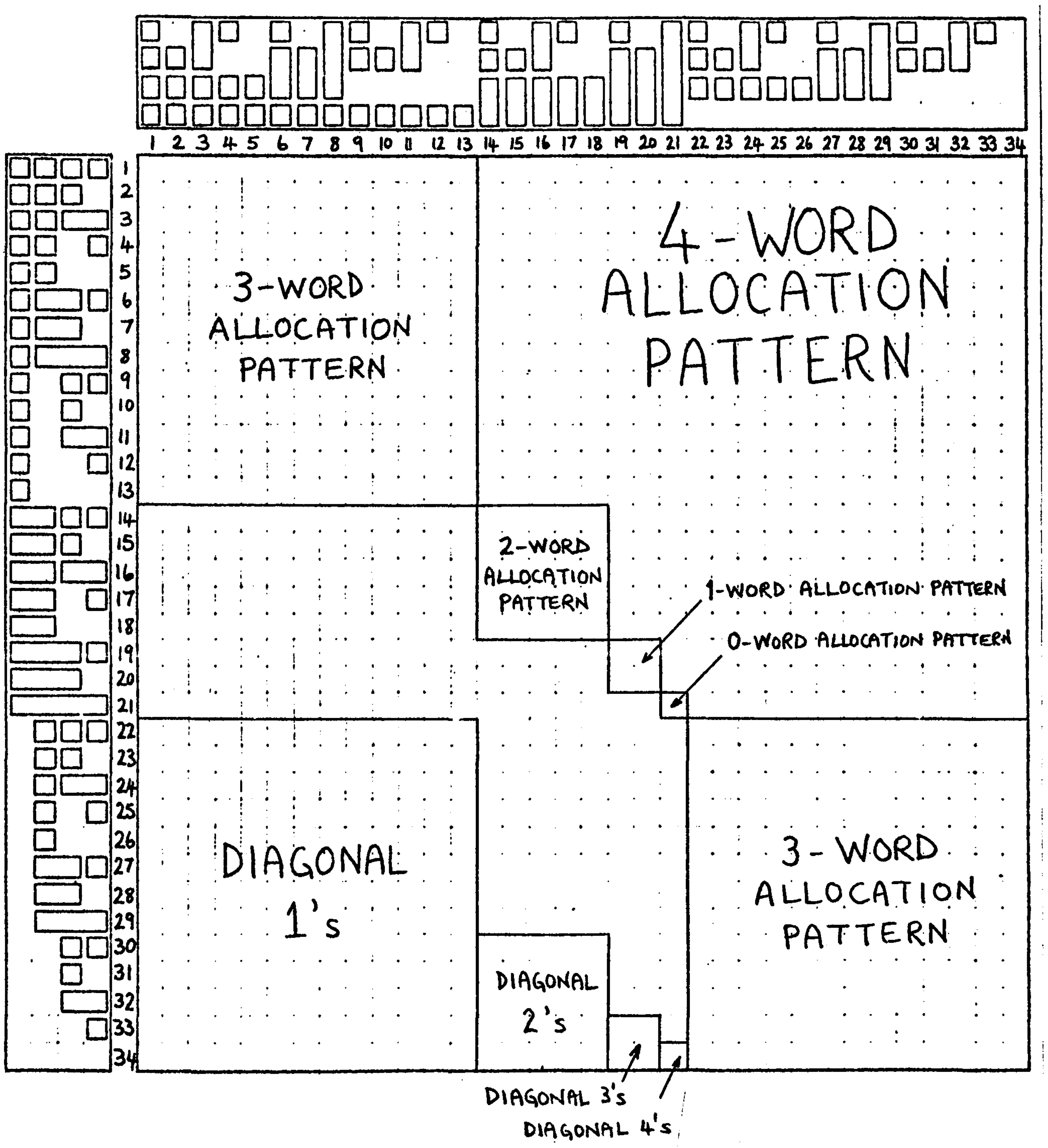

3-WORD ALLOCATION PATTERN
4-WORD ALLOCATION PATTERN
2-WORD ALLOCATION PATTERN
1-WORD ALLOCATION PATTERN
0-WORD ALLOCATION PATTERN
DIAGONAL 1's
3-WORD ALLOCATION PATTERN
DIAGONAL 2's
DIAGONAL 3's
DIAGONAL 4's

Figure 5.13 Recursive structure of the generalised allocation matrix pattern, figure 5.7 for N=4.

The pattern is composed recursively of similar smaller patterns and diagonals.

concerned one is the transpose of the other. Of course this must be so, as the former include all possible deallocations of one block for all possible states, while the latter include all possible allocations of one block into all states. Just as the deallocation matrix D is upper triangular, each matrix A_n is strictly lower triangular with the given state ordering.

5.1.3 Basic allocation termination matrix

Corresponding to each basic allocation probability matrix A_n, define a diagonal matrix T_n as follows. For a given request size n, $n = 1,\dots,N$, T_n contains the value 1 on the diagonal for all rows for which the corresponding configuration of memory contains no gap large enough to allocate the request, that is, all gaps in the configuration have size $< n$. All other entries of T_n are zero. T_n represents the unsuccessful allocation attempts, T for terminal, those in which the request is tried but does not fit in memory. The state of memory is unchanged, hence the 1 on the main diagonal, and no more allocations will be attempted. A_n represents all the successful allocations, subsequent to which more allocations will be attempted. For a given request size n, the sum (T_n+A_n) describes completely all the possible transitions the model may make when allocation of the request is attempted, including the identity transitions for all the cases in which it is not possible.

By applying definition 3.4 of the request distribution (r_n) and the theorem of conditional probability, the transition probability matrix for a single allocation step which is not the first, and so for which the size of the request is drawn from the (r_n) distribution, is

$$\begin{aligned} & r_1(T_1+A_1) + r_2(T_2+A_2) + \ldots + r_N(T_N+A_N) \\ & \quad = T + A \text{ , } \quad \text{where} \\ & T = r_1T_1 + r_2T_2 + \ldots + r_NT_N \\ & A = r_1A_1 + r_2A_2 + \ldots + r_NA_N \end{aligned} \qquad \ldots.\ 5.9$$

T, the allocation termination matrix, is diagonal and A, the allocation probability matrix, is strictly lower triangular. As an example with N = 4 figure 5.14 shows T, and A for the first fit allocation algorithm. The matrix A depends in general on the choice of allocation algorithm. T is independent of this choice, since for each n, T_n has non-zero elements only where a request of size n will not fit.

5.1.4 Eventual termination of the allocation process

This section shows that, corresponding to attempting after a deallocation in the model to continue to allocate requests until it is no longer possible, forming repeated products of allocation probability matrices eventually leads to transition probability matrices which are all zero. As a result, it is possible to write down completely the probabilities of all the multiple transitions which can occur in terms of finite products of the allocation probability matrix A with the allocation termination matrix T.

From the definition in section 5.1.2 of the A_n matrices, any matrix product ending with A_n , n = 1,...,N , and consequently A by equation 5.9, can have non-zero elements only in columns corresponding to states with at least one block allocated. See figure 5.15 which illustrates the N=4 case. Products ending with two A's, i.e. $\ldots A_iA_j$, any $1 \leq i,j \leq N$, can have non-zero elements only in columns corresponding to states with at least two blocks allocated, and so on. Products of more than N A-matrices must be

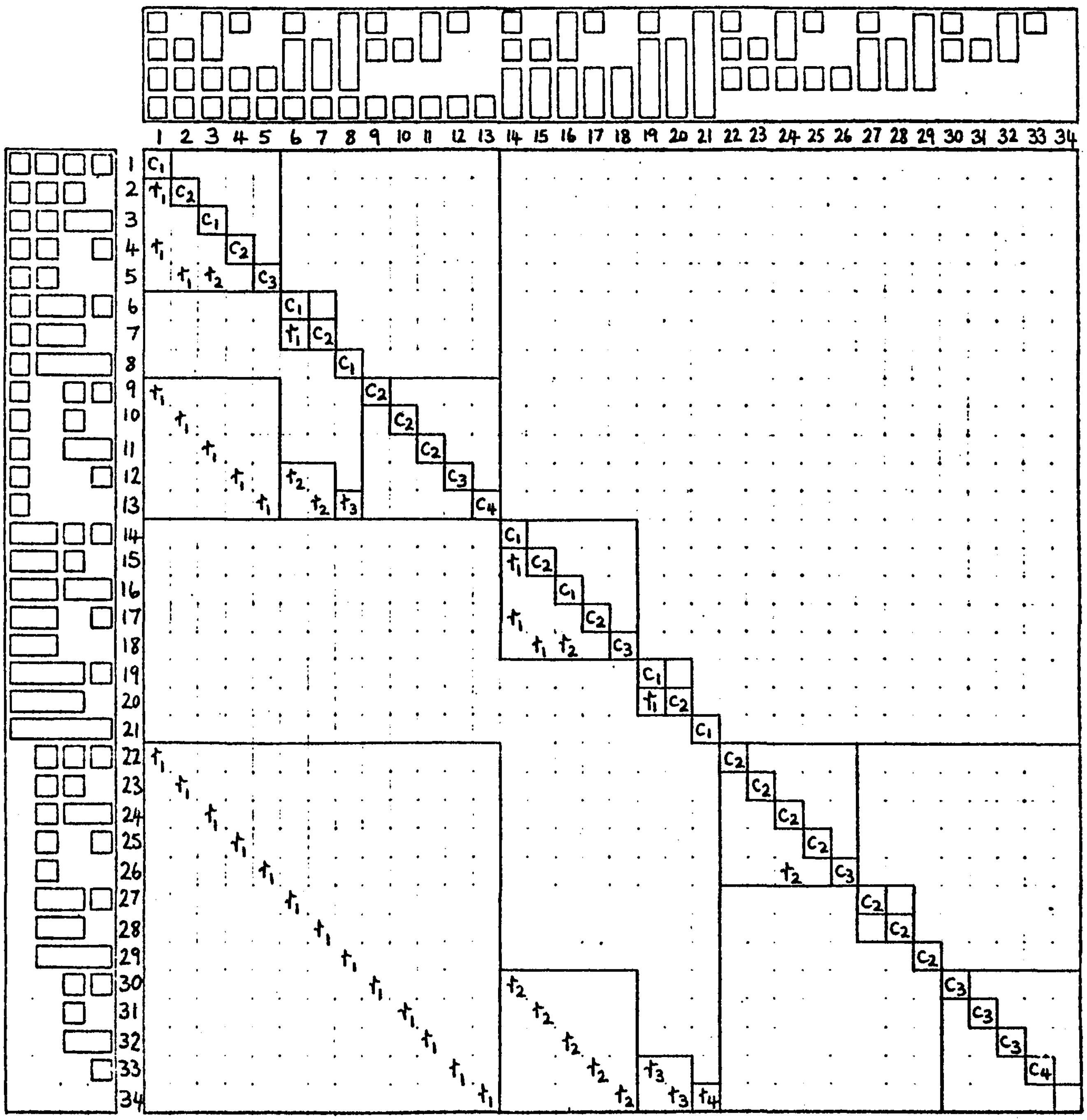


Figure 5.14 Diagonal: <u>T = termination matrix,</u>
Lower triangle: <u>A = first fit allocation matrix,</u> N=4.

This figure contains two separate but complementary matrices, for convenience. Both contain probabilities.

$T = r_1T_1 + r_2T_2 + \dots + r_NT_N$ $A = r_1A_1 + r_2A_2 + \dots + r_NA_N$

r_n = Probability[new request joining the queue is for n words]
$c_n = r_n + \dots + r_N$, so that $c_1 = 1$, $c_N = r_N$ always

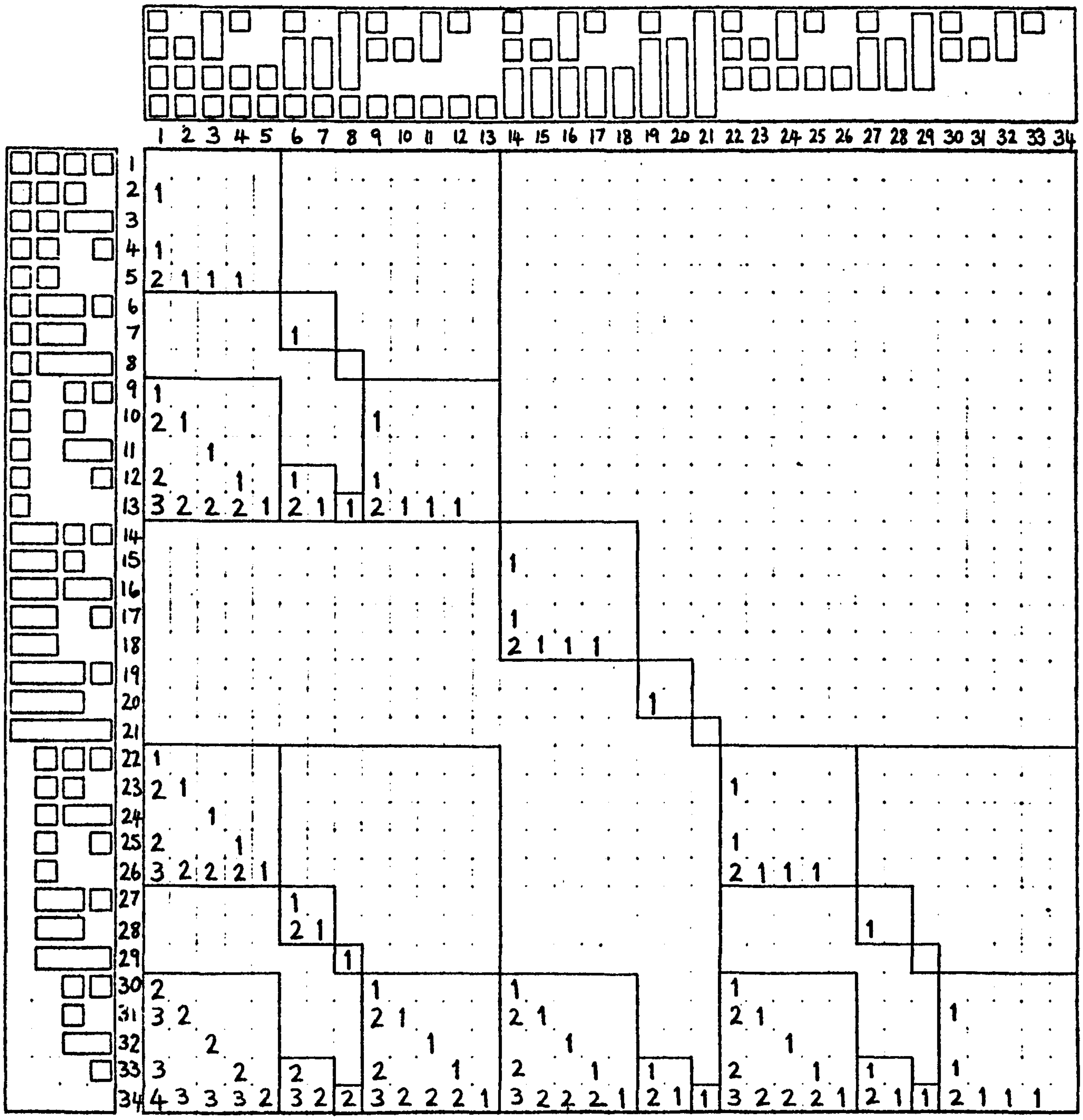


Figure 5.15 Powers of allocation matrices (silhouettes), N=4

This matrix does not contain probabilities. A 1-digit appears in the i,j-th position if state j can be reached from state i by allocating just one block, of any size. A 2-digit appears at i,j if it is possible to get to j from i by allocating two blocks of any size, etc. All possibilities are shown, for N=4.

identically zero, in particular:

$$A^{N+1} = A^{N+2} = \ldots = 0 \qquad \ldots . \; 5.10$$

since no state can have more than N blocks allocated. In other words, it is impossible to make more than N successive allocations, and the only way to make as many as N is to start from the empty state and allocate N one-word blocks.

Another way of considering this limit on the non-zero powers of A is illustrated by figures 5.15, 5.16, which concentrate on the positions of the non-zero elements of any of the A_n (or A) matrices, their actual values being unimportant. As pointed out in section 5.1.2 any allocation matrix for an N-word memory has non-zero elements taken from a recursive pattern, see figure 5.13 for example. Let the ordinary incidence matrix of the generalised allocation incidence matrix defined in that section be denoted AI_N, for an N-word memory. That is, AI_N contains a 1 entry in all positions (i,j) where some allocation probability matrix A could have a non-zero transition probability from i to j, and zeroes everywhere else. Then AI_N includes as part of itself the pattern AI_{N-1} for an (N-1)-word memory, and AI_{N-2} and so on, as well as identity (diagonal) submatrices below the main diagonal, arranged as shown in figure 5.16. Raising any such matrix to successively higher powers results in matrices with incidence patterns as shown, coefficients having been left out as they affect only the values and not the positions of the non-zero elements. By inspection of the multiplication in the general case, as shown in the figure, the recursive block structure is retained and the non-zeroes do not spread into the other areas of the matrix. Consequently it is easy to see by induction that if powers of AI_{N-1} greater than N-1 are identically zero, then powers of AI_N greater than N must also be, and that this is trivially true for N = 1,2,3 say.

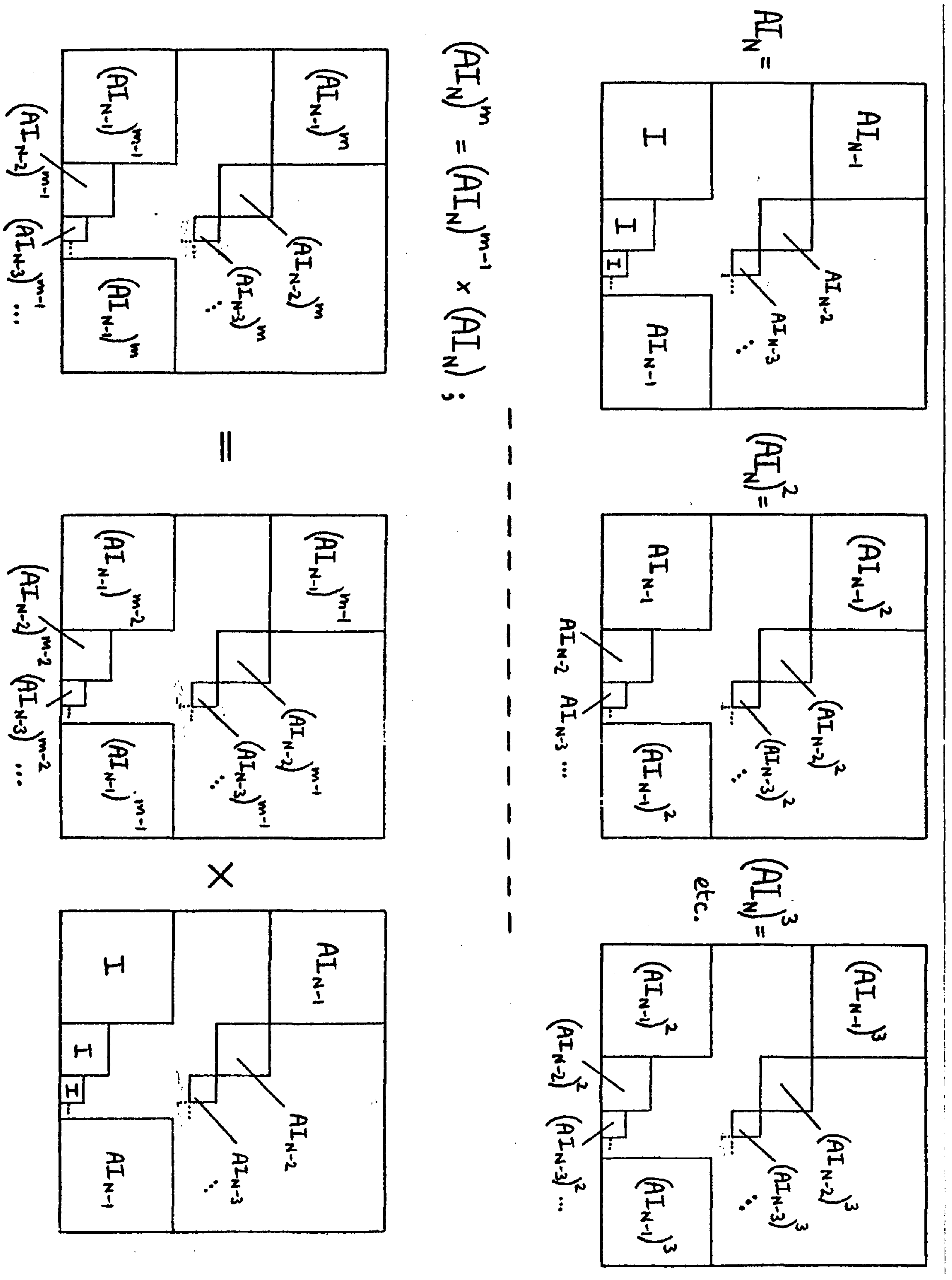

$AI_N =$
AI_{N-1}
I
AI_{N-2}
AI_{N-3}
$(AI_N)^2 =$
$(AI_{N-1})^2$
$AI_{N-2}\ AI_{N-3}\ \ldots$
$(AI_N)^3 =$
$(AI_{N-1})^3$
$(AI_{N-2})^2\ (AI_{N-3})^2\ \ldots$
etc.
$(AI_N)^m = (AI_N)^{m-1} \times (AI_N)$;
$(AI_{N-1})^m$
$(AI_{N-1})^{m-1}$
$(AI_{N-2})^{m-1}\ (AI_{N-3})^{m-1}\ \ldots$
$(AI_{N-2})^m$
$(AI_{N-3})^m$
$(AI_{N-1})^{m-2}$
$(AI_{N-2})^{m-2}\ (AI_{N-3})^{m-2}\ \ldots$
$(AI_{N-3})^{m-1}$

Figure 5.16 Powers of generalised allocation incidence matrix AI

Incidence patterns only, ignore values of non-zero elements.

5.1.5 Allocation of first queued request

As implied in section 5.1.3, the one-step allocation matrix

$$(T+A) = \sum_{n=1}^{N} r_n(T_n+A_n) \qquad \text{.... 5.9}$$

does not apply to the first allocation after the deallocation. This is because the probability distribution of the size of the request at the head of the queue, q, is not in general the same as (r_n), as explained in section 3.2.4. To overcome this difficulty the diagonal matrices

$$Q_n = (q_n^{(i)}) \ , \qquad n = 1,\ldots,N \qquad \text{.... 5.11}$$

are introduced, where for each state i = 1,...,S(N) down the main diagonal

$$q_n^{(i)} = \text{Probability}[q = n \text{ in state } i] \ .$$

By section 3.2.4 these diagonal matrices are known and depend only on the request distribution (r_n) :

$$\begin{aligned} q_n^{(i)} &= 0 && \text{if } n \leq g_i \\ &= r_n/c_{g_i+1} && \text{if } n > g_i \end{aligned} \qquad \text{.... 3.8}$$

where g_i = maximum gap in state i,

and $c_k = r_k + \ldots + r_N \ , \qquad k = 1,\ldots,N$ 3.9

As an example, the four Q_n diagonals for N = 4 are shown diagrammatically in figure 5.17.

5.1.6 Algebraic expansion of the transition matrix P

With the results of the previous sections established, this expansion can now be written down. It follows the general plan:

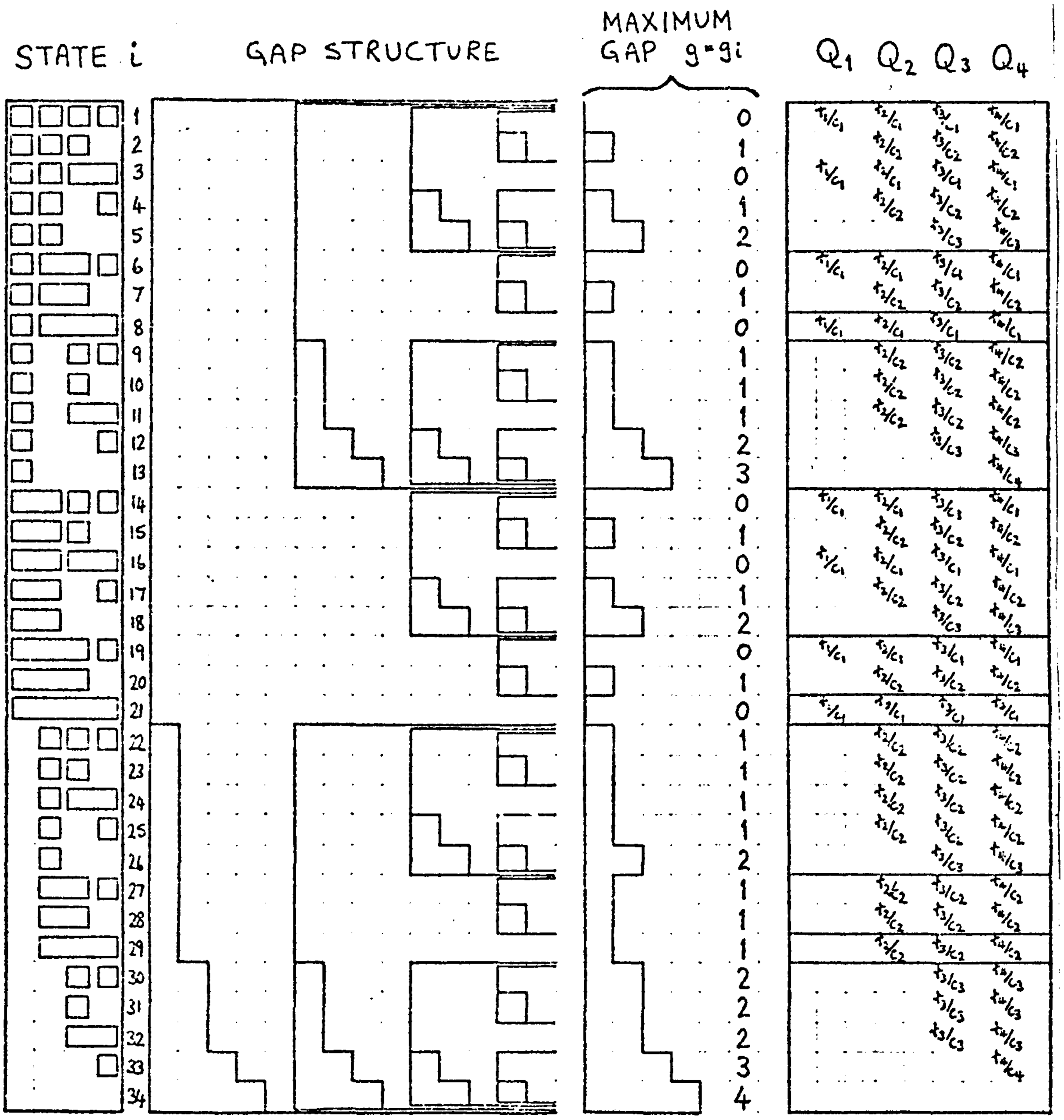


Figure 5.17 First queued request matrices Q_1, Q_2, Q_3, Q_4 for N=4

Each Q_n matrix is diagonal and square. To save space, the diagonals are here shown as columns.

r_n = Probability[new request joining queue is for n words] $c_n = r_n + \ldots + r_N$

Each entry $q_n^{(i)}$ is a probability, dependent on i as well as (r_n):

$q_n^{(i)}$ =Prob[first queue request in state i is for n words]. Blanks are all zero

- Pick a starting state (down the left hand side of the deallocation matrix D).
- Calculate the transition (deallocation) probabilities from this state (corresponding row of D).
- Post-multiply this by some allocation matrix to obtain the corresponding row of matrix P, i.e.

$$D \times \text{allocation matrix} = P \ .$$

The transition probability matrix P can now be specified exactly, in several steps:

1) Pick a starting state i (configuration of memory).

2) Fix on the size q of the first queued request. Consider the N possibilities, $q = n$, $n = 1,\dots,N$ in turn, with probabilities from the $(q_n^{(i)})$ distribution. Some of these probabilities will be zero.

3) For each of these possibilities $q = n$, calculate the deallocation transition probabilities from the starting state i. This gives the i-th row of the deallocation matrix D, multiplied by $q_n^{(i)}$. Taking all the states i together gives N probability matrices Q_nD for $n = 1,2,\dots,N$.

4) For each of the $q = n$ possibilities, calculate the transition probabilities up to the end of the first allocation step, by post-multiplying by the basic transition probability matrix for a single allocation step (T_n+A_n) corresponding to the value n of q. Here the benefit of knowing q is obtained at the previous expense of splitting into N distinct possibilities. Resulting from this are the N distinct probability matrices

$$Q_nD(T_n+A_n), \qquad n = 1,\dots,N$$

5) At this point the N possibilities can be recombined (added) as subsequent

allocations depend on requests in the original queue after the first, which are independent of q due to the first come first served queue discipline and independence of successive queued requests. The result of this addition is a single matrix:

$$\sum_{n=1}^{N} Q_n D(T_n+A_n) = \sum_{n=1}^{N} Q_n DT_n + \sum_{n=1}^{N} Q_n DA_n$$

6) The cumulative probabilities $\Sigma Q_n DA_n$ so far calculated, i.e. those which represent successful first-step allocations, are further multiplied by the one-step allocation matrix (T+A) to accumulate the transition probabilities up to the second allocation step:

$$\sum_{n=1}^{N} Q_n DT_n + \left(\sum_{n=1}^{N} Q_n DA_n \right)(T+A)$$

Corresponding to continuing to attempt to allocate requests as long as they continue to fit, this process of multiplying the part of the allocation matrix which represents successful allocations so far by the one-step allocation matrix (T+A) is repeated, in principle indefinitely. In fact, the process can be stopped after N allocations. Physically, no more can be possible. Algebraically, further multiplications by (T+A) factors has the identity effect, since by section 5.1.4

$$\text{if } n > 1, \quad A_n A^{N-1} = 0 = A_n A^{N-1} T$$
$$\text{if } n = 1, \quad A_1 A^{N-1} = A_1{}^N$$

and so post-multiplication by T again has the identity effect;

$$= A_1{}^N T \ .$$

Eventually therefore, P is reached,

$$P = \sum_{n=1}^{N} Q_n DT_n + \left(\sum_{n=1}^{N} Q_n DA_n \right)(T+A(T+A(\ldots(T+A)\ldots))) \qquad \ldots.\ 5.12$$

where there are N-1 factors T+A .

Equation 5.13 for P follows immediately:

$$P = \sum_{n=1}^{N} Q_n D T_n + \left(\sum_{n=1}^{N} Q_n D A_n \right)(I+A+A^2+\ldots+A^{N-1})T \qquad \ldots.\ 5.13$$

$$= \sum_{n=1}^{N} Q_n D T_n + \left(\sum_{n=1}^{N} Q_n D A_n \right)(I+A+A^2+\ldots+A^{N-1}+A^N+\ldots)T$$

5.1.7 Summary of section 5.1 : transition matrix expansion

Refer again to figures 4.1 to 4.7 for examples of the transition probability matrix P, especially figure 4.4 for N=4 which follows from figures 5.1, 5.7, 5.8, 5.14, 5.17. As mentioned in the introduction to this chapter, the elements of the simpler matrices Q_n, D, T_n, A_n, A, T which appear in the expansions 5.12, 5.13 of P are arranged in simple recursive patterns the general form of which can be extended for any memory size N. These simple matrices are either diagonal (Q_n, T_n, T) or else very sparse indeed for increasing N, generally much less than N entries per row in a square matrix of order $S(N) = f_{2N}$, $= 0.72 \times (2.6)^N$ approximately. The apparent complicatedness of P arises therefore mainly from what complexity there is in equations 5.12, 5.13. Reducing this complexity and making use of these equations, is the subject of the next section 5.2.

5.2 Algebraic simplification of the steady state equation

The expression for P so far derived (5.12 or 5.13) as sums and products of simple matrices, is complicated enough that it is unlikely that any useful properties could be derived from it as it stands. However, by good fortune, the eigenvector equation:

$$\underline{\pi} = \underline{\pi}\, P \qquad \text{.... 3.11}$$

does simplify considerably and somewhat unexpectedly. Two preliminary steps are necessary.

5.2.1 The allocation inverse matrix

Any allocation transition probability matrix A_n, and so in particular A itself from equation 5.9, is strictly lower triangular because of the choice of ordering of the states. Hence the matrix I-A is non-singular and has an inverse, $(I-A)^{-1}$. In any case since $A^{N+1} = 0$ from equation 5.10,

$$(I + A + A^2 + \ldots + A^N)\,(I-A) = I$$

so $$(I-A)^{-1} = (I + A + A^2 + \ldots + A^N) \qquad \text{.... 5.14}$$

5.2.2 The pseudo-inverse of the allocation termination matrix

T is a diagonal matrix, but not all the elements on the main diagonal are non-zero. From section 5.1.3, each T_n, n = 1,...,N , has elements $t_n^{(i)}$ on the diagonal:

$$T_n = \text{diag}(t_n^{(i)}) , \quad (i = 1,\ldots,S; \ T_n \text{ is square of order } S \times S)$$

where

$$\begin{aligned} t_n^{(i)} &= 0 \quad \text{if } n \leq g_i \\ &= 1 \quad \text{if } n > g_i \end{aligned} \qquad \ldots. \ 5.15$$

and g_i = maximum gap in state i .

Let $T = \Sigma r_n T_n = \text{diag}(t_i)$ have elements t_i on its diagonal, i = 1,...,S . Then from equations 5.9, 5.15,

$$\begin{aligned} t_i &= r_{g_i+1} + \ldots + r_N \\ &= c_{g_i+1} , \quad \text{from definition 3.9.} \end{aligned}$$

T is thus a diagonal matrix whose main diagonal elements c_{g_i+1} have values from the (descending) cumulative probability distribution (c_k), k = 1,...,N . T is shown for N=4 in figures 5.14 and 5.18. In particular, the last (or S-th, i.e. bottom right) element is identically zero since it is impossible not to be able to allocate a block into the empty state (algebraically, $g_S = N$), and others may be zero depending on the request distribution. For example, if $r_{N-1} > 0$, $r_N = 0$; then $c_N = 0$ although $c_k > 0$ for $k < N$. For the time being, assume that all the c_i, i = 1,...,N are non-zero (or, which is the same thing, that $r_N > 0$) and do not worry about the bottom right corner, the empty state. Define T' to be a diagonal matrix with main diagonal elements which are the reciprocals of the corresponding elements of T:

$$T' = (1/t_i) , \quad i = 1,\ldots,S'=S-1 \qquad \ldots. \ 5.16$$

except for the last (i=S) which will turn out to be immaterial, and so may as well be zero. Then

$$TT' = T'T = I' ,$$

where I' is the identity matrix except for a zero in the bottom right corner.

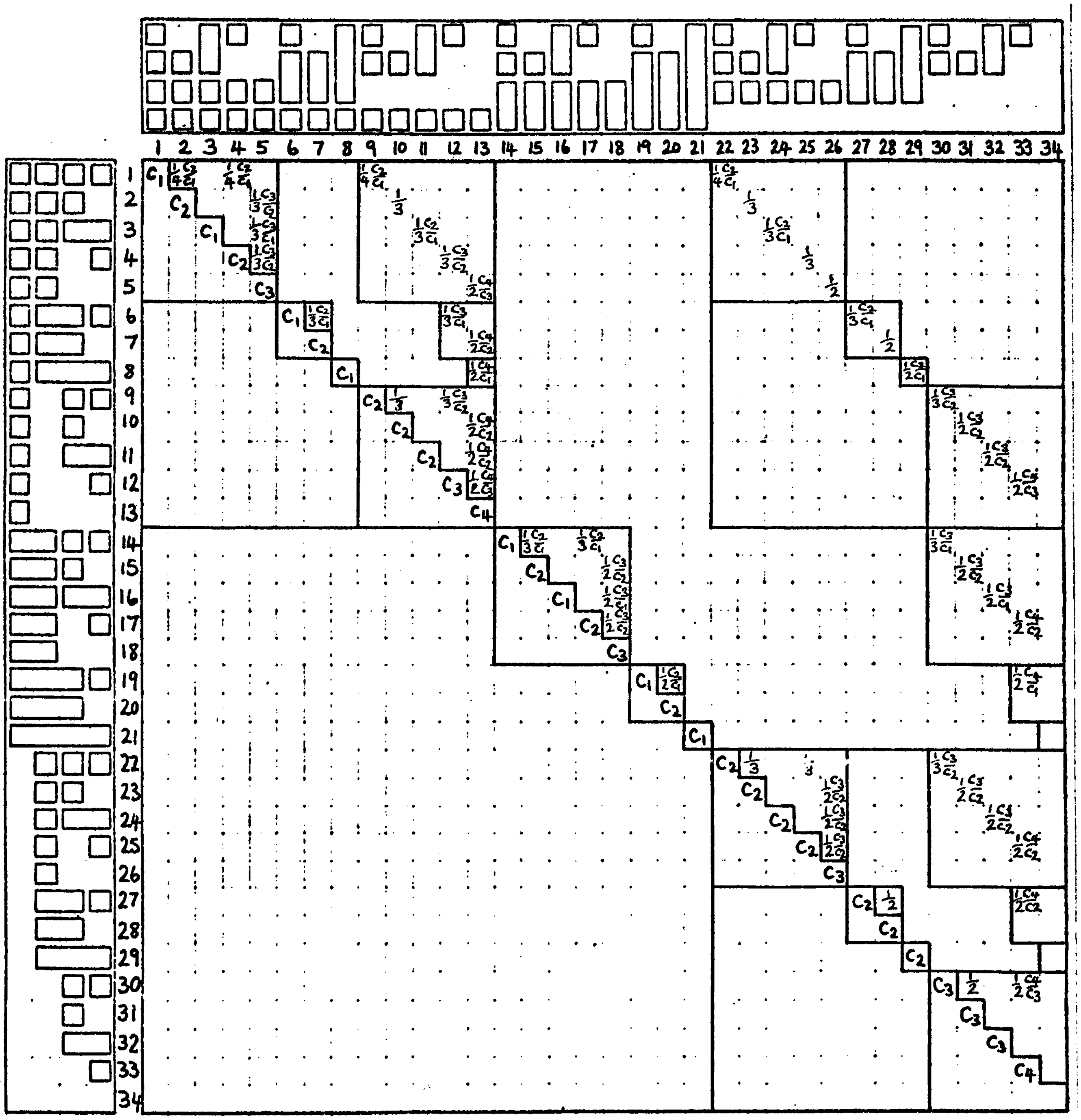


Figure 5.18 Upper triangle: $\Sigma Q_n D T_n$ (compare figures 5.17, 5.1, 5.14) for N=4.

Diagonal: T, repeated for convenience, see figure 5.14.

This figure contains two separate matrices.

T' is known as a pseudo-inverse of T.

5.2.3 Reduction of $\pi = \pi P$ to a simple form in terms of the allocation matrices

Substituting expansion 5.13 for P in the steady state equation 3.11,

$$\begin{aligned}\underline{\pi} &= \underline{\pi}(\Sigma Q_n D T_n) + \underline{\pi}(\Sigma Q_n D A_n)(I + A + A^2 + \ldots)T \\ &= \underline{\pi}(\Sigma Q_n D T_n) + \underline{\pi}(\Sigma Q_n D A_n)(I-A)^{-1}T \qquad \ldots.\ 5.17\end{aligned}$$

On the left hand side of this vector equation, the last element of $\underline{\pi}$ is zero since it represents the steady state probability of being in the empty state, which can never be possible between transitions. On the right hand side, the last column of every T_n and every A_n is identically zero, and so on multiplying throughout by T', its arbitrary last element is eliminated and the product I' = TT' can be replaced by the identity matrix:

$$\underline{\pi}T' = \underline{\pi}(\Sigma Q_n D T_n)T' + \underline{\pi}(\Sigma Q_n D A_n)(I-A)^{-1}$$

and hence

$$\underline{\pi}T'(I-A) = \underline{\pi}(\Sigma Q_n D T_n)T'(I-A) + \underline{\pi}(\Sigma Q_n D A_n)$$

Rearranging and collecting the A's together,

$$\underline{\pi}[[(\Sigma Q_n D T_n)-I]T'A - (\Sigma Q_n D A_n)] = \underline{\pi}[(\Sigma Q_n D T_n)-I]T'$$

Define

$$K = [(\Sigma Q_n D T_n)-I]T' \qquad \ldots.\ 5.18$$

K is a (square, S $\times$ S) matrix which is constant with respect to the A's. See figures 5.18, 5.19 which show K and its components. Then

$$\underline{\pi}(KA - \Sigma Q_n D A_n) = \underline{\pi}K$$

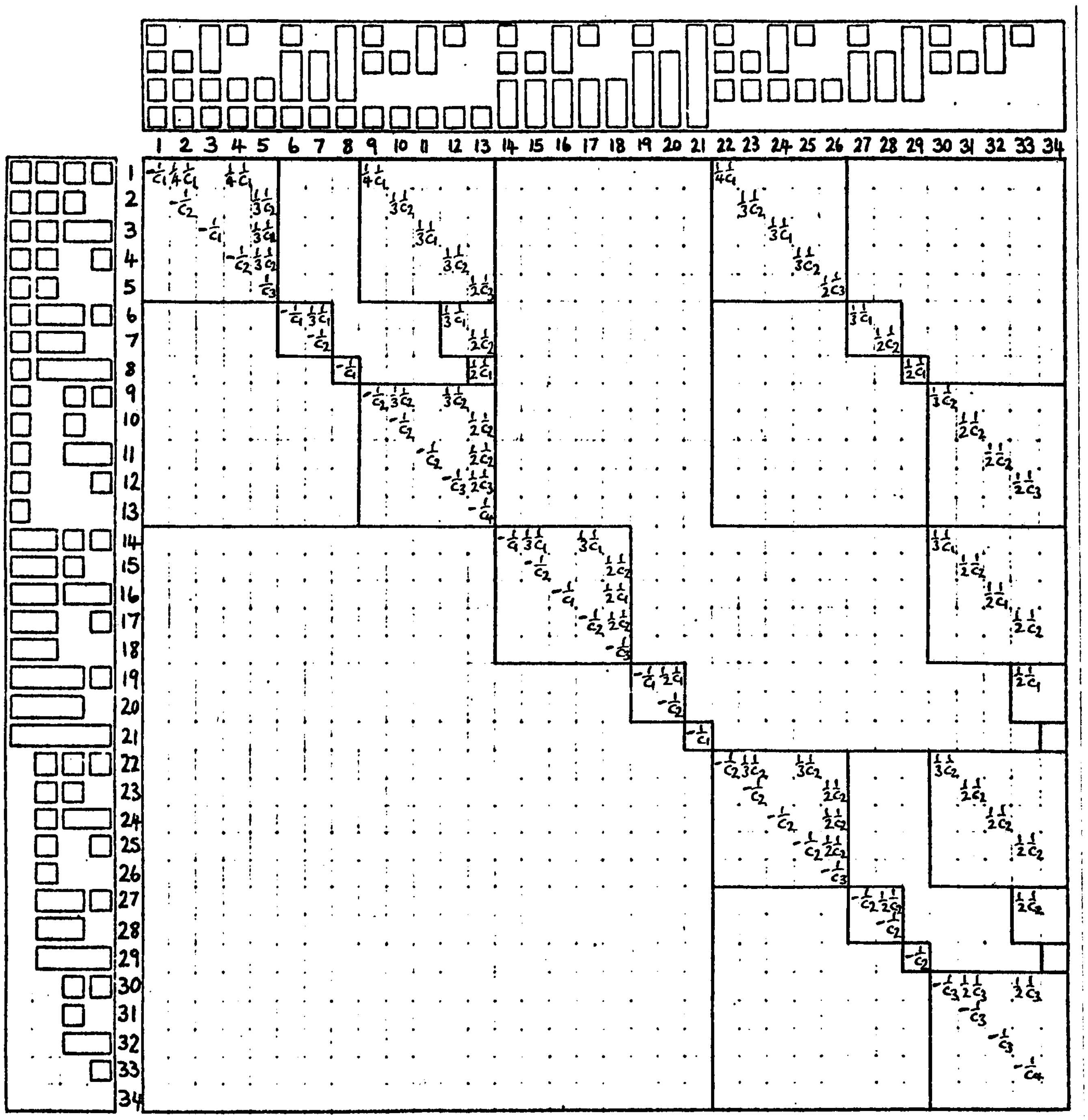


Figure 5.19 $\underline{K = ((\Sigma Q_n DT_n)-I)T'}$ for N=4

See figure 5.18 and equation 5.18; also see figure 5.20 and equation 5.21.

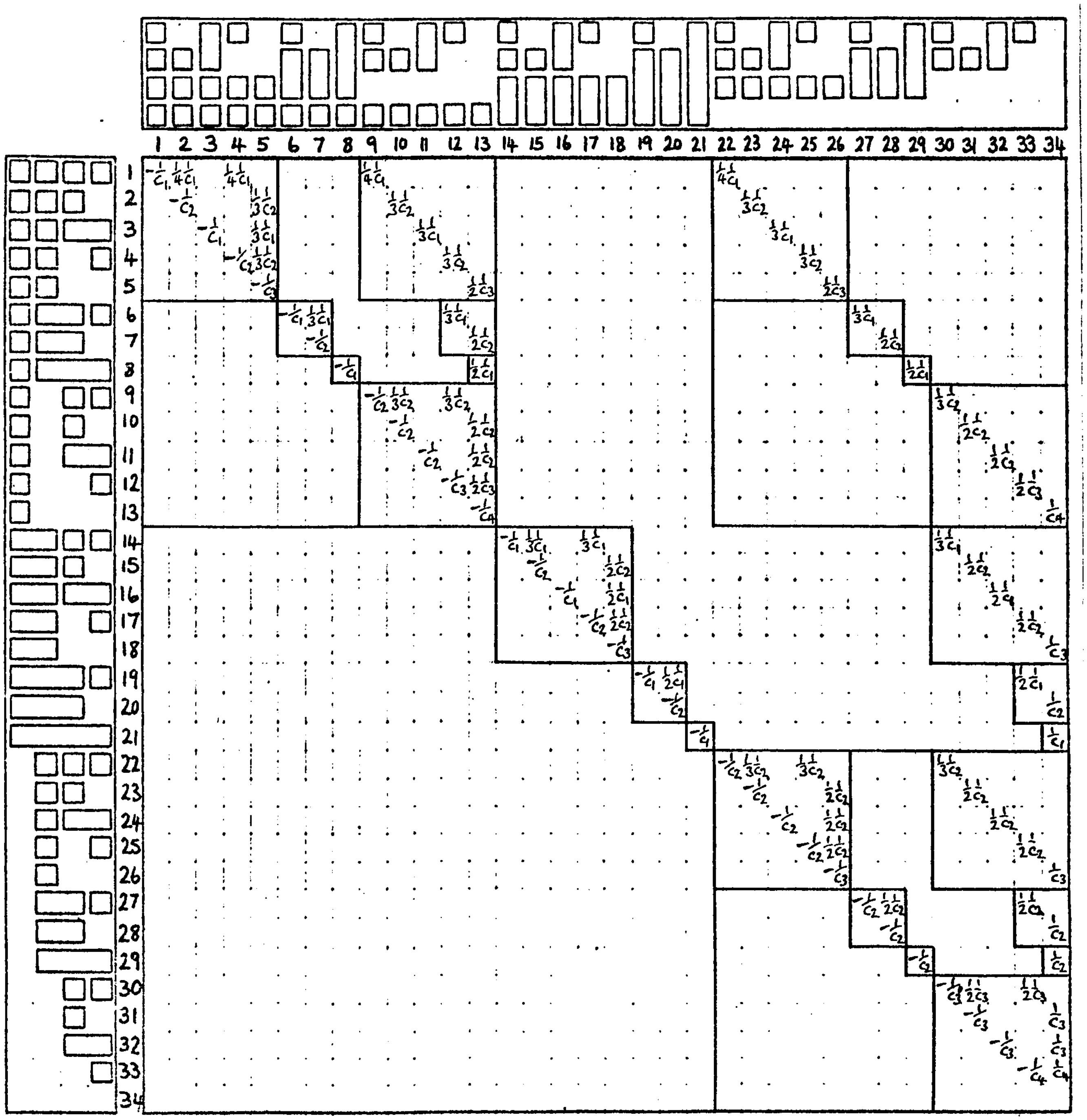


Figure 5.20 <u>T'(D-I)</u> for N=4

See equation 5.21 and section 5.2.4, which compares this matrix with that in the last figure 5.19 for $K=((\Sigma Q_n D T_n)-I)T'$; they are identical except in the last column (the empty state).
See figure 5.1 for D (deallocation) and 5.14, 5.18 for T (termination matrix)

Define

$$K_n = r_n K - Q_n D , \quad n = 1,\ldots,N \qquad \ldots .\ 5.19$$

(see figures 5.21 to 5.24), then since $A = \Sigma r_n A_n$ from definition 5.9, the steady state equation becomes

$$\underline{\pi}\left(\sum_{n=1}^{N} K_n A_n \right) = \underline{\pi} K \qquad \ldots .\ 5.20$$

This last equation looks somewhat simpler than equation 5.17. As examples, figures 5.25 to 5.27 show the matrices $\Sigma K_n A_n$, $(\Sigma K_n A_n)$-K, and K^{-1} for N=4, first fit allocation algorithm. Other matrices $(\Sigma K_n A_n)$-K can be constructed for different allocation algorithms, using the general A_n matrices shown in figures 5.9 to 5.12. Of course, some of the complexity of equation 5.17 has disappeared into the definitions of the K and K_n matrices. However there is an unexpected further simplification. It turns out that

$$K = T'(D-I) , \quad \text{except in the last column} \qquad \ldots .\ 5.21$$

Figure 5.20 shows the matrix T'(D-I) and figure 5.19 shows K, for N=4. Each K_n is also much simpler than expected, terms in $r_n K$ and $Q_n D$ cancelling each other, see figures 5.21 to 5.24.

5.2.4 Unexpectedly simple structure of the constant coefficient matrix K

Before discussing the implications of the simplification of the steady state equation to equation 5.20, it is necessary to show that the equality in equation 5.21 and the cancellation in equation 5.19 do indeed occur. The present section 5.2.4 shows the former and so reveals how the matrix K is much simpler than might be expected from its definition, and the next section 5.2.5 similarly explains why the subsequently derived K_n matrices are also simpler

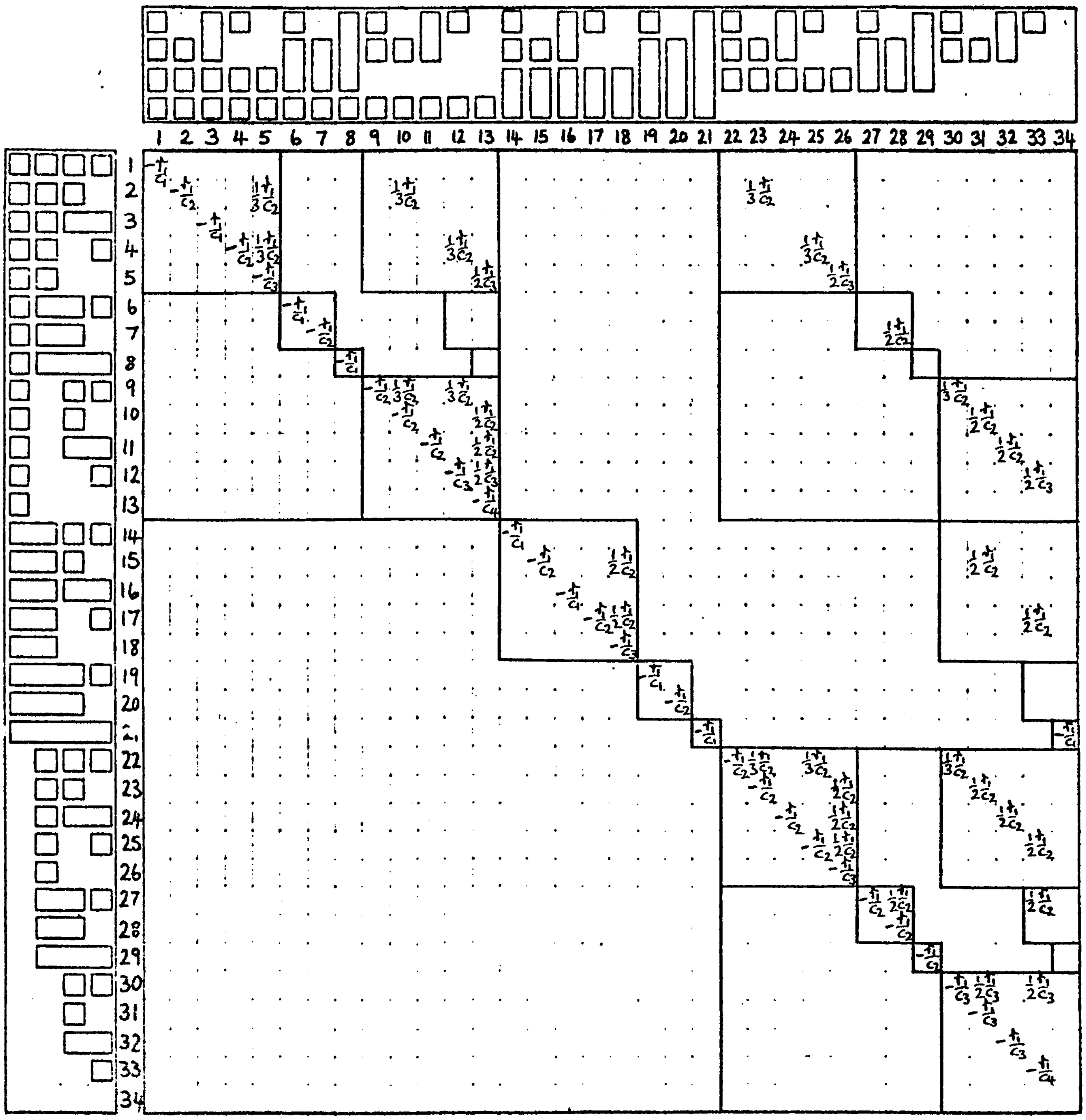


Figure 5.21 $K_1 = r_1K - Q_1D$, for N=4

See equation 5.19. Also see figures 5.19 for K, 5.17 for Q_1, and 5.1 for D. Notice the cancellation between r_1K and Q_1D.

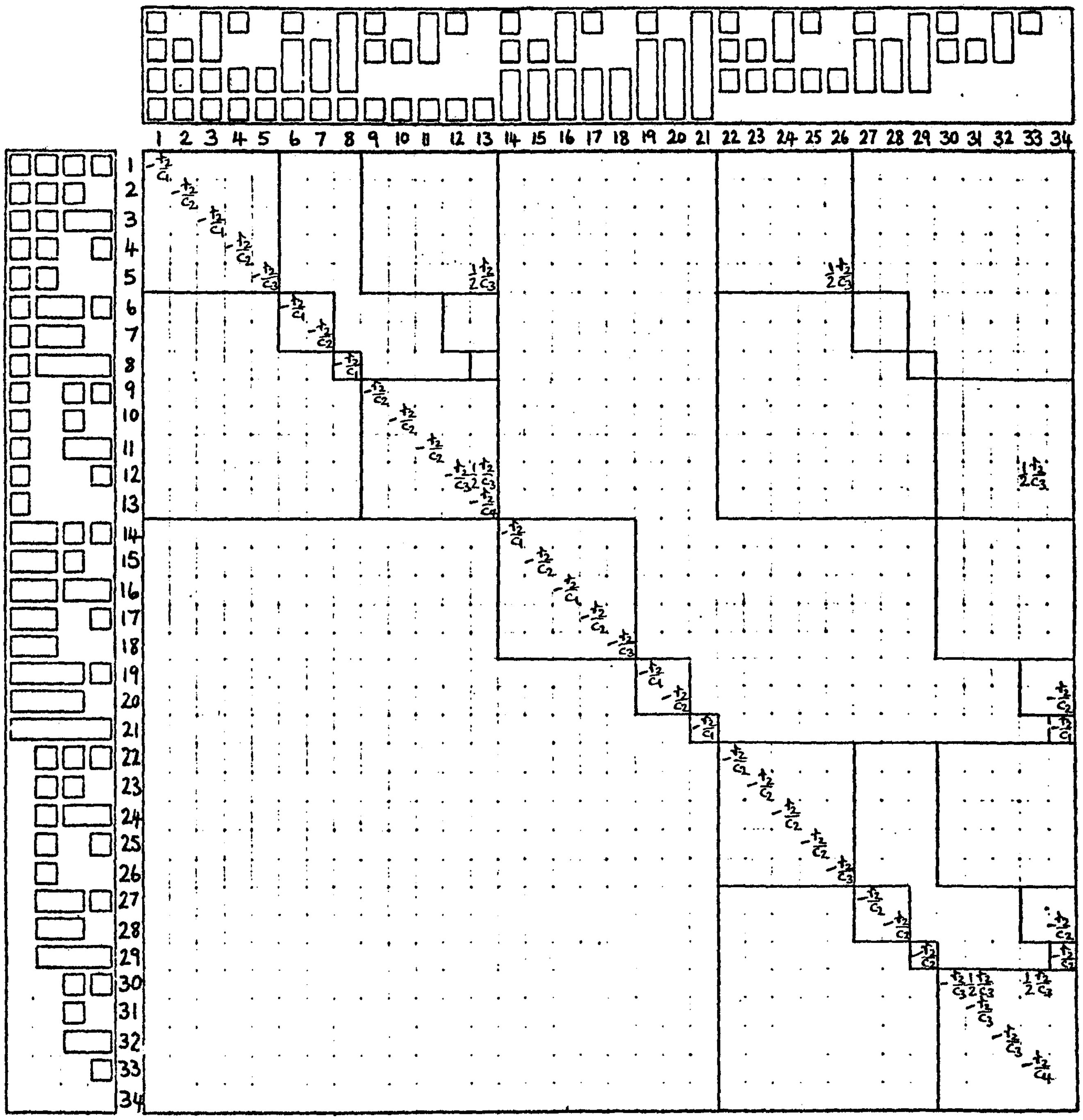


Figure 5.22 $K_2 = r_2K - Q_2D$, for N=4

See equation 5.19. Also see figures 5.19 for K, 5.17 for Q_2, and 5.1 for D. Notice the cancellation between r_2K and Q_2D.

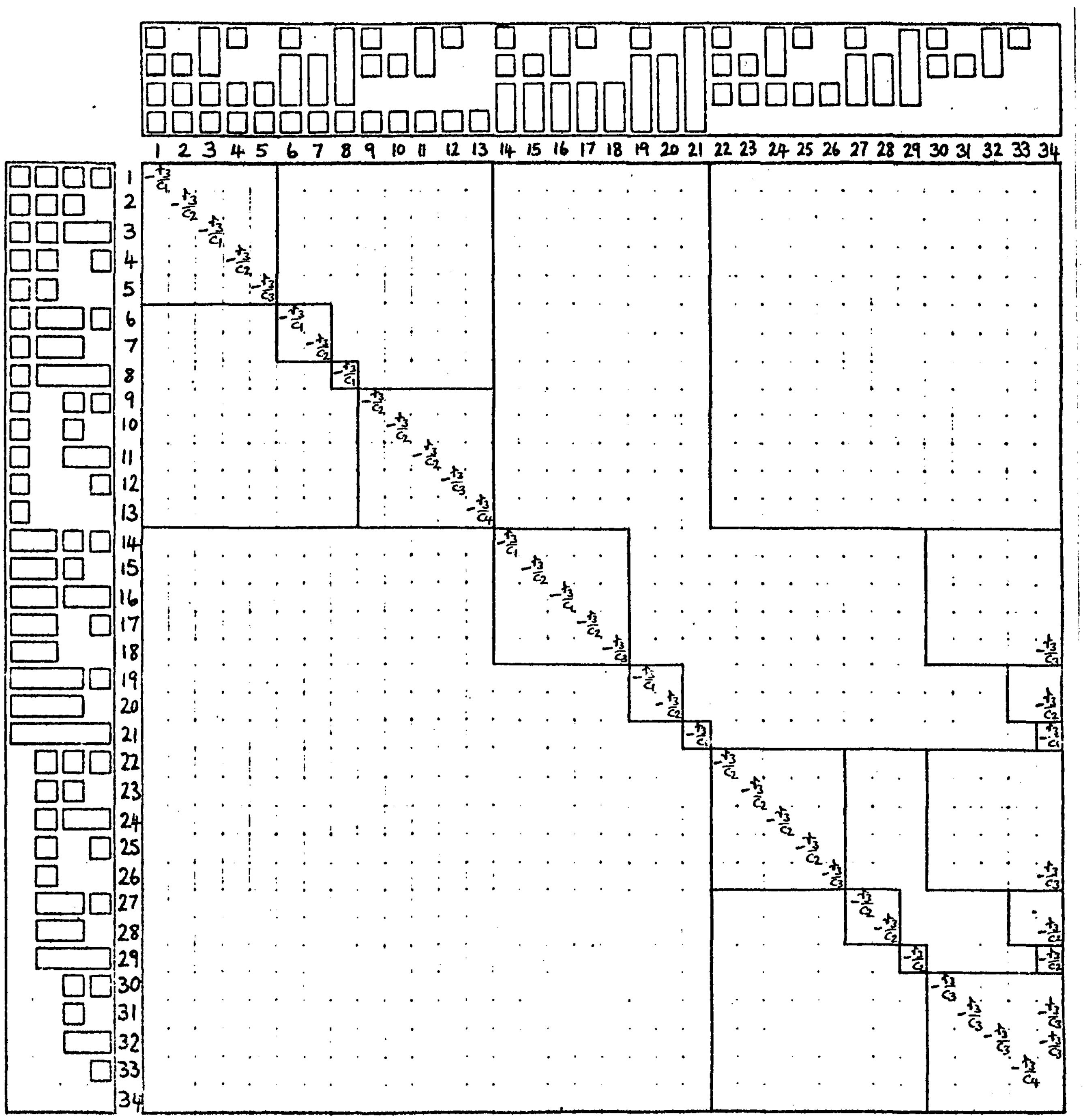


Figure 5.23 $K_3 = r_3K - Q_3D$, for N=4

See equation 5.19. Also see figures 5.19 for K, 5.17 for Q_3, and 5.1 for D. Even more cancellation between r_3K and Q_3D.

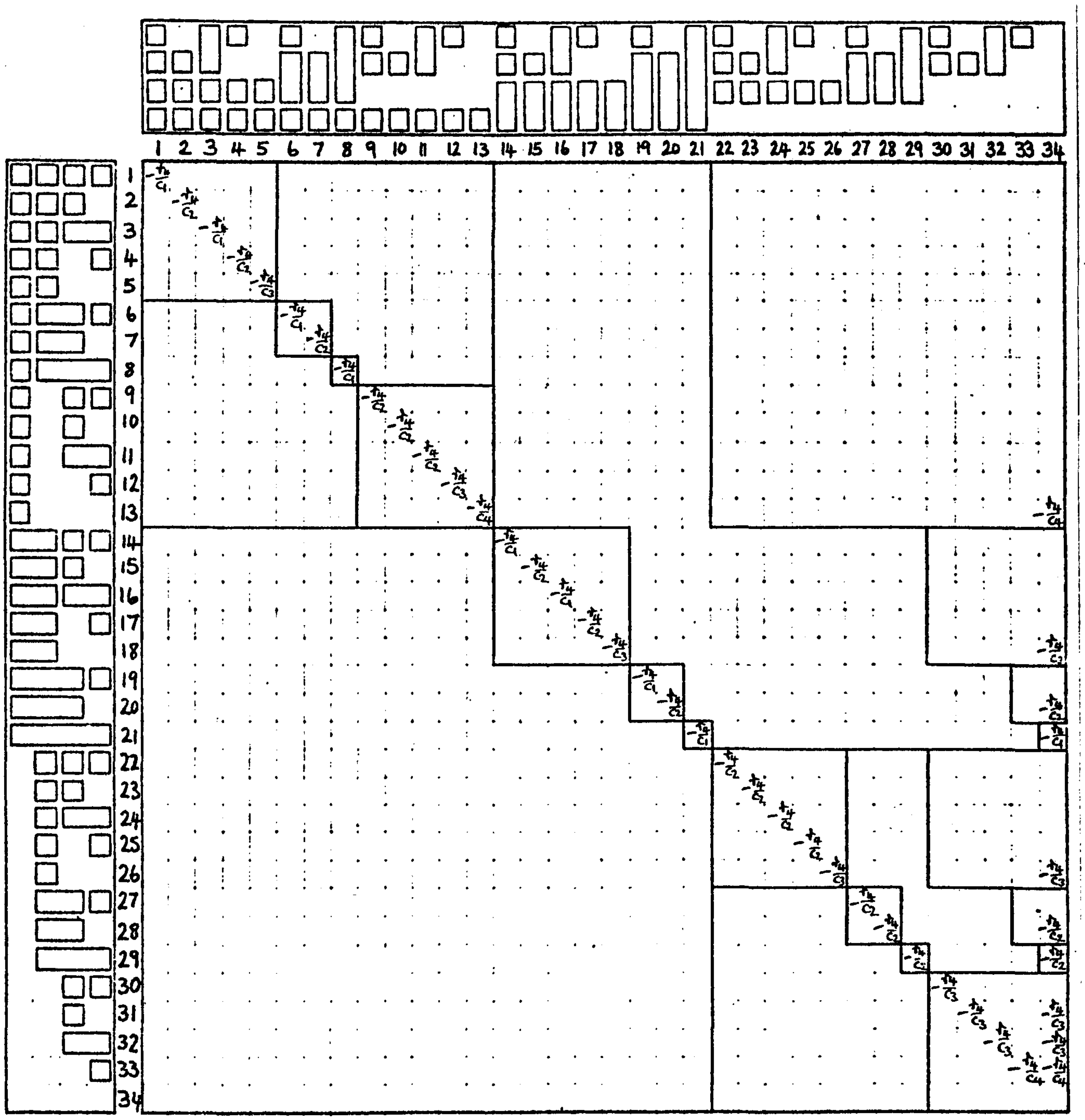


Figure 5.24 $K_4 = r_4K-Q_4D$, for N=4

See equation 5.19. Also see figures 5.19, 5.17, 5.1 for K, Q_4 and D. Cancellation is complete except for the last column and the diagonal.

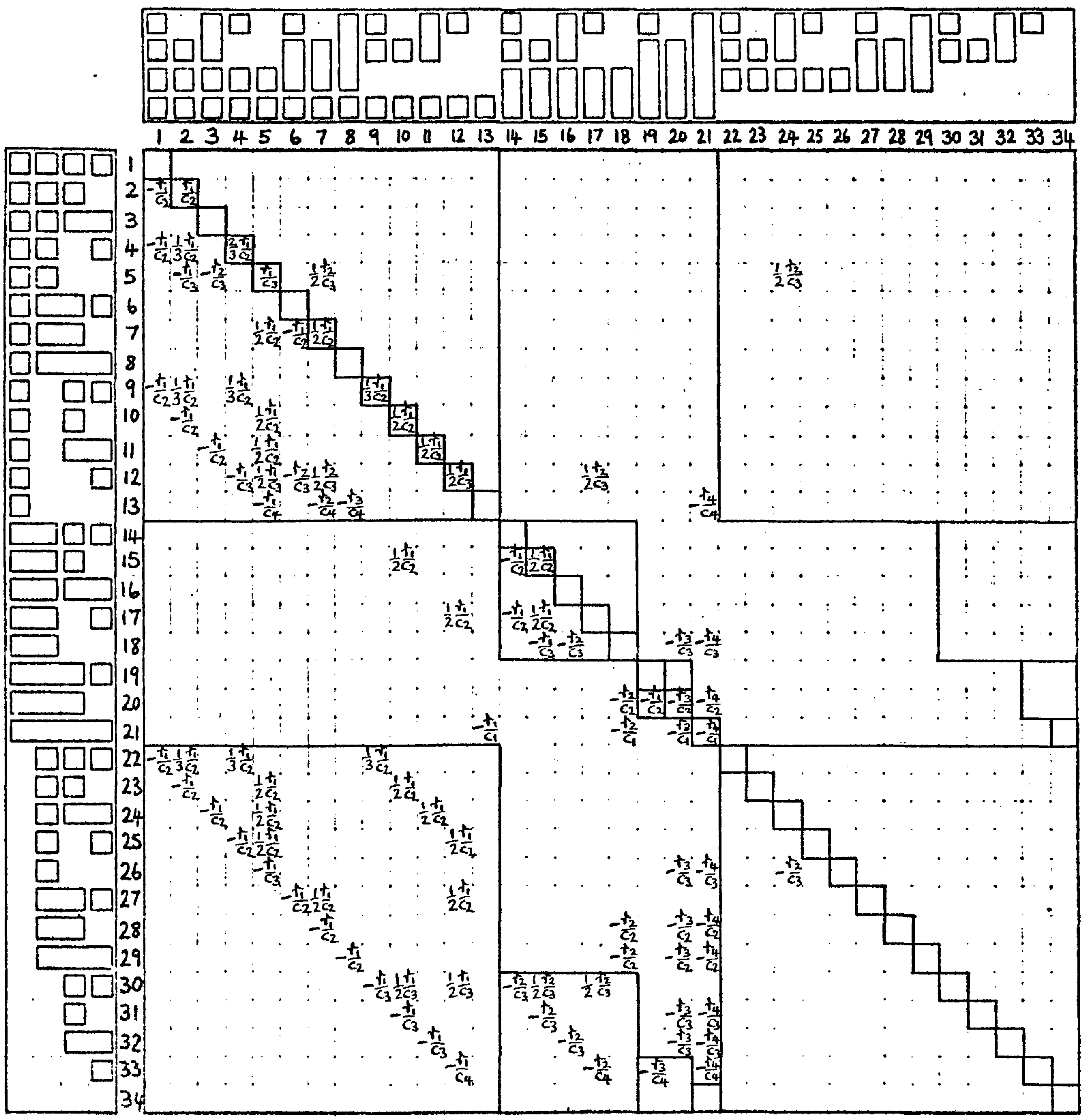


Figure 5.25 $\Sigma K_n A_n$ for first fit allocation algorithm, N=4

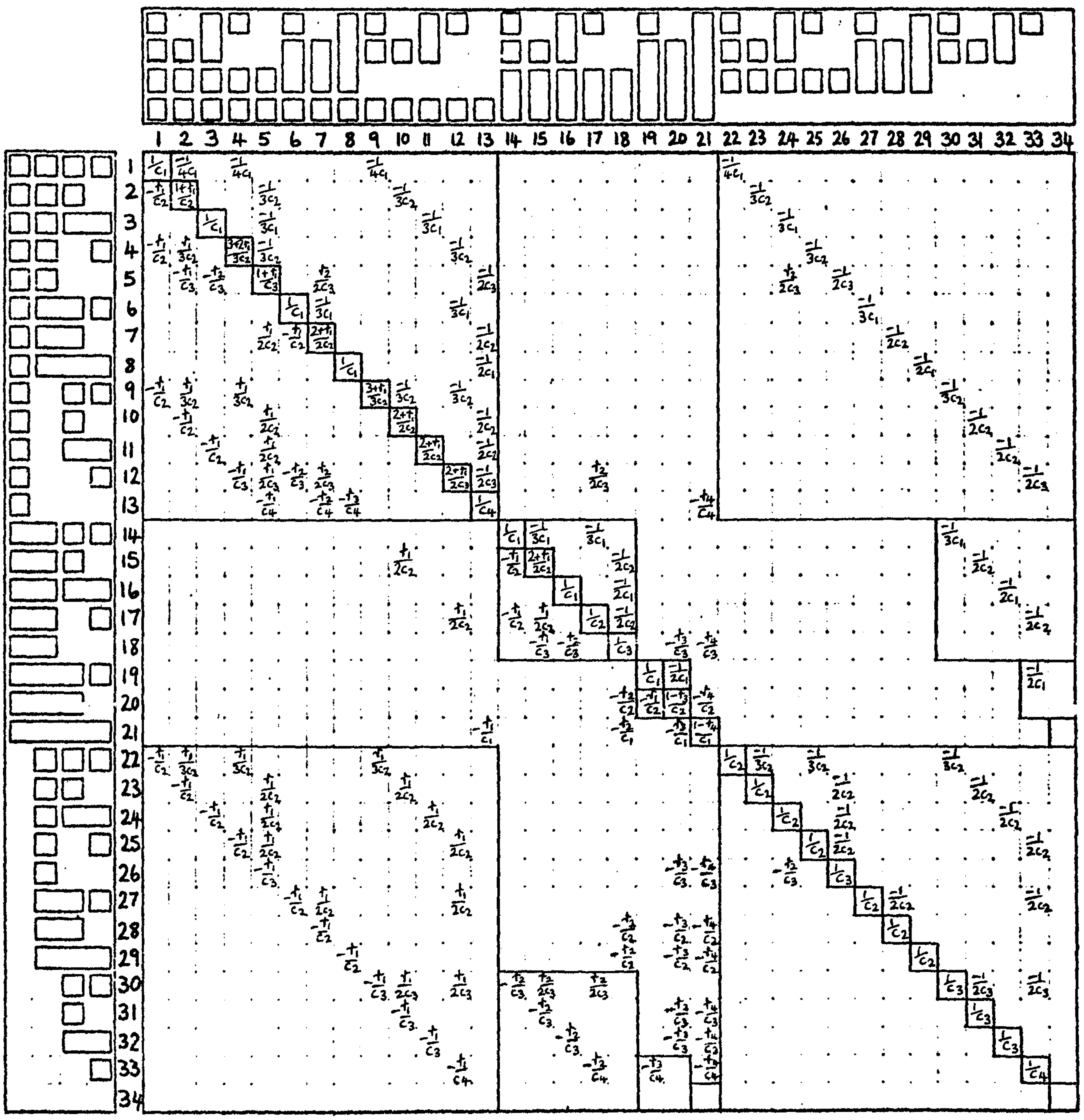


Figure 5.26 $(\Sigma K_n A_n)$-K for first fit, N=4

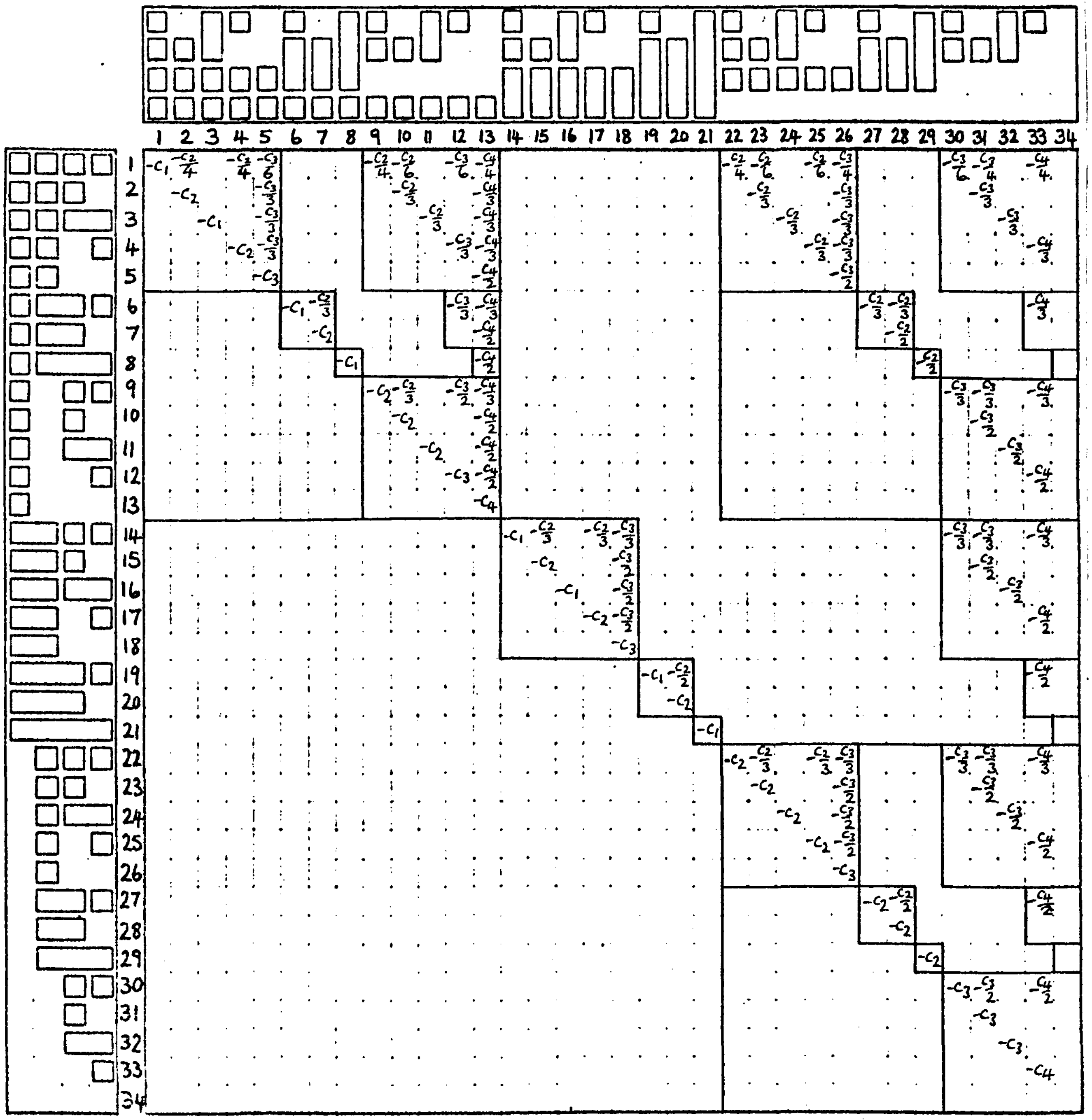


Figure 5.27 K^{-1}, where $K=((\Sigma Q_n DT_n)-I)T'$, see equation 5.18, figure 5.19

than their definition might lead one to believe.

Concerning K and equation 5.21, it is required to show that $(\Sigma Q_n D T_n)T' = T'D$ except for the last column. Both sides of this equation (5.21, with K replaced by its definition from 5.18) are square matrices of size $S = S(N)$. See equation 4.4 and Table 4.1 for the definition and some early values of the total number of states S, and refer to figures 5.1, 5.17 to 5.20 throughout this section. Choose any states with indices i,j such that $1 \leq i,j < S(N)$. Then the i,j-th element of $(\Sigma Q_n D T_n)T'$ is

$$\left(\sum_{n=1}^{N} q_n^{(i)} \, d_{ij} t_n^{(j)} \right) \frac{1}{t_j} = \frac{d_{ij}}{t_j} \sum_{n=1}^{N} q_n^{(i)} \, t_n^{(j)}$$

since $Q_n = (q_n^{(i)})$, $T_n = (t_n^{(i)})$ and $T' = (1/t_i)$ are diagonal by equations 5.11, 5.15 and 5.16.

If $i \geq j$, then $d_{ij} = 0$ since D is strictly upper triangular from section 5.1.1, and clearly the equality with T'D holds. Consider the remaining triangle $1 \leq i < j < S$. For $d_{ij} = 0$ in this triangle there is again equality with T'D, so restrict attention to $d_{ij} > 0$. Let g_i, g_j be the sizes of the largest gaps in states i, j respectively. Since i, j < S then g_i, $g_j < N$. From the definition by equation 5.1 of D, if $d_{ij} > 0$ then $g_i \leq g_j$, for a deallocation can not cause the maximum gap size to decrease.

From the definition by equations 5.11, 3.8 of $q_n^{(i)}$:

$$\begin{aligned} q_n^{(i)} &= 0 && \text{if } n \leq g_i \\ &= r_n/c_{g_i+1} && \text{if } n > g_i \end{aligned} \qquad \text{.... 3.8}$$

and the definition by equation 5.15 of $t_n^{(i)}$:

$$\begin{aligned} t_n^{(j)} &= 0 && \text{if } n \leq g_j \\ &= 1 && \text{if } n > g_j \end{aligned} \qquad \text{.... 5.15}$$

For such $d_{ij} > 0$ with $g_i \leq g_j$ it follows that

$$\frac{d_{ij}}{t_j} \sum_{n=1}^{N} q_n^{(i)} t_n^{(j)} = \frac{d_{ij}}{t_j} \sum_{n>g_j}^{N} \frac{r_n}{c_{g_i+1}}$$

But $t_j = \sum_{n=1}^{N} r_n t_n^{(j)}$ (from the definition by equation 5.9)

$= c_{g_j+1}$ (by equations 3.9 and 5.15)

and so the i,j-th element with $d_{ij} > 0$ becomes

$$\frac{d_{ij}}{c_{g_j+1}} \times \frac{1}{c_{g_i+1}} \sum_{n>g_j}^{N} r_n = \frac{d_{ij}}{c_{g_i+1}} = \frac{d_{ij}}{t_i}$$

which is the i,j-th element of the matrix T'D, which completes the proof.

The last column of $(\Sigma Q_n D T_n)$ is zero as every T_n has a zero last column. But T'D has a non-zero last column, so there is a definite inequality here.

5.2.5 The simplification of the derived constant coefficient matrices K_n

Refer again to figures 5.1, 5.17, 5.19 to 5.24 for examples of the matrices in this section. By the definition in equation 5.18,

$K = [(\Sigma Q_n D T_n) - I]T'$

$= T'(D-I)$ except for a zero last column, by section 5.2.4.

Therefore, continuing the notation of this last section, the i,j-th element of $r_n K$

$= \frac{r_n}{t_i} d_{ij}$ if $i < j < S$

$= - \frac{r_n}{t_i}$ if $i = j < S$

$= 0$ otherwise.

Similarly, the i,j-th element of Q_nD

$= \frac{r_n}{t_i} d_{ij}$ if $i < j$ (D is upper triangular), and if $n > g_i$ (definition 3.8 of $q_n^{(i)}$)

$= 0$ otherwise.

Combining these, the i,j-th element of $K_n = r_nK - Q_nD$ (definition 5.19) is as follows:

$i = j < S$, diagonal :	$-\frac{r_n}{t_i} = -\frac{r_n}{c_{g_i+1}}$	(r_nK only)
$i < j < S$ and $n > g_i$:	0	(cancellation occurs)
$i < j < S$ and $n \leq g_i$:	$\frac{r_n}{t_i} d_{ij} = \frac{r_n}{c_{g_i+1}} d_{ij}$	(r_nK only)
$i < j = S$:	$-\frac{r_n}{t_i} d_{ij} = -\frac{r_n}{c_{g_i+1}} d_{ij}$	if $n > g_i$ (Q_nD only)
$i = j = S$, bottom corner :	0	(both zero)

The bigger n is, the more often $n > g_i$ and the more that cancellation occurs. So K_1 has relatively quite a lot of elements, while K_{N-1}, K_N have very few.

5.2.6 Summary of section 5.2: Algebraic simplification

In this section the algebraic expansion (equations 5.12, 5.13) developed in section 5.1 of the transition probability matrix P was applied to the equation of steady state 3.11: $\underline{\pi} = \underline{\pi}P$, to show that the eigenvector of steady state probabilities must also satisfy equation 5.20: $\underline{\pi}(\Sigma K_nA_n) = \underline{\pi}K$. This result holds for any non-zero request distribution ($r_n > 0$ all $n=1,\ldots,N$) and for any allocation algorithm of the class described in section 5.1.2. For reasons of continuity as well as intuition one may expect that it will continue to hold when some of the (r_n) are allowed to take zero values, although the treatment of the pseudo inverse matrix T' of T will need careful consideration.

As explained in section 5.1.2, any allocation algorithm is realised algebraically as a set of basic allocation transition matrices A_n, and conversely these matrices determine the algorithm. Equation 5.20 can therefore be seen as a very promising development, especially for questions such as studying the effect of the choice of allocation algorithm on the steady state behaviour. From being buried inside the transition matrix P, expressed either implicitly as in $\underline{\pi} = \underline{\pi}P$ or explicitly but apparently intractably in the expansion of that equation as the A_n matrices in equations 5.9 (definition of $A = \Sigma r_n A_n$), 5.12, 5.13, the allocation algorithm now appears in equation 5.20 in a most simple way.

1) Each matrix A_n, $n = 1,\ldots,N$, appears exactly once.

2) The K_n and K matrices, each of which also appear just once, turn out unexpectedly to be quite simple, certainly much simpler than their original definitions would indicate.

3) The K_n and K matrices are variable only in depending on the request distribution (r_n), and so may be considered constant as far as varying the allocation algorithm is concerned.

4) The A_n matrices have no dependence on the request distribution.

5) There is therefore a complete separation of the influence of allocation algorithm and request distribution into the A_n, and K_n and K, matrices in equation 5.20.

6) The whole are combined as a simple linear combination.

It is not easy to imagine how the influence of the allocation algorithm on the steady state could have turned out to have been algebraically expressed any more simply than the actual reduction which has been found in equation 5.20.

5.3 Making use of the algebra : first steps

A first possibility in studying the effect on the model of choosing different basic allocation transition probability matrices A_n would be to determine the limits within which the effects may vary. It surprisingly turns out that the equations governing or constraining the model's stochastic behaviour, independently of whatever allocation algorithm may be being used, can be derived very satisfactorily from the simplified steady state equations 5.20 by combining the various choices available in the A_n matrices so as to ignore these choices, and most of this section is devoted to explaining how and why this can be done. The result is a smaller set of lumped states and corresponding equations 5.23 with an interesting pattern, or structure. The behaviour which they determine is common to all allocation algorithms, which cannot therefore be compared by means of these equations alone.

By contrast, a second idea which if it is successful will allow the performance of different allocation algorithms to be compared, is to treat the model as a Markov decision process and apply the dynamic programming techniques of Bellman (1957) and Howard (1960). This might perhaps even lead to a way of determining the allocation algorithm that has the best possible performance in a given model, or it might allow the dependence of a given algorithm's performance on the request distribution to be studied.

To see how this might be done, consider again the basic allocation matrices A_1, A_2, A_3, A_4 shown in figures 5.9 to 5.12 for N=4. Most of the elements of these matrices are fixed and must be zero (all the blank spaces in fact). Many of the comparatively few remaining elements must take the value 1, indicating that there is no possible choice of how to allocate a given sized block into a particular configuration of memory. However as pointed out in section 5.1.2, where choices are possible they can be represented as variables, for example h and h' in figure 5.11. Since h + h' = 1, there is just one degree of freedom for this pair, although more are possible in the cases for which there is a choice between three or more positions where a block in memory could be placed. In terms of the dynamic programming approach as developed by Howard, the choice of a particular allocation algorithm is equivalent to a choice of policy which then determines the transition matrix (the request size distribution being regarded as given), and each variable element of an allocation matrix such as h becomes a policy variable. The policy space is then the rather large set of all possible allocation algorithms, or choices of values for all the variables such as h in the allocation matrices. The payoff or expected return for a given policy can be measured as usual by the expected equilibrium utilisation (since this is a linear function of the steady state probabilities) obtained with the allocation algorithm which is represented by the policy. Howard's technique for finding the optimal policy consists of iterating round an alternating sequence of "value determination" (finding the expected return for a given policy) and "policy improvement" operations (using the values obtained with this policy to find a better one) until the choice of policy converges, which it will do if the expected return is a linear function of the state and transition probabilities, and the Markov chain is ergodic.

In principle the sheer size of the policy space, the number of different possible allocation algorithms (represented by all the possible different sets of values that the variable elements in the allocation matrices might take) appears to make this idea impractical before it even gets started, and so of course it would be in general. However the hope behind the idea of using this technique is that it might be possible to take advantage of the strong recursive structure which has been displayed in the components of the transition matrix, to see how the (value determination - policy improvement) iteration will work in the light of the knowledge of this structure. The hope is an optimistic one perhaps but a closer study, which has not been pursued further in the present work, may enable some kind of comparison between algorithms to be made. For example, it should indicate that in the given model with N=4 the values of h and h' in figure 5.11 are immaterial since they represent the choice of where to place a 3-word block in the empty 4-word memory. For k,k' and j,j' in figure 5.10 however, it seems clear that it is better to place the 2-word block at one end rather than in the middle of the memory, so that the prediction should be k' = j = 1, k = j' = 0. It will be interesting to see if these kinds of statements, and more, can be predicted by using dynamic programming techniques.

The rest of this section now returns to explaining the first possible development mentioned above. This is the natural grouping of the states which has been discovered from the simplified steady state equations 5.20, and the effect of this grouping on these equations.

5.3.1 Grouping states by columns and by rows

In constructing a set of allocation matrices A_n, there are generally many (increasingly many with N) states or memory configurations for which there is a choice of where a new request of a given size n should be placed, that is, there is an increasing choice of places in which non-zero probabilities may occur, and therefore of what their values should be; see figures 5.7, 5.9 - 5.12. These choices indicate that two separate grouping operations can be performed, first on the columns of equations 5.20, and then on the rows. These operations lead as a result to equations in which new steady state probability variables appear. They hold true regardless of the choice of allocation algorithm.

Column grouping. The individual (column) equations of 5.20, are grouped together if their corresponding "to" states contain the same number and sizes of allocated blocks, without regard to their allocated positions. Figure 5.28 shows how the matrix $(\Sigma K_n A_n)-K$ of these equations is grouped for the example case N=4.

This guarantees that if any two "to" states are possible alternatives in the allocation of a request into any of the "from" states, that is, if any row of the A_n matrices could contain positive probabilities in the columns corresponding to these "to" states, then these columns of equations 5.20 will be grouped together. In terms of figures 5.9 - 5.12 this grouping brings together again all the terms containing factors which are allocation alternatives, such as terms containing z and z' ; w, w' and w" ; etc. All the column equations in each group are added together and a new column equation results. This grouping replaces the uncertainty of not knowing which of a group of possible "to" states any allocation transition may lead to, by the certainty that it must go to one of them in the group. Again in terms of the diagrams, when the addition is performed the variables z and z' are replaced by their constant sum, $z+z' = 1$, and similarly $w+w'+w'' = 1$, etc. The groups of "to" states, which appear along the top of figures 5.28, 5.29, 5.30 for the N=4 case, can be used to label the new column equations which result from this addition.

Row grouping. Following the column grouping, the "from" states, each of which corresponds to a steady state probability variable π_i or row shown on the left hand side of figure 5.28, also fall naturally into groups, see figure 5.29. States, or rows of equations 5.20, are grouped together if they contain the same numbers and sizes of allocated blocks and also if they have the same maximum gap size, without any other regard to the allocated blocks' positions.

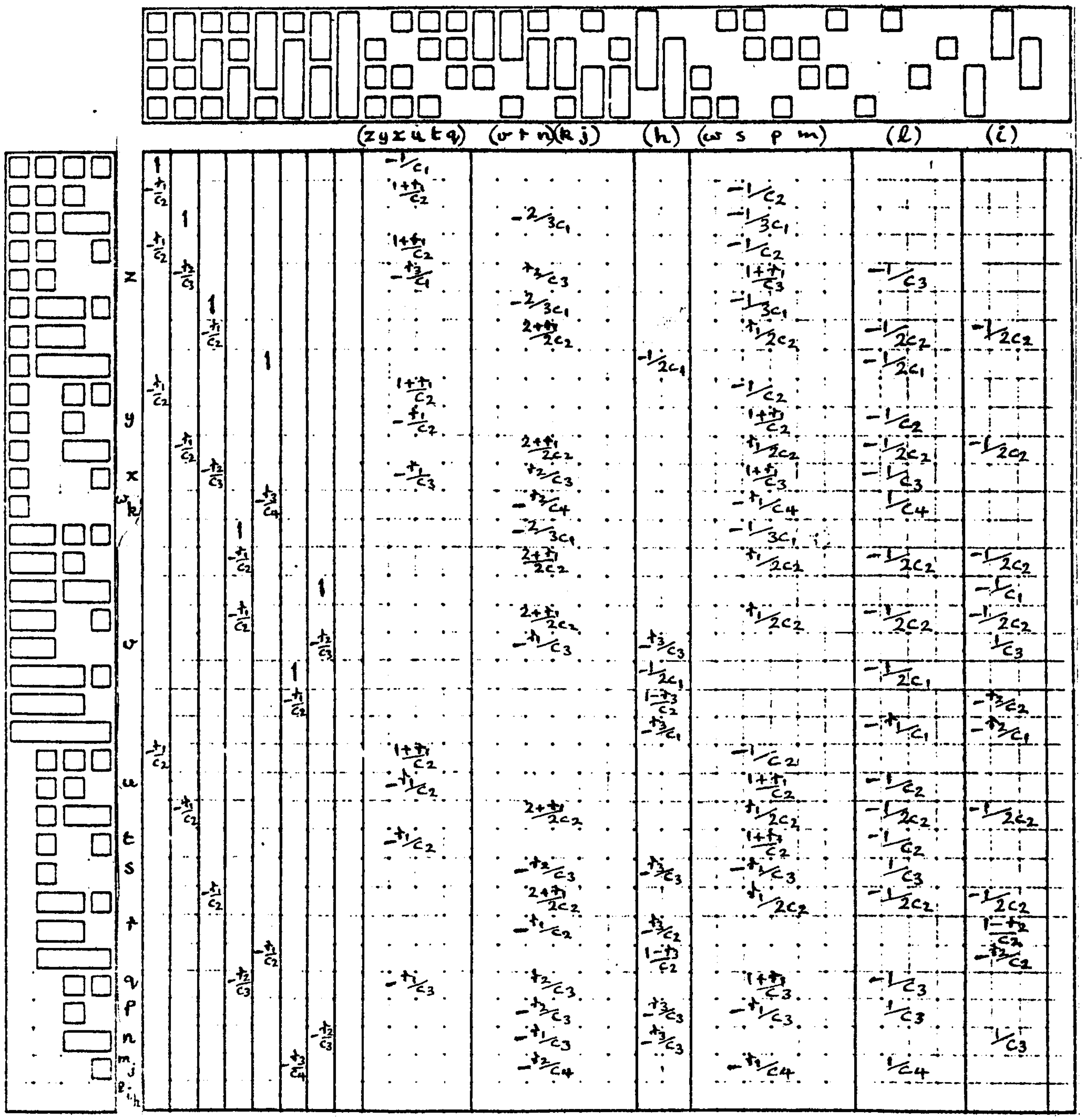


Figure 5.28 Grouping columns of matrix $(\Sigma K_n A_n)$-K of column equations 5.20, N=4

The labels z,y,x,...,h attached to the columns and rows have been taken from figures 5.9 to 5.12 to indicate how and where the alternative choices available to any possible allocation algorithm have been recombined to form the columns shown here.

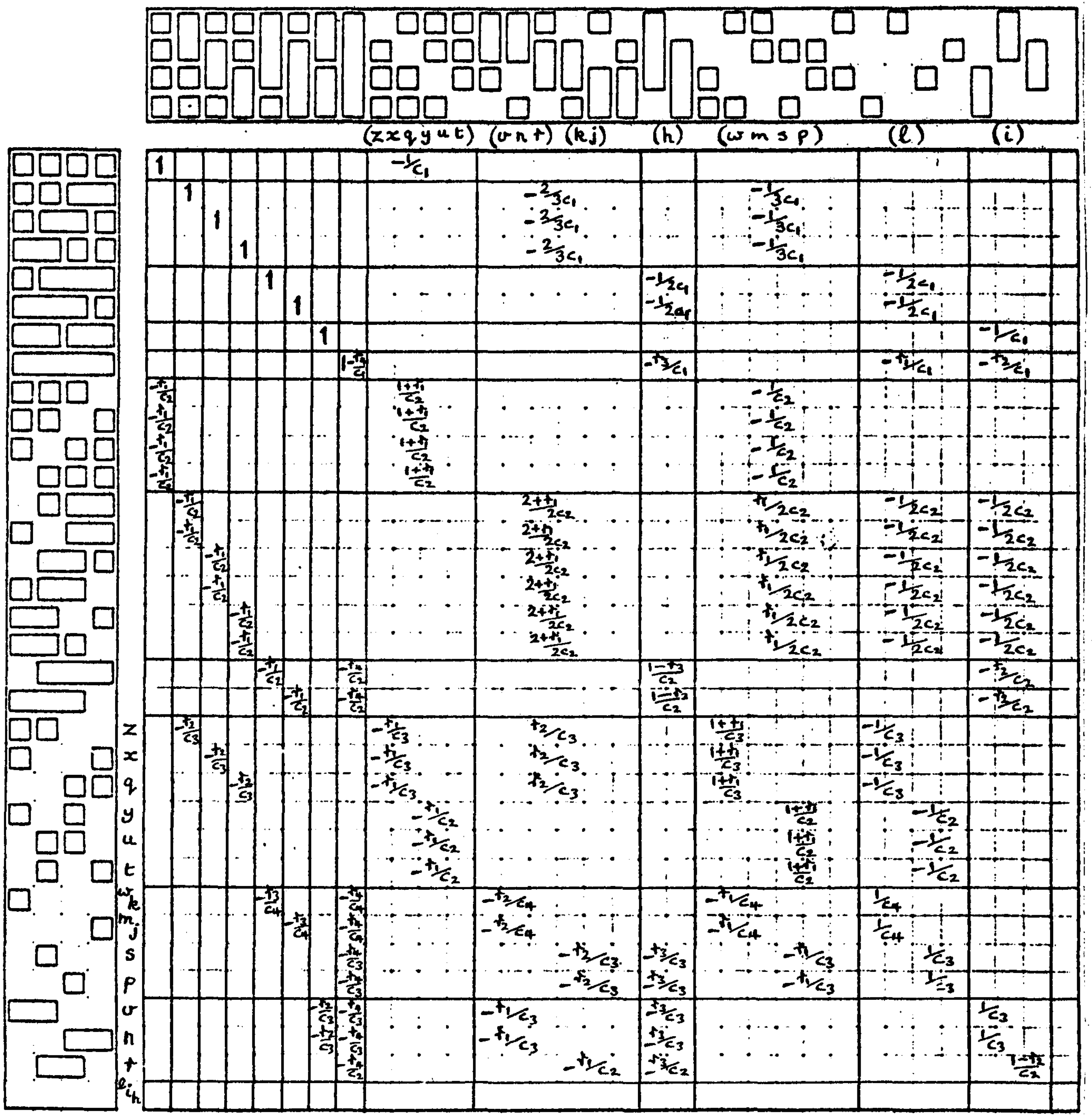


Figure 5.29 Row grouping of equations 5.20 following the column grouping, N=4

This figure is just a rearrangement of the rows of figure 5.28.

All the states i in such a row group k are then found to have exactly the same coefficients in the modified set of column equations resulting from the column grouping. By replacing the individual steady state probability variables (π_i) in each group k by their sum π_k^*, i.e.

$$\pi_k^* = \sum_{\substack{i \text{ is in} \\ \text{group } k}} \pi_i \qquad \text{.... 5.22}$$

these coefficients can therefore be retained, one for each group, and equations 5.20 are condensed:

$$\underline{\pi}^* \sum_{n=1}^{N} K_n^* \; A_n^* = \underline{\pi}^* \, K^* \qquad \text{.... 5.23}$$

The number of "unknown" variables (π_i) is significantly reduced to the number of groups (π_k^*).

Equations 5.20 are thus reduced to a grouped set of equations in fewer grouped unknowns, as shown in figure 5.30. The two groupings are similar but not quite the same, in fact one is a subset of the other. The matrix of equations 5.20 is square so that there are an equal number S'(N) of equations as there are unknowns, the steady state probabilities (π_i), but the columns or "to" states combine into fewer but larger groups than the "from" states or rows, so that the result is less new equations than new unknowns. Their solution therefore contains some degrees of freedom which must be wholly or partly the variation which is dependent on the choice of allocation algorithm. This introduces the possibility of treating the optimisation of the expected storage utilisation U for example as a linear programming problem in the variables (π_k^*). These variables have to satisfy the constraints 5.23 and the two additional requirements that they remain probabilities, that is:

$$\Sigma\pi_k^* = 1 \qquad \text{and} \qquad \pi_k^* \geq 0 \, , \quad \text{all } k$$

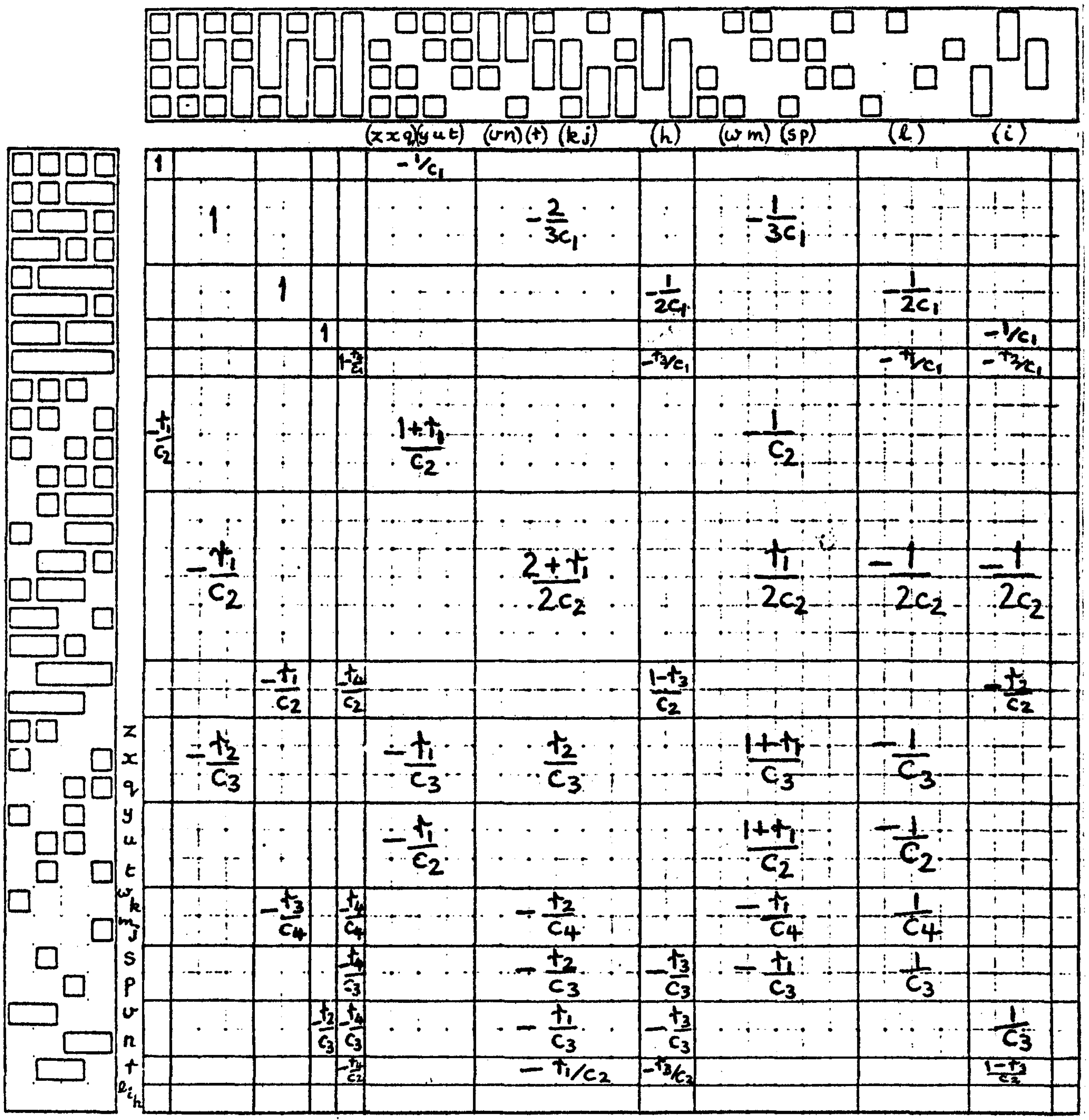


Figure 5.30 Matrix $(\Sigma K_n^* A_n^*)-K^*$ of the condensed column equations 5.23, N=4

but they can otherwise be varied. As they do so, any quantity such as the average fraction U of memory allocated to blocks which is a linear combination of the π^*_k will also vary and can be optimised by the standard techniques of linear programming. From equation 3.3 :

$$U = \underline{\pi}.\underline{u}' = \underline{\pi}^*.\underline{u}^{*\prime} \quad ,$$

where $\underline{u}^{*\prime}$ is the transpose of $\underline{u}^* = (u^*_k)$. Each u^*_k is defined as in chapter 3 equation 3.2 in the natural way as that fraction of the memory which has been allocated to blocks in any of the states i in group k.

5.3.2 Explanation and Justification of the state grouping

Combining groups of columns of equations 5.20 by adding them together is trivially legitimate since $\underline{\pi}$ is a row vector premultiplying the K and A matrices. It merely corresponds to replacing linear equations by their sum. To be able to reduce the number of rows, it has to be shown that in each resulting condensed equation the coefficients of all the states i in a row group k are the same, as was claimed in the last section, so that the replacements indicated by equations 5.22 can be made. This is the purpose of the present section, which proceeds by examining the individual basic matrix components which contribute to equations 5.20.

The n-th component of the LHS of equations 5.20 can be expanded:

$$\begin{aligned} K_nA_n &= (r_nK-Q_nD)A_n \\ &= (r_nT'(D'-I)-Q_nD)A_n \\ &= r_nT'D'A_n - r_nT'A_n - Q_nDA_n \end{aligned}$$

where D' = D except for the last column which is zero, by section 5.2.4.

Using the expansion for K :

$$K = T'(D'-I) = T'D'-T' \qquad \text{.... 5.21}$$

established in that section, equations 5.20 can be rewritten

$$\underline{\pi}(\Sigma r_n T'D'A_n - \Sigma r_n T'A_n - \Sigma Q_n DA_n - T'D' + T') = \underline{0} \qquad \text{.... 5.24}$$

Each of these components makes a contribution to the matrix of the uncondensed equations 5.20, depending on the values of the maximum gap size g_i and the number of allocated blocks m_i in each of the "from" states $i = 1,\dots,S$, as follows.

1) $r_n T'D'A_n$. For "from" states i with one block, $m_i=1$, the contribution to $\underline{\pi}(\Sigma r_n T'D'A_n)$ is zero as D' has a zero right hand column. For states i with more than one block, $m_i > 1$, the cases with $n > g_i$ cancel with corresponding cases from $-Q_n DA_n$, 3) below, leaving contributions with values

$$\frac{r_n}{c_{g_i+1}} \cdot \frac{1}{m_i} \times \text{(allocation probability)}$$

from all the cases $n = 1,\dots,g_i$. The basic transitions represented are formed by deallocating one block from each "from" state i with more than one block, in all possible ways, and then allocating a new block of size n up to g_i words, with corresponding probabilities.

2) $-r_n T'A_n$. The contribution is for all states i with $g_i > 0$ and consists of all possible ways of allocating a new block of size n not greater than the maximum gap size, that is, $n = 1,\dots,g_i$. The value of each contribution is

$$-\frac{r_n}{c_{g_i+1}} \times \text{(allocation probability)} .$$

There can be no contribution from states i for values of $n > g_i$ by the definition of the A_n matrices, which have zero rows whenever the block to be allocated exceeds the maximum size gap for the configuration of memory corresponding to the row.

3) $-Q_nDA_n$. For all states i with exactly one allocated block, $m_i = 1$, the contributions have values

$$-\frac{r_n}{c_{g_i+1}} \times \text{(allocation probability)}$$

and for each state i they represent all possible ways of allocating a new block of any size n greater than the original maximum gap size, $n > g_i$, into the empty state. The contribution in the cases $m_i = 1$, $n \leq g_i$ is zero by definition of Q_n, as it is for $m_i > 1$, $n \leq g_i$. In the remaining cases $m_i > 1$, $n > g_i$, it happens that $Q_n = r_nT'$ and $D = D'$ in the i-th row, so that as promised the contribution from $-Q_nDA_n$ is cancelled by the corresponding cases from $r_nT'D'A_n$, 1) above.

4) $-T'D'$. Since D' has a zero right hand column, the non-zero contribution here is for all states i with more than one allocated block, $m_i > 1$. For each such state i, all possible deallocations of one block are included, with contribution values

$$-\frac{1}{c_{g_i+1}} \cdot \frac{1}{m_i}$$

5) T' has a contribution to all states i, with values

$$\frac{1}{c_{g_i+1}}$$

on the main diagonal in the matrix diagram.

From this list of contributions it can be seen that, as claimed above, for any state i of any given row group k the sum of the contributions occurring within a given column group is constant. Any variation in the allocation matrices A_n can only redistribute the contribution within the columns of a column group and can not cause variation between column groups, as the resulting total number and sizes of blocks must stay the same. If an allocation is possible into a column group for one state i of a row group k, then it must be equally possible for all the other states of k. In cases 1) and 3) above it does not matter that the resulting maximum gap size may possibly increase from g_i as a result of a deallocation, and moreover possibly increase differently for different i in a row group k, because of the very fortunate cancellation of the allocation contributions from $r_n T'D'A_n$ and $-Q_n DA_n$ in the cases $n > g_i$. The contributions from the other components D', T', Q_n depend in value only on g_i and m_i and are equally confined to column groups.

For example, consider the contributions to equations 5.20 from the first component, $r_n T'D'A_n$, which occur for those states i with more than one block, $m_i > 1$, and (for a given n) with maximum gap size $g_i \geq n$. Consider the factors of this component in turn. r_n is a constant irrespective of choice of state. T' is diagonal with elements $1/c_{g_i+1}$ which will remain constant for a given row group, since g_i is constant for all states i in the group. The non-zero values in D' depend only on m_i, in fact they are $1/m_i$, and so will be constant for a row group, further, it is easy to see that if one state in a row group has a deallocation transition (non-zero element of D') to just one (two, three, ...) state(s) in a group of columns, then all the other states in the same row group must also have just one (two, three, ...) such transitions to some state (not necessarily the same one of course) in the same group of columns. Finally, a block of size n can be allocated into any of the resulting states since $n \leq g_i$ and the maximum gap size can not decrease on a deallocation. Whatever allocation choice is made (including possibly mixtures of choices with varying probabilities) for any of the states in the row group, the addition of all the column equations in a group reduces the choice to the certainty of knowing the resulting set of sizes of allocated blocks in memory, that is, the allocation must take place and the result is confined to the states labelling the columns of the group.

The following rules for writing down the coalesced set of equations 5.23, or filling in the condensed matrix (figure 5.30 for N=4), are derived by collecting together the contributions from 1) - 5) above.

1) Row groups k with more than one allocated block.

a) -T'D' : deallocate one block all possible ways,

$$\text{coefficient} = -(\text{deallocation fraction})^{\text{N.B.}} \times \frac{1}{c_{g_k+1}}$$

b) $r_n T'D'A_n$: deallocate one block all possible ways, then allocate one block for all possible sizes n up to and including the original maximum gap size g_k.

$$\text{coefficient} = -(\text{deallocation fraction}) \times \frac{r_n}{c_{g_k+1}}$$

2) Row groups k with exactly one allocated block.

$-Q_n DA_n$: allocate one block for all possible sizes n greater than the maximum gap size g_k, into the empty state.

$$\text{coefficient} = -\frac{r_n}{c_{g_k+1}}$$

3) All row groups.

a) $-r_n T'A_n$: allocate one block for all possible sizes n up to and

N.B. The term "deallocation fraction" means, for each column group, that proportion of the total ways of deallocating one block which leads to that group. For example, if a row group contains three 2-word blocks, and two 1-word blocks, then there are two possibilities, either deallocate a 2-word block (three ways), deallocation fraction = 3/5 , or deallocate a 1-word block (two ways), deallocation fraction = 2/5 .

including the maximum gap size g_k.

$$\text{coefficient} = -\frac{r_n}{c_{g_k+1}}$$

b) T' : enter a value on the "diagonal", i.e. in the column group entry which matches (contains) the row group.

$$\text{coefficient} = \frac{1}{c_{g_k+1}}$$

Each coefficient contains a factor $1/c_{g_k+1}$, which is constant along a given row group k. As they stand, equations 5.23 are homogeneous linear equations in the unknowns π_k^* , with the added constraint that $\Sigma\pi_k^* = 1$. It is therefore possible, if convenient and desired, to absorb this constant factor by redefining the "variables" so that the equations 5.23 govern the "unknowns" π_k^* /c_{g_k+1}, and the matrix is simpler, as appears in figure 5.31 for N = 4. The new unknowns no longer add to 1, but the previous constraint $\Sigma\pi_k^* = 1$ is unchanged of course.

5.3.3 Summary of section 5.3: Allocation independent equations

Following the reduction of the steady state equation to equation 5.20, a study with the aid of figures 5.9 to 5.12, the basic allocation matrices for the general allocation algorithm in the N=4 case, revealed that there is a natural way of grouping together the states corresponding to the individual column equations of 5.20. This is by reference to the choices possible when constructing the allocation transition matrices A_n, that is, the choice of allocation algorithm. States which are possible alternatives as the result of any particular allocation in any configuration, are put into the same group. This grouping nullified the allocation choice in the sense that in any individual case, whatever alternative resulting state is actually chosen, the

1					-1					
	1					$-\frac{2}{3}$		$-\frac{1}{3}$		
		1					$-\frac{1}{2}$		$-\frac{1}{2}$	
			1							-1
				$1-t_4$			$-t_3$		$-t_1$	$-t_2$
$-t_1$					$1+t_1$			-1		
	$-t_1$					$1+\frac{t_1}{2}$		$\frac{t_1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$
		$-t_1$		$-t_4$			$1-t_3$			$-t_2$
	$-t_2$				$-t_1$	t_2		$1+t_1$	-1	
					$-t_1$			$1+t_1$	-1	
		$-t_3$		$-t_4$		$-t_2$		$-t_1$	1	
				$-t_4$		$-t_2$	$-t_3$	$-t_1$	1	
			$-t_2$	$-t_4$		$-t_1$	$-t_3$			1
				$-t_4$		$-t_1$	$-t_3$			$1-t_2$

Figure 5.31 Simplified matrix of condensed equations 5.23, N=4

The simplification from figure 5.30 is by absorbing the factors $1/c_{g_k+1}$, constant along each row, into the definition of the "variables" π_k^* / c_{g_k+1}.

resulting group must be the same.

Having performed this column grouping, perhaps to some surprise it was then found that the rows of the resulting modified form of equations 5.20 can also be similarly grouped as a consequence. This is because in all the modified equations the coefficients of the states in each row group turn out to be the same, so that the group probability variables (π_K^*) defined by equation 5.22 can be introduced. The rows can be formed into the same groups as the columns but less completely as an extra property ignored by the column grouping, maximum gap size, is preserved.

The details of both sets of groupings are confirmed in section 5.3.2, the reasons for the unexpected row grouping displayed and examined, and the rules for writing down the condensed equations 5.23 directly without reference to equations 5.20 are derived.

These condensed allocation independent equations 5.23 were discovered initially from curiosity aroused by constructing alternative matrices $(\Sigma K_n A_n)$-K to that of the first fit allocation algorithm shown in figure 5.26 for N=4. These reduced equations represent an advance on equations 5.20, which they really are in a composite form. The original number S(N) of states and equations is reduced to the number of grouped state probability variables π_K^*, with a further reduced number of equations. The difference between these, the maximum number of degrees of freedom allowed to the allocation algorithm in equations 5.23, introduces the possibility of doing linear programming on these equations, for example to find the best or worst possible utilisation since this is a linear function of the variables π_K^*. The condensed equations have so far resisted attempts at solving them directly or reducing them further by appealing to the apparently fairly complicated structure of their matrix (figures 5.29 - 5.31).

A related possibility instead of removing the possible allocation choices by grouping the states together is to try using dynamic programming techniques to see if it is possible to compare these choices in terms of some linear performance function, such as the storage utilisation. This was discussed at the beginning of this section although the investigation has not yet been carried very far, and has not been examined further in this thesis.

Chapter 6 : Computing the exact probability behaviour of small memories

One way of investigating the model of storage allocation developed so far is actually to construct the transition probability matrix P and use the power method of equation 3.13 to calculate the convergent sequence of successive vectors $\underline{\pi}(T)$ for T = 1,2,... from a given initial vector of state probabilities $\underline{\pi}(0)$. Although, as will become clear below, the memory sizes for which such an investigation can be attempted are very small, it can provide an indication of performance behaviour in models with larger memories. The choice of allocation algorithm is almost irrelevant for the 4-word memory which appears in most of the example figures (these have been constrained by the practical difficulty of displaying larger matrices in detail, such as the 89 $\times$ 89 matrix needed for N=5, on one page), but there are increasingly many memory configurations where an allocation choice is possible as N increases. It is possible that any differences in utilisation or fragmentation which may be observed, for N=10 for example, will be enough to indicate what may be the differences between alternative allocation algorithms when N is large. This applies equally when comparing the effects of different request distributions, in an effort to find out how the relative performance of different allocation algorithms depends on the choice of distribution.

Such differences would not be apparent at N=4. However, there may equally be effects that are invisible at N=10, say. It must be pointed out that the influences of both the end effects and the lack of choices available in memories as small as this will be very strong when compared with memory sizes which are one or more orders of magnitude greater, and so any inferences about the behaviour in a larger memory must be made with caution. Because of this and of the unlikely possibility of being able to extend these computations very much further with the present implementation, only a representative set of computations have been made to try to find out what comparisons this implementation of the model is able to make.

One significant advantage of the power method is that it gives an indication of how quickly or slowly the successive state probability vectors $\underline{\pi}(T)$ converge to the steady state equilibrium vector $\underline{\pi}$. The convergence of this method depends on the relative magnitude of the dominant eigenvalue (which has value 1) of the transition matrix P to the next largest. If this next largest magnitude eigenvalue is close to 1, then convergence will be slow and it is at least possible that better numerical methods such as simultaneous iteration will speed it up. See Jennings and Stewart (1975), or Stewart (1977, 1978) for a description and comparison of simultaneous iteration and some other methods. However, unlike these other possibly more direct techniques which may proceed more quickly to the solution, the power method reflects the actual behaviour of the model so that if convergence is slow with the power method then the transient behaviour from a given starting state in the model will take a correspondingly long time to die away. In practice the rate of convergence has been found to vary mainly with the request distribution, but never to be so slow that no progress is apparent at each iteration of the computation. In case the rate of convergence did turn out to be unreasonably slow, a simple and cheap numerical technique which ought to be appropriate for the power method, the epsilon algorithm, has been used as the computation proceeds to provide an estimate of the final converged value of the equilibrium storage utilisation U. There is no dependence of the power method computation on the epsilon algorithm.

The practical computation of the transition probability matrix P is worthwhile if it leads to insights and understanding both of its structure and of other aspects of the working of the model. This was in fact the case when an early implementation of the calculations described here led to a careful examination of the method which was used to construct the transition matrix, and so to the realisation of the algebraic structure of P as described in chapter 5. In principle P can be constructed if the memory size N and allocation algorithm are given, for it is then determined. For any starting state (or row of P) all the possible transitions that can be made to other states can be calculated with their respective probabilities, and so each row of P can be filled in accordingly. Alternatively, the basic matrices (Q_n), D, (T_n), (A_n), A, T defined in chapter 5 can first be calculated, so that expression 5.12 or 5.13 can then be used to calculate P in terms of these simpler matrices. This will have exactly the same result as working P out row by row, for as chapter 5 showed, the two procedures are essentially the same. In practice of course the rapidly increasing number $S = S(N)$ of states as the memory size N increases is bound to mean that any such computation is eventually limited either by storage space or by computation time, and by both sooner rather than later. With the computing resources available to the author at the University of Newcastle, the IBM 360/67 and subsequently the 370/168 computers running under the MTS operating system, it has been found possible to compute models of memory size up to N=12 words. At this point the time required to compute models of larger memory sizes was increasing by a factor of over three times as much for each extra word of memory. A model with N = 13 words would have required over 24 hours of computing time, and this was judged to be not worthwhile with the present implementation.

6.1 A straightforward implementation of the power method

This was programmed in Fortran on the IBM 360/67, before the 370/168 computer was available. After reading in the parameters: memory size N and distribution of request size (r_n), the transition matrix P was first constructed and then used to compute successive vectors $\underline{\pi}(T)$, T = 1,2,... starting with an initial probability vector $\underline{\pi}(T=0)$. As each vector $\underline{\pi}(T)$ was found, the scalar product $U(T) = \underline{\pi}(T) \cdot \underline{u}'$ was formed to obtain the expected utilisation U(T).

This method will not be described in detail since a subsequent different implementation described below proved to be more efficient. It is worth mentioning that the representation and accessing of P, $\underline{\pi}(T)$, $\underline{\pi}(T+1)$ and $\underline{u}$ presented a problem since in general each was too large to be stored in main memory to allow random access. To get round this, the row vectors and the matrix were partitioned into equal width subvectors and submatrices respectively, and all were stored sequentially on secondary memory (disc storage). Each submatrix of P was calculated and stored row by row, and only the non-zero values kept so as to take advantage of the sparseness of P. A gap of n intervening zeroes was represented and easily distinguished by storing (-n) in place of a probability. The operation of equation 3.13 was performed once with a single scan of P, $\underline{\pi}(T+1)$ and $\underline{u}$, and several sequential scans of $\underline{\pi}(T)$. Equation 3.13 takes the partitioned form:

$$\underline{\pi}_k(T+1) = \underline{\pi}(T).P_k \quad , \qquad k = 1,2,... \qquad \text{.... 6.1}$$

where the suffix k is used to index the subvectors and submatrices. The largest memory size computed with this method was N=8, which required over an hour of computing time on the 360/67 computer.

6.2 Using the algebraic form of P to try to improve the power method

It turned out that roughly one third or more of the processing time to perform one complete computation for a given size of memory and choice of allocation algorithm and request distribution in this first implementation, was taken up with computing and storing the transition probability matrix P, before it could be used to produce the $\underline{\pi}(T)$ vectors in successive iterations. Moreover, although P is very sparse it does contain a fairly approximate average of 2N non-zero elements per row, all of which had to be stored along with roughly N or more indicators of gaps in each row; compare figures 4.6, 4.7 for example. In contrast to this, in the algebraic expansion 5.12 of P:

$$P = \Sigma Q_n DT_n + (\Sigma Q_n DA_n)(T+A(T+A(\ldots(T+A)\ldots))) \qquad \ldots\ 5.12$$

the basic matrices in this expansion have an average of one non-zero element per row so that they are very sparse, and moreover these are arranged in diagonal patterns according to the recursive structures described in chapter 5. It seemed a possibility therefore that instead of multiplying the row vector $\underline{\pi}(T)$ by a pre-computed and stored P, the vector could be multiplied one step at a time by each of the basic matrices according to the expansion 5.12. The advantage of this would be that none of these basic matrices would need to be kept anywhere in storage, with a consequent saving in both space for storage and time for accessing the array, although some of this saved processing time would have to be used to continually recompute the elements of the matrix.

The space advantage gained by not storing the matrix has indeed allowed larger models to be computed, and it has been accompanied by a drop by a factor of more than two in the processing time needed compared with the simple implementation. This is possible because the second implementation can and does recognise and avoid those parts of the computation which are either all zero or are subsequently unused. Although this justifies the structured approach as being more successful with regard to both space and time, its increased efficiency has not been found sufficient to progress as far as would have been liked along the exponential curve of increasing computation time with increasing memory size. In both implementations of the model, the processing time has been found to increase by over three times for each extra word of memory size so that there is approximately thirty-seven times as much computation in the N=11 model as there is for N=8 with the same implementation. N=12 is the largest model that has been computed with the structured model, 120 times as much computation or over two orders of magnitude larger than the N=8 model. Although the difference between 8 and 12 is not great, each iteration of the N=8,9,10,11,12 models took respectively 23, 78, 262, 850 and 2762 seconds of computation time on the 370/168 computer to complete. For the uniform request distribution, the N=8 model required 10 iterations to achieve convergence to four decimal places of accuracy and was complete in 218 seconds (the first two iterations are shorter due to the efficiency checks detecting and bypassing whole blocks of initially zero probabilities). This compares favourably with the 70 minutes needed for N=8 by the former more simple implementation on the 360/67, even though that computer is about five times slower. However, the N=12 model required 11 iterations and 28160 seconds (7 hours, 49 minutes, 20 seconds) of computation time to complete with the second implementation on the 370/168.

6.2.1 Explanation of the idea by which no matrices need be stored

The idea is that the position of the diagonals in any given basic matrix should dictate which segments of $\underline{\pi}(T+1)$ will be contributed to by which segments of $\underline{\pi}(T)$, and the values of the elements in the diagonals of each basic matrix are sufficiently simple to be easily calculated on the spot by knowing the position of the diagonal and of the element within it. This scheme has been implemented using recursive procedures in Algol W on the 370/168.

As an example to explain how this is intended to work, more or less in the pictorial way that the idea originally occurred, consider the operation displayed in figure 6.1. An arbitrary (row) vector $\underline{x}$ is to be post-multiplied by a basic matrix (D, say) to produce a row vector $\underline{y}$:

$$\underline{x} \cdot D = \underline{y} \qquad \ldots.\ 6.2$$

In this conceptual example $\underline{x}$ would start off at the beginning of the (T+1)-th iteration as $\underline{\pi}(T)$, at the end of the iteration some vector $\underline{y}$ would contain $\underline{\pi}(T+1)$, and during the iteration the matrix would be any of those in equation 5.12. In figure 6.1, the deallocation matrix D has been chosen as the example (compare figure 5.1). The vector $\underline{x}$ is shown turned on its side (transposed) down the left hand side of the diagram, and the vector $\underline{y}$ appears along the top. The multiplication $\underline{x}D$ produces the elements (y_j) of $\underline{y}$:

$$y_j = \sum_{i=1}^{S} x_i d_{ij} \quad , \qquad j = 1,\ldots,S \qquad \ldots.\ 6.3$$

This can be viewed pictorially as producing the elements y_j one at a time, by "marching" the column vector $\underline{x}$ across the matrix D one column at a time; for each column j of D thus marched over, y_j at the top is formed by taking the scalar product of $\underline{x}$ with the j-th column of D. Since D is sparse, this can be

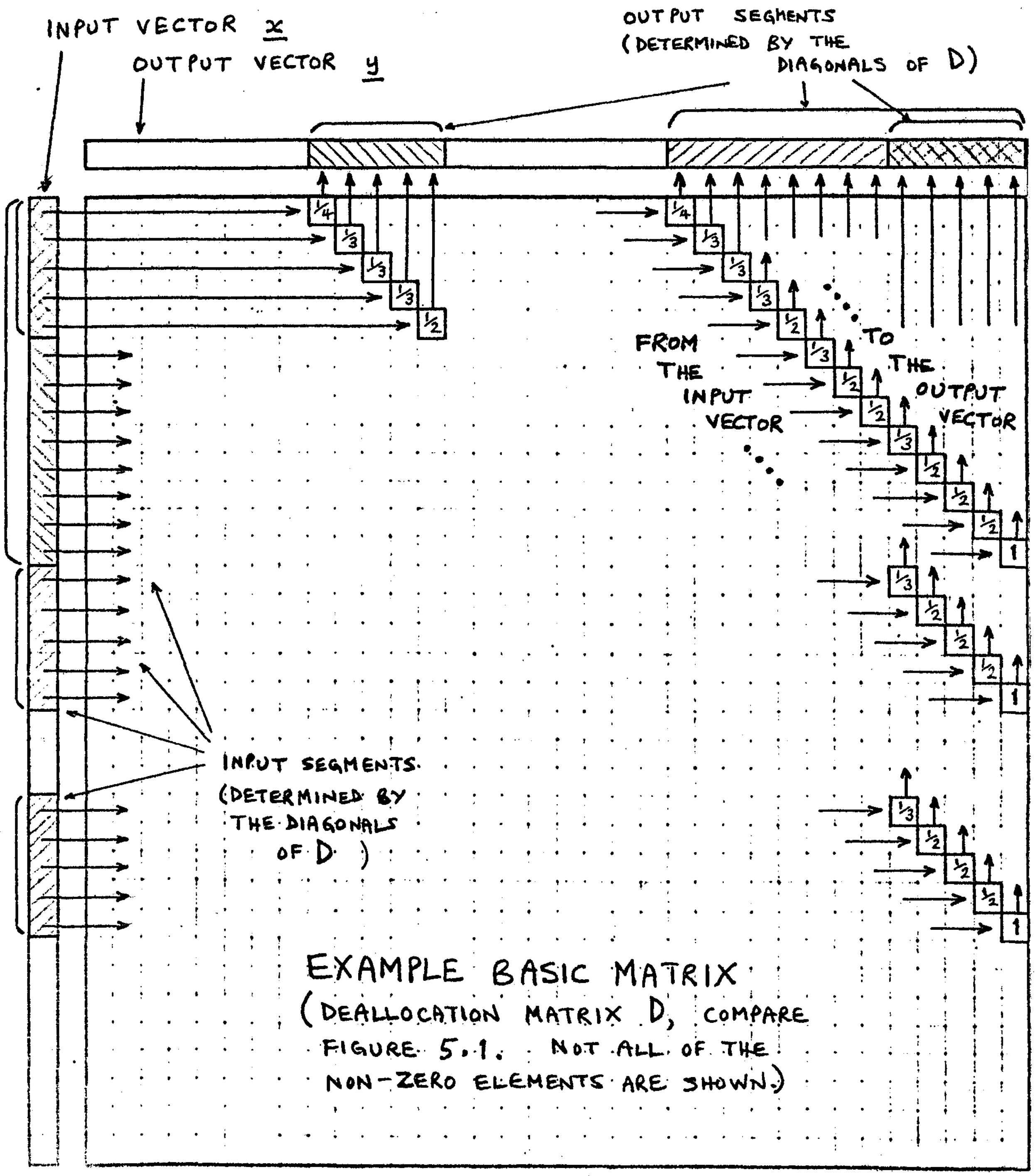

INPUT VECTOR $\underline{x}$
OUTPUT VECTOR $\underline{y}$
OUTPUT SEGMENTS (DETERMINED BY THE DIAGONALS OF D)
FROM THE INPUT VECTOR
TO THE OUTPUT VECTOR
INPUT SEGMENTS (DETERMINED BY THE DIAGONALS OF D)
EXAMPLE BASIC MATRIX
(DEALLOCATION MATRIX D, COMPARE FIGURE 5.1. NOT ALL OF THE NON-ZERO ELEMENTS ARE SHOWN.)
1/4
1/3
1/2
1

Figure 6.1 <u>Operation of equation 6.2 : x . D = y</u>

Contributions to the output vector y and from the input vector x occur in contiguous segments because the elements of D occur in contiguous diagonals.

viewed as accumulating contributions to the appropriate elements of $\underline{y}$ whenever a non-zero element of $\underline{x}$ "hits" a non-zero element of D; the product of each such "collision" gets sent up to be added in to the element of $\underline{y}$ sitting at the top of the column in which the collision occurred.

The point of labouring this piece of elementary matrix algebra now appears. The matrix D consists of a collection of non-zero diagonals (some of which may be only one element long). As $\underline{x}$ "marches" across any particular diagonal of D, successive elements (x_i) of $\underline{x}$ make contributions to successive elements (y_j) of $\underline{y}$, for as long as the diagonal lasts. The diagonal has the effect of choosing a segment of $\underline{x}$ to be contributed to $\underline{y}$. The length of the segment is the length of the diagonal. Its position depends only on the position of the diagonal; if the diagonal is near the top of the matrix then the segment of $\underline{x}$ comes from near the beginning of $\underline{x}$, and if the diagonal is near to the right of the matrix, then the segment contribution to $\underline{y}$ is near to the right hand end of $\underline{y}$. The values of the contributions to each element of $\underline{y}$ depend on the corresponding values of the elements of $\underline{x}$ and of the diagonal of the matrix.

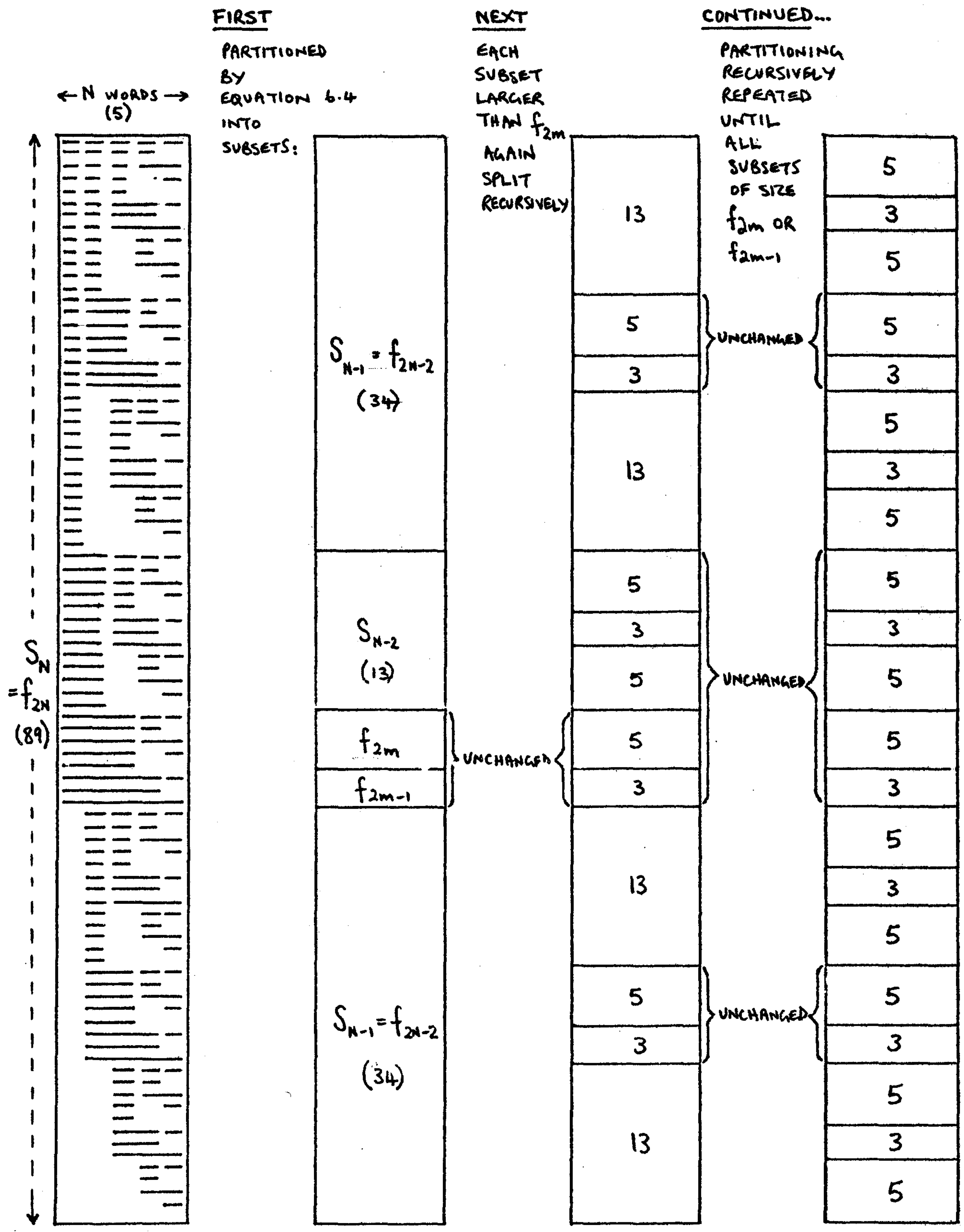
Hence if the matrix D (or any other of the basic matrices in the expansion of P) is viewed without the aid of figure 6.1 just as a collection or list of diagonals, then the multiplication operation can be viewed as a set of selections of segments of the input vector to be contributed to the output vector, the actual positions and values of the segment contributions being determined by the actual diagonals. In terms of a computer implementation, this becomes a set of sequential operations on an input vector and an output vector, the actual values used in each sequential operation (the values of the elements of the diagonal of the matrix) being computed each time they are required instead of being stored.

These sequential operations, resulting directly from the sparse, simple diagonal nature of each of the basic matrices, are the key to the implementation, as they allow the length S(N) of the input and output vectors to greatly exceed the amount that can be stored and randomly accessed in random access memory. The vectors are kept on secondary (disc) storage and the sequential access with relatively few jumps needed, one for the beginning of each new diagonal, makes it possible to use this storage medium reasonably efficiently.

6.2.2 Partitioning the computation by using the recursive algebraic structure

The earlier implementation of the power method described in section 6.1 which started by computing and storing the transition matrix P, took no account of its structure other than its sparseness, either to compute or to store it. The partitioning of the computation into equal sized segments limited in size so that they fitted into main memory was chosen without regard to any other properties of the matrix. In the revised implementation, the recursive structure of the transition matrix P has been used to choose the sizes and positions of the segments, and in fact to some extent this is necessary for the dynamic computation of the elements of the basic matrices.

In this section, the notation S(N) is written as S_N for convenience. The states 1,...,S(N) are partitioned into segments of two different sizes, f_{2M} and f_{2M-1} as follows; see figure 6.2. The maximum number of such segments ever required to be in main memory at once in order to be able to carry out the power method computation turns out (below) to be two. A positive integer M, the "memory size index", is chosen for which it is assumed to be possible that two segments or vectors of length f_{2M} can be accommodated in memory at



FIRST
PARTITIONED BY EQUATION 6.4 INTO SUBSETS:
NEXT
EACH SUBSET LARGER THAN f_{2m} AGAIN SPLIT RECURSIVELY
CONTINUED...
PARTITIONING RECURSIVELY REPEATED UNTIL ALL SUBSETS OF SIZE f_{2m} OR f_{2m-1}
← N WORDS → (5)
$S_N = f_{2N}$ (89)
$S_{N-1} = f_{2N-2}$ (34)
S_{N-2} (13)
f_{2m}
f_{2m-1}
UNCHANGED
13
5
3

Figure 6.2 S_N states split by eqn. 6.4 into segments of size f_{2M} and f_{2M-1}

In this illustration, N=5 and M=2, so that S_N=89, f_{2M}=5, f_{2M-1}=3 once. Here as before, f_n defined in equation 4.2 is the n-th Fibonacci number. The set of states indexed from $1,\ldots,S(N)=f_{2N}$ is split into N-M+2 subsets:

$$\begin{aligned} S_N = f_{2N} &= f_{2N-2} + f_{2N-4} + \ldots + f_{2M} + f_{2M-1} + f_{2N-2} \\ &= S_{N-1} + S_{N-2} + \ldots + S_M + f_{2M-1} + S_{N-1} \end{aligned} \qquad \ldots. 6.4$$

that is, from 1 to S_{N-1}, from $(S_{N-1}+1)$ to $(S_{N-1}+S_{N-2})$ and so on. As can be seen in figure 6.2, the first S_{N-1} states are all those which begin with a 1-word block, and so on until the group of size S_M which are all those beginning with an M-word block. The term f_{2M-1} which follows is the number of states which begin with a block of M+1 words or more, and the final S_{N-1} states are all those for which the first word of memory is empty.

After this initial partitioning, all the segments larger than S_M, those of sizes S_{N-1}, S_{N-2}, ... S_{M+1}, are themselves partitioned in the same way, and this process is repeated until all the resulting segments are either $S_M=f_{2M}$ or f_{2M-1} in size. There will be $f_{2(N-M)}$ of size S_M, and $f_{2(N-M)-1}$ of size f_{2M-1}. The resulting set of segments forms the partitioning of the S_N states with which the computation is carried out.

6.2.3 Multiplying a partitioned vector in stages by the transition matrix

At the start of each iteration in this implementation of the power method, there are three partitioned vectors in secondary storage. The first called $\underline{\pi}_0$, is to be multiplied by P and the result placed in the second, called $\underline{\pi}_1$. A third work vector $\underline{w}$ is initialised to contain all zeroes. The computation proceeds in several stages.

For n = 1,2,...,N the following is carried out: $(\underline{\pi}_0 \times Q_n D)$ is placed temporarily in $\underline{\pi}_1$, and then $(\underline{\pi}_1 \times A_n)$ is accumulated in $\underline{w}$. After n=1, $\underline{w}$ contains $\underline{\pi}_0 Q_1 D A_1$; after n=2 $\underline{w}$ contains $\underline{\pi}_0(Q_1 D A_1 + Q_2 D A_2)$; and eventually after n=N, $\underline{w}$ contains $\underline{\pi}_0(\Sigma Q_n D A_n)$.

Next, $(\underline{w} \times (I-A)^{-1}T)$ is placed in $\underline{\pi}_1$, which then contains $\underline{\pi}_0(\Sigma Q_n D A_n)(I-A)^{-1}T$.

Next, $(\underline{\pi}_0 \times T'D'T)$ is accumulated in $\underline{\pi}_1$. D' is the same as the deallocation matrix D except that it has a final zero column, and T' is the pseudo-inverse of T defined by equation 5.16 in section 5.2.2. From section 5.2.4,

$$T'D'T = \sum_{n=1}^{N} Q_n D T_n \qquad \text{.... 6.5}$$

After this operation therefore, the vector called $\underline{\pi}_1$ contains the final product. A simple linear scan is made to collect the utilisation and, in order to be able to compute the expected internal fragmentation E[IF] in equation 6.10 below, the subtotal sums of the individual probability elements of $\underline{\pi}_1$ separated into subgroups according to the number of allocated blocks in each state. As a check, the sum of all the elements of $\underline{\pi}_1$ should be unity, and so this sum is also collected for inspection after each iteration. After a test for convergence, the pointers to $\underline{\pi}_0$ and $\underline{\pi}_1$ are interchanged and then the next iteration is started if convergence to sufficient accuracy has not been reached.

The details of each of these stages are complicated. Although simple in principle, the program was not easy to write and extensive and careful checking was necessary to eliminate mistakes.

The order in which the procedure written to calculate $(\underline{\pi}_0 \times Q_n D)$ performs its operations is illustrated in figure 6.3(a), and it follows equation 5.5

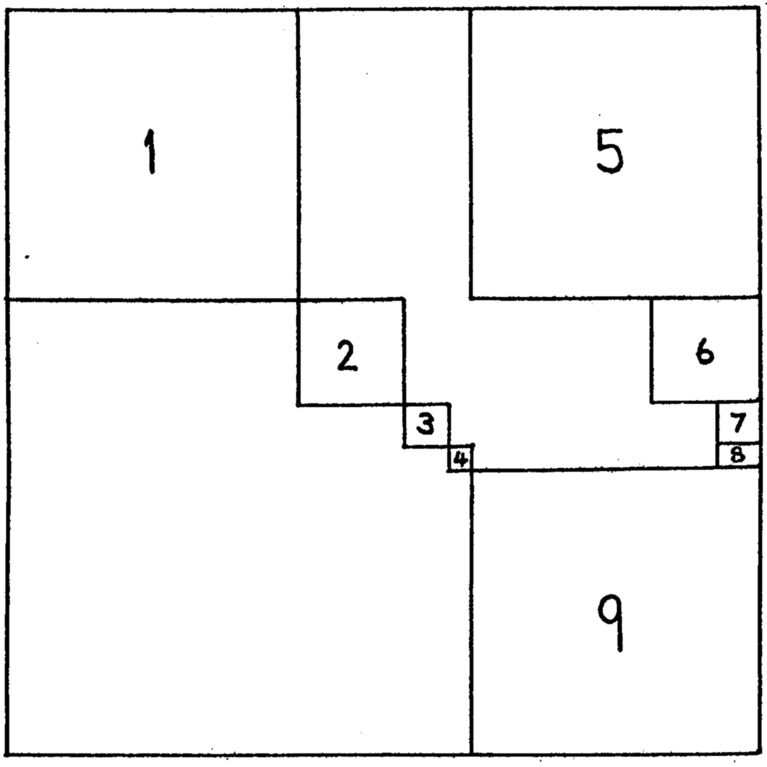
for the recursive composition of the deallocation matrix D. The procedure is written to perform any part of this computation for whatever submatrix of D is passed as a parameter, and it is initially called with parameters to indicate the whole of D. It breaks up whatever submatrix of D it is passed according to equation 5.5, and calls itself recursively passing as parameters the successively diminishing submatrices on the main diagonal, then again recursively calls itself passing the successively diminishing diagonals on the right hand side and finally calls itself for the submatrix in the lower right corner. No computation takes place until a call of this procedure is passed a parameter to indicate a submatrix of D equal to one or other of the segment sizes f_{2M} or f_{2M-1}. When this happens the appropriate segment of $\underline{\pi}_0$ is read into main storage and is first multiplied by the corresponding segment of Q_n. Since Q_n is diagonal this requires only a linear scan of the segment of $\underline{\pi}_0$. This vector is then used to compute the output segment to be written to $\underline{\pi}_1$, by multiplying it by the specified submatrix of D. The arrangement of this multiplication is also performed recursively in exactly the same way that D is recursively split into f_{2M} and f_{2M-1}-sized segments. This is possible because the recursive structure of D expressed in equation 5.5 does not of course stop just because the arbitrary segment size M is reached, but continues to be valid for smaller submatrices. The order in which each segment-sized submatrix of D is dealt with as a result of this recursive splitting is illustrated in figure 6.3(b). $\underline{\pi}_1$ is initialised to zero before this multiplication is started, and each segment of the matrix requires one read of the corresponding segment of $\underline{\pi}_0$ from secondary storage, a read of the segment of $\underline{\pi}_1$ (since a previous submatrix in the same column but a different row position of D may have already accumulated some probabilities in this segment of $\underline{\pi}_1$), and a write of the updated $\underline{\pi}_1$ segment back again to secondary storage.

The other stages of the computation for one iteration, $(\underline{\pi}_1 \times A_n)$,

(a)

NOTE:

N-M = 3
IN THIS
EXAMPLE



(b)

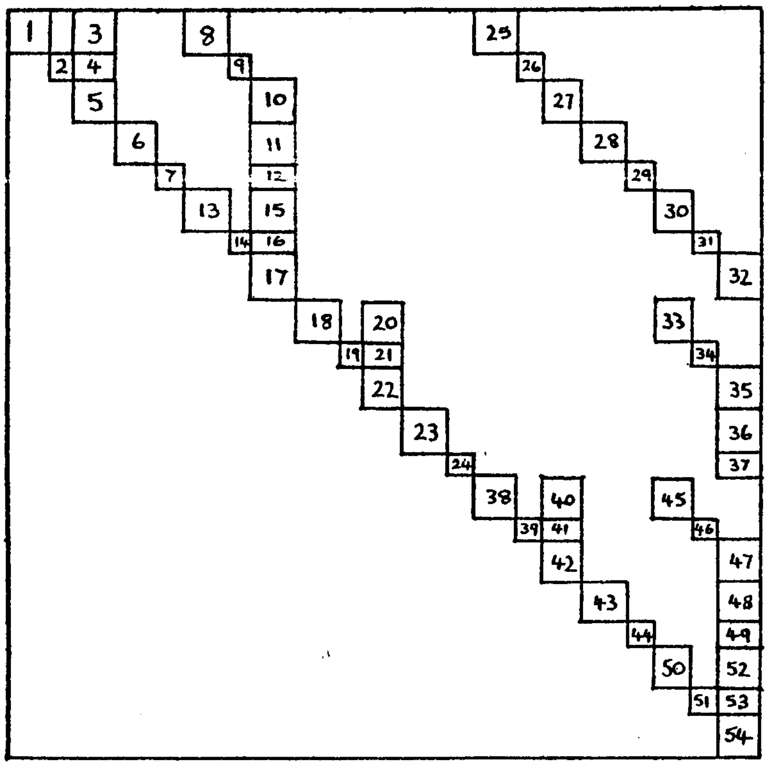


Figure 6.3 Multiplication by $Q_n D$: the order in which segments are computed

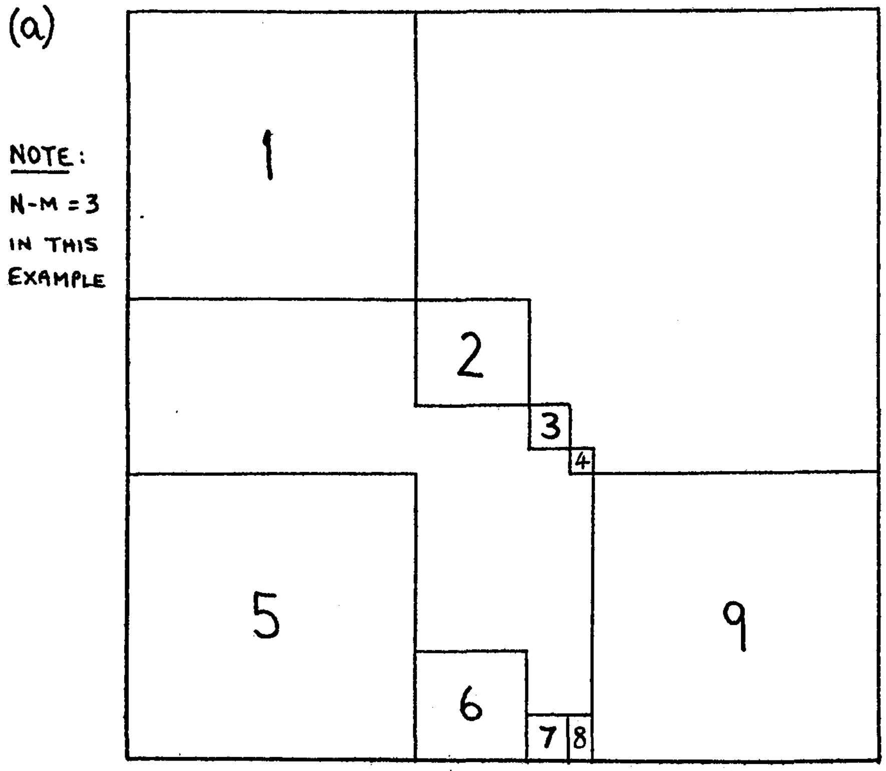
The top figure (a) shows how a submatrix of D is split by equation 5.5 into smaller submatrices, and the order in which these are processed.
The bottom figure (b) shows the resulting split into submatrices of size f_{2M}, f_{2M-1} and the order of processing these. Compare figures 5.1, 6.2.

($\underline{w} \times (I-A)^{-1}T$), ($\underline{\pi}_0 \times T'D'T$) are similar in the manner by which the computation for the whole matrix is recursively split up until segments of a size for which vectors can be brought into the program's main storage are achieved. Each computation contains only one basic matrix (A_n; $(I-A)^{-1}$; or D') which is non-diagonal, the multiplication by the diagonal T' and T matrices being accomplished as it is for Q_n by a single sequential scan of the input or output vector, either just after being read in or before being written out to secondary storage.

The multiplication by T'D'T is very similar to that by Q_nD. A check is necessary to make sure that the elements in the final column of D' are set to zero. T has elements c_{g_i+1}, and T' has reciprocals of these, where as before g_i is the size of the largest gap in state i and c_{g_i+1} is the probability that a request will be for a block larger than this size. The calculation of the elements of T and T' is therefore straightforward and, as in the case of Q_n which has elements r_n/c_{g_i+1} or 0, depends only on the maximum gap size g_i in each state i, which has to be calculated each time it is required.

The multiplication by A_n, and by $(I-A)^{-1}$, is arranged in a way which allows any possible allocation algorithm to be as easily implemented, and in just the same way, as it would be in any actual storage allocation system. The arrangement of the splitting of either of these two computations into segments is the same in both and is illustrated in figures 6.4(a),(b); it mirrors that of the deallocation matrix. Figure 6.4(a) shows how any submatrix which is still too large is split first into successively smaller blocks down the main diagonal, then into correspondingly sized blocks along the bottom, and one final block in the bottom right hand corner. Figure 6.4(b) shows the order in which the submatrices of size f_{2M} and f_{2M-1} into which the matrix is eventually split, are processed as a result of arranging

(a)



(b)

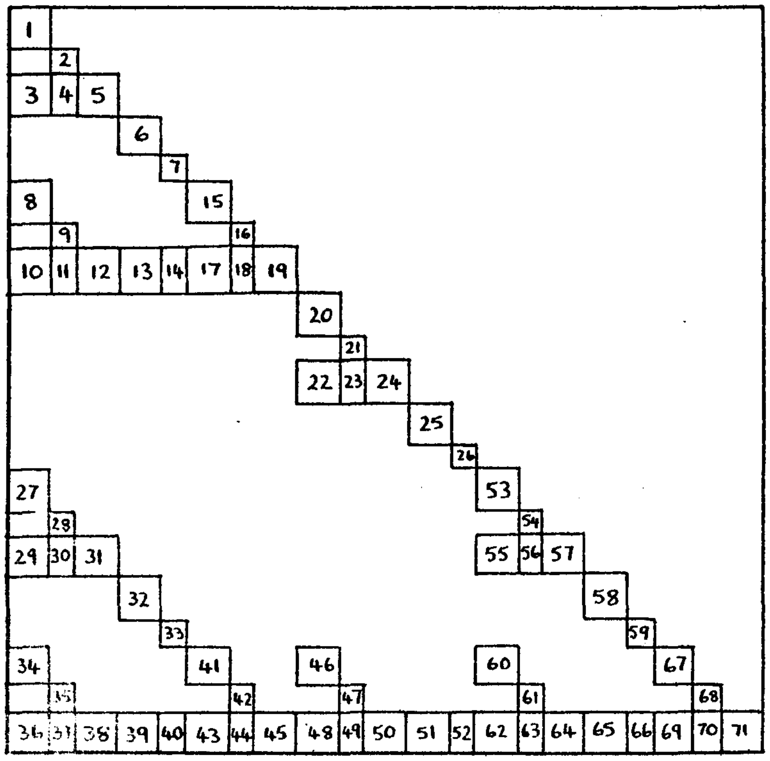


Figure 6.4 Allocation matrix multiplication, A_n or $(I-A)^{-1}$: order of segments

(a) Top figure: how each submatrix is recursively split, and the order

(b) Bottom figure: the order in which segments are processed as a result the recursion in this order. When such a submatrix is reached, the input segment of either $\underline{w}$ or $\underline{\pi}_0$ is read into main storage as before but the computation within the submatrix is not recursively split as it can be for the deallocation matrix, owing to the more complicated possible variation of the elements in either of these allocation matrices, A_n or $(I-A)^{-1}$. Instead, each row or input state of the submatrix is considered in turn and the configuration of memory which it represents is calculated as a sequence of gaps and blocks. For multiplication by A_n, the allocation algorithm is invoked with this configuration and request size n, to find the resulting possible configuration, or configurations, and their probabilities if more than one is possible with a non-deterministic algorithm. If any of these configurations have state indices within the columns of the submatrix, then the probabilities which are the elements of the submatrix are used to perform the resulting multiplication by contributing their product with the input vector element to the correct place in the output vector. Resulting allocated configurations with indices outside the column range of the submatrix are discarded (they will cause contributions when another submatrix is considered), and also so are the cases for which no allocation is possible. This method has the disadvantage that it can compute many of the allocation transitions from each state for all the submatrices lying along the row corresponding to the state, in order to find and use only those which lie within a given submatrix, and so has to "throw away" a fair amount of computation. On the other hand, because it disregards any (possibly very complicated) structure possessed by the allocation algorithm, which would require a different traversal of the structure to be re-programmed in each case, this method is able to cope with any possible allocation algorithm. The algorithm itself is programmed in the same way that it would be in a real situation, by placing a block of a given size into a specified configuration of the memory, and not by reference to any structure that may be imposed on the allocation matrix.

For the more complicated multiplication by $(I-A)^{-1}$, this is of course equal to the sum $(I+A+A^2+...)$ and this is the way in which the multiplication is effected. The procedure is the same as for the multiplication by A_n. A check is made for each segment-sized submatrix, to see if it is lying on the main diagonal, and if it is then the entire input vector segment is contributed to the output segment to give the multiplication by the identity matrix. Allocations are then computed row by row as for A_n. In this case there is a recursion, but a different one from the structure derived method of partitioning which obtains the segment sized submatrices. The allocation of all possible sizes of request from 1 to N is attempted into the currently reached configuration at each stage. Each allocation that is made is not only checked to see if it lies between the column boundaries of the particular segment-sized submatrix being considered to give the contributions for the multiplication by A, but is also used as the starting point for further allocations to give the multiplications by elements of A^2, A^3,... and so on. The accumulation products of these double, triple, ..., allocations are kept in a stack and all possible multiple allocations are enumerated by backtracking. Whenever an allocation is unsuccessful, then all larger request sizes at the current level must also fail, so that they need not be tried, and the stack level can be reduced by backtracking to try the next size of request at the previous level in the stack. The same is true when an allocation is successful but has a lower index than the first or leftmost column of the output segment; it is not required for this segment since it is outside the segment columns and also all its successors in still higher powers of A will lie even further to the left because of the particular index order of the states, so that they need not be considered. Both these tests serve to cut down the amount of backtracking computation. When an allocation is successful and the index of the resulting configuration has not become too low for the output column range, the configuration and its accumulated probability product are stored and used to begin a new level of the backtracking stack.

6.2.4 Reducing inefficiencies in the "no matrices stored" implementation

There are a number of ways in which this revised implementation of the power method as described so far would be more inefficient than necessary, if no further action was taken to avoid them. The main cause of inefficiency is that many of the segment sized submatrices contain all zeroes or operate on segments which become all zero at a particular stage of the computation. Checks have been included wherever possible in the program to avoid computing numbers which must always be zero.

The array of information describing whereabouts on secondary storage each segment of the vectors $\underline{\pi}_0$, $\underline{\pi}_1$, $\underline{w}$ can be found, also describes segments which are known to contain all zeroes. This description of zero segments is kept up-to-date as segments are rewritten. Whenever such a segment is input to an operation which must as a consequence produce all zeroes, the operation is bypassed at the point of input if it has not already been avoided by another check. Even if the operation is not nullified, for example if the segment is from the as yet still empty output vector being accumulated, the input operation itself is avoided and the input vector merely set to zero instead. Setting an entire output vector to zero becomes the simple operation of setting all the segment descriptors to indicate zero.

Operations which in general update vectors by first reading them in, modifying them and then writing them out again, check that modification has actually occurred as the operation proceeds, and if there is no change, the output operation is bypassed.

Many of the segments of $\underline{\pi}_0$ which are input to the multiplication by Q_nD, $n = 1,\ldots,N$, are completely set to zero by the multiplication by Q_n, especially the later segments and for low values of n. The maximum gap in the first state of each segment is the least such maximum for all the states in the segment, and if it is not less than the request size n then the multiplication by Q_nD, which would produce a zero segment after the multiplication by Q_n, is bypassed. Approximately half these particular segment computations are avoided in this way.

As has already been mentioned, in the multiplication by $(I-A)^{-1}$, the backtracking computation of all possible multiple allocations is cut off at any stage by observing whether the resulting state from any particular allocation lies outside and to the left of the columns of the particular submatrix being considered. Since further allocations into this state can only result in states even further to the left, there is no point in attempting them, so that the next request size at the current level can be immediately attempted.

The recursive splitting of the general form of an allocation type matrix into segment sized submatrices is wasteful of computing time as it is more general than is necessary for the multiplication by A_n, since the blocks along the bottom of the matrix in figure 6.4(a) not including the last one can only be diagonal. Further, only one of these diagonal blocks can have non-zero entries if at all, corresponding to n-word allocations into the beginning of the subset of memory represented by the allocation-type matrix, which can only occur if the subset of memory is at least n words long. The segmentation of much more than half of all the submatrices in the complete multiplication by

A_n is avoided by only computing the correct bottom diagonal block.

In the two stage multiplication by Q_nDA_n, there are some segments of $\underline{\pi}_1$ computed from $\underline{\pi}_0Q_nD$ which, although they contain non-zero probabilities, are unused in the subsequent multiplication by A_n. This happens if the state corresponding to the last column of the particular submatrix of D, which will be the state containing the largest maximum gap of all the output (column) states of this submatrix, still has no gap large enough to allocate a request of size n. It is simple to check for this during the partitioning of Q_nD into segments and bypass those for which this is the case.

As a further comment on the efficiency of the computation, the simple action of turning off array subscript bound checking which by default is normally performed in Algol W, reduced the processing time required by the finished program to three quarters that of the time needed with subscript checking. A program performance measuring tool of the MTS operating system on the 370/168 called TIMETALLY was used on the working program. TIMETALLY divides the memory space occupied by any program it is measuring into separate areas of arbitrary equal size, and then samples and records as a histogram the areas from which instructions are being fetched while the program is running. This useful measurement tool gives a good indication of which parts of a program use the most processing time. For the present program it showed that between a third and a half of the time was being spent in four particular procedures, three of which are concerned with converting state indices to state configurations and vice versa, and the fourth was the procedure which implemented the allocation algorithm by placing a given request into a given configuration. Normally, this indication would be sufficient justification for examining these reasonably simple procedures with a view to replacing the Algol W with Assembler code for example, to make them faster. Although they were examined, this extra recoding effort, which it was estimated would possibly have saved as much as another twenty or thirty per cent of the time used, was judged not to be worthwhile. The exponential growth in the number of states and processing time caused by increasing the memory size N means that an increase in speed by a factor of more than three times is required to be able to compute one more point on any graph of performance (utilisation for example) against increasing values of N.

A necessary convenience in the program's operation is that after each of the (2N+2) multiplication stages in each iteration: $(\underline{\pi}_0 \times Q_n D)$, $(\underline{\pi}_1 \times A_n)$, n=1,...,N, $(\underline{w} \times (I-A)^{-1}T)$, $(\underline{\pi}_0 \times T'D'T)$, all three of the vectors $\underline{\pi}_0$, $\underline{\pi}_1$, $\underline{w}$ and a small number of other necessary variables in the computation are copied to a "checkpoint" file on secondary (disc) storage. This allows the program's operation to be interrupted at any point and resumed later at the most recently checkpointed stage of the computation. When the program is restarted, it cycles quickly around the control structure of the computation until, from a count written with each set of checkpoint information, it determines that the point at which the most recent checkpoint was written last time has been reached. The vectors and all the variables that were saved are restored and the computation then proceeds. By the crude method of doubling the amount of secondary storage checkpoint space available and alternating between essentially two checkpoint files, it was found possible to avoid the chance that interrupting the program at random (by operator intervention, or by job time limit exceeded) during the middle of a checkpointing operation would make the checkpoint data useless by only being partially updated. A pointer indicating which checkpoint file contains the most current set of data is updated in one non-interruptible operation only after all the data has been successfully written. (A better way would be to delay the acceptance of such an interruption if checkpointing is in progress until it is completed. This would be possible with a little more ingenuity. Further savings in the space required to save the intermediate stages of a computation could have been made by noting which vector changes, and which two do not, in each stage of the computation between checkpoints and only saving a copy of the one that has changed. These precautions and measures were not found to be necessary as in practice there was sufficient temporary space on secondary storage for the described operation to be carried out.)

6.3 Predicting converged values with the epsilon sequence transformation

As the successive vectors $\underline{\pi}(T)$ computed by the power method converge to the steady state eigenvector $\underline{\pi}$ for increasing T, the utilisation:

$$U(T) = \underline{\pi}(T) \cdot \underline{u}' \qquad \text{(scalar product)} \qquad \ldots\ 3.1$$

also converges to the steady state value

$$U = \underline{\pi} \cdot \underline{u}' \qquad \text{(scalar product)} \qquad \ldots\ 3.3$$

where $\underline{u}'$ is the column transpose of a row vector $\underline{u}$ with elements representing the fraction of memory occupied in each state, as defined in equation 3.2. If the eigenvalues of the transition matrix P are (λ_s), s = 1,...,S', and the corresponding left eigenvectors are $(\underline{\pi}_s)$, s = 1,...,S', where $\lambda_1 = 1$ and $\underline{\pi}_1 = \underline{\pi}$, then in general the power method produces successive approximations $\underline{\pi}(T)$ to $\underline{\pi}$ of the form

$$\underline{\pi}(T) = \sum_{s=1}^{S'} c_s . (\lambda_s)^T \underline{\pi}_s \qquad \ldots\ 6.6$$

for some set of constants (c_s), s = 1,...,S' determined by the starting vector $\underline{\pi}(T=0)$. It follows that the successive values of U(T) are of the form

$$U(T) = \sum_{s=1}^{S'} C_s(\lambda_s)^T \qquad \text{.... 6.7}$$

where the constants $C_s = c_s\underline{\pi}_s.\underline{u}'$ (scalar product), $s = 1,\ldots,S'$. Convergence follows for the (stochastic, irreducible) transition matrix P because as shown in chapter 3, $\lambda_1 = 1$; $|\lambda_s| < 1$ for all $s = 2,\ldots,S'$. The expression for U(T) in equation 6.7 is just the form for which the so-called "epsilon algorithm" or method of extrapolation is most suitable. This method enables the eventual limit of the convergent sequence (U(T)) to be estimated from the initial terms of the sequence.

For a full description of the epsilon algorithm, see for example Wynn (1961), or more readably, Gragg (1972). Given any sequence U(0),U(1),U(2)... known or guessed to be of the form 6.7 or at least close to such a form, the table of differences in table 6.1 is constructed. This table uses the

	U(0)			
0		D(2,1)		
	U(1)		D(3,2)	
0		D(2,2)		...
	U(2)		D(3,3)	
0		D(2,3)		...
	U(3)		D(3,4)	
0		D(2,4)	.	
	U(4)		.	
.		.	.	
.	.	.	.	
.	.	.	.	

Table 6.1 <u>Array of differences for the epsilon algorithm</u>

"rhombus rule":

$$\begin{array}{ccc} & a & \\ b & & c \\ & d & \end{array} \qquad (d-a)(c-b) = 1, \quad \text{so that} \quad c = b + 1/(d-a) \qquad \text{.... 6.8}$$

or in table 6.1,

$$D(j+1,T) = D(j-1,T-1) + 1/(D(j,T)-D(j,T-1)) \quad . \qquad \ldots. 6.9$$

An initial column of zeroes is placed in the j=0 column, and the sequence to be extrapolated (in the present case, the successive numbers U(0), U(1), U(2),...) appears in the j=1 column. As usual for difference tables, successive columns contain elements staggered halfway between those in adjacent columns. If the sequence U(T) is of the form $U(T) = A + Bx^T$ for some x and constants A and B, then all the differences in the second computed column (D(3,T) for T = 2,3,4,...) will be identically equal to the constant A. If U(T) is of the form $U(T) = A + Bx^T + Cy^T$, then the differences in the fourth computed column (D(5,T) for T = 4,5,6,...) will all be identically equal to A, and so on. The theory behind the epsilon algorithm and the reasons why it works, are not obvious and in fact are far from trivial; they are buried in the analysis of functions and Padé approximants (rational functions of a complex variable). There is an extensive literature much of which can be found by reference to the survey article by Gragg (1972). Because the form of U(T) as given by equation 6.7 is exactly suited to it, the epsilon algorithm has been used in the above described implementation of the power method to give advance predictions of the storage utilisation U from the values U(T). The algorithm costs almost nothing anyway, being simple, cheap and quick to implement, as the rhombus rule is so simple. Only the upward sloping diagonal vector of elements U(T), D(2,T), D(3,T),... in table 6.1 needs to be kept and not the whole table, as this is sufficient to calculate the succeeding diagonal as soon as a new element U(T+1) is available. The results of the prediction obtained from the epsilon algorithm appear with the results in section 6.5. In general its accuracy was such that it was usually giving the value for U finally arrived at by the power method after about half of the iterations of the power method were complete, when the uniform distribution of request sizes was being used, but it was disappointingly much slower for the negative exponential distribution; see section 6.5.2 below.

6.4 An index mapping of the memory configurations onto the integers 1,...,S(N)

In implementing the power method it is both convenient and necessary to be able to map the total number of states S(N) including the empty state, onto the integers 1,...,S(N) so that each integer corresponds to just one state or memory configuration. Whatever index mapping is used thereby defines an ordering of the states. The implementations described used a mapping which gives the same ordering as that in section 4.3 and shown in most of the figures. For completeness' sake and because it is not quite trivial the description of this mapping is given here.

Any memory configuration is described as a sequence of (gap block) pairs as in section 4.3. For any given memory size N a mapping table is constructed and used to map the states expressed as (gap block) pair sequences into the index integers and back again. As an example, table 6.2 shows this for N = 4. Each row of the table corresponds to a (gap block) pair, in the order defined by the relations 4.5 in section 4.3, so that the top row is for the pair (0 1) and the bottom row is for the "special" (no pair) used to indicate a final gap of any length. Each of the N columns of the table corresponds to a memory size n = 1,...,N. Each valid entry of the table contains a displacement from the first index integer, 1, to be used (added) when developing the index integer for a given state.

The table is quite simple to construct. The top row contains all zero entries. There is a valid entry at a given row and column position if the (gap block) pair corresponding to the row can possibly exist (that is, it is

	N=	1	2	3	4
(gap block) pairs :	(0 1)	0	0	0	0
	(0 2)		2	5	13
	(0 3)			7	18
	(0 4)				20
	(1 1)		3	8	21
	(1 2)			10	26
	(1 3)				28
	(2 1)			11	29
	(2 2)				31
	(3 1)				32
	(no pair)	1	4	12	33

Table 6.2 Index table for mapping states to integers, N=4

not too big) in the memory size n corresponding to the column. Where this is not possible a special marker is used to indicate this fact (in the implementation, for efficiency in column searching this marker is made to point to the next valid entry in the column). The difference between any two consecutive valid entries in any given column n, is the number of possible configurations that there are of an n-word memory which begin with the (gap block) pair corresponding to the first entry, and so it is a term from the Fibonacci sequence. For example, there are $f_{2(N-(1+2))} = f_2 = 2$ configurations of an N=4 word memory which begin with the pair (1 2), so that 2 has to be added to the entry in the table in the n = 4 column at the (1 2) row to get the entry at the (1 3) row; 2 + 26 = 28.

For N = 4, to index the state (1 1)(0 1) for example, first consider the pair (1 1). Since there are initially 4 memory words, find the ((1 1),n=4) entry in the N = 4 table which is 21. This pair uses up 1 + 1 = 2 words and so there are 2 left; take the next pair, (0 1), and find the ((0 1),n=2) entry which is 0. With one word left, the final contribution is the ((no pair),n=1) entry which is 1. The required index is then one more than the sum of these, or

$$\text{Index}[\ (1\ 1)(0\ 1)\ ] = 1 + 21 + 0 + 1 = 23$$

Conversely, given the index 23, first subtract 1 to obtain the current "index displacement", and start by searching the N = 4 column of the table for the largest entry not greater than this value 22. In this case it is 21, giving the pair (1 1). This uses up 2 words, so proceed by subtracting 21 from the index displacement and skip to the n = 4-2 = 2 word column, to repeat the search for the next pair. Continue until both word count and index displacement reduce to zero. The special (no pair), if encountered, exhausts the word count immediately.

6.5 Results

The largest size of memory for which it has been practical to perform the computations of either of the above implementations of the power method, is N = 12. It has been found that there are relatively small differences between different allocation algorithms at these low memory sizes, undoubtedly because there are relatively few of the total number of possible configurations in which a genuine choice is possible to any algorithm. Because of this, although the original intention was to calculate values of the expected utilisation U and internal fragmentation E[IF] for a range of allocation algorithms and request distributions, it is sufficient to compare just three algorithms, best fit, first fit and worst fit as well as the results obtained from the relocation model described in chapter 4. The best fit and first fit algorithms were described in chapter 2. Because they give results which are very similar indeed up to N=12, the same computations have been repeated for the "worst fit" algorithm. This algorithm, which is understandably not to be found described elsewhere (certainly not in any practical system), has been designed so that intuitively one may expect that its performance is close to the worst possible for any algorithm without regard to the shape of the request distribution. As explained in chapter 2 section 2.2, given a request to be allocated in a particular configuration of memory, the worst fit algorithm allocates the request into the middle of the largest available gap, leaving an amount of space on either side of the newly formed block in two new gaps which are either equal in size or else different by only one word. (For any particular request distribution it may be possible to devise an algorithm with even lower values for the expected storage utilisation, by taking the distribution into account. For example, if one particular request size, say 10 words, has a very much higher probability of occurring in the queue than any other, then to perform badly it might well be a good idea when allocating requests to try and leave or create as many gaps of just 9 words as possible.) The purpose of trying worst fit was to get at least some experimental indication of how much variation in performance might be possible just by changing the allocation algorithm. The results of all this computation appear in figures 6.5, 6.6 and 6.7 and tables 6.3, 6.4 and 6.5 and show, again at these low memory sizes, that although the difference is appreciable it is not very great.

As with the choice of allocation algorithm, when the largest possible variation in request size is from 1 to 12 words, there is not a lot of scope for comparing the resulting performance for very many different shapes of request size distribution. Having regard to this and to the very few measurements of actual request distributions which have been published as noted in chapter 2, two such distributions have been considered here; negative exponential and uniform. Of course, the computations could be repeated for any distribution. One may intuitively expect that any distribution such as negative exponential in which the smaller request sizes occur more frequently and larger sizes infrequently should result in better performance than is the case with the uniform distribution which gives equal weight to all request sizes.

6.5.1 Convergence of the power method

One reason for preferring to obtain the vector of steady state probabilities by the power method, besides its simplicity, is that it also naturally indicates the transient behaviour from any given starting point. This has turned out to be unimportant, for small memory sizes anyway, as can be seen from the examples in table 6.3, plotted in figure 6.5. Values of the expected utilisation U(T) are shown after successive iterations T=1,2,3... , starting at T=0 from the configuration which was called E in section 4.4.1. Other starting states have been tried, and they give essentially the same behaviour. After an initial short period of adjustment convergence to the equilibrium vector is steady and at a constant rate which is neither very fast nor, fortunately for the power method, excessively slow. The rate of convergence is determined by the sub-dominant eigenvalues of the transition matrix P and is discussed further in the next chapter, in section 7.1. The number of iterations required depends on the memory size N and also on the request size distribution as can be seen from figure 6.5.

It should be noted in passing that another reason for deciding to use the power method is that it is computationally stable. Because any starting vector will always lead by exact computation to the unique steady state eigenvector, any rounding errors which may be introduced into the computation of the successive vectors $\underline{\pi}(T)$ do not build up but automatically decay, the

(a) : Uniform request size distribution, first fit algorithm

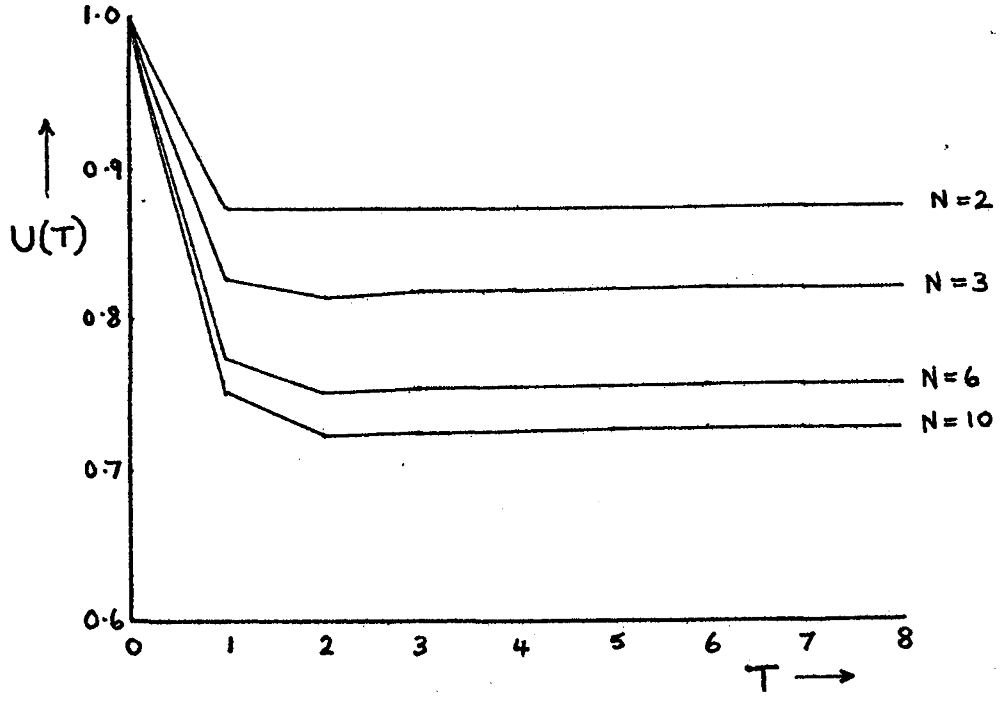
T	N=1	N=2	N=3	N=6	N=10
1	1.0	0.8750000	0.8271605	0.7752308	0.7531167
2	1.0	0.8750000	0.8148148	0.7518197	0.7243359
3	1.0	0.8750000	0.8189300	0.7547872	0.7258510
4	1.0	0.8750000	0.8193111	0.7559696	0.7270166
5	1.0	0.8750000	0.8195334	0.7564573	0.7275472
6	1.0	0.8750000	0.8195831	0.7566610	0.7277897
7	1.0	0.8750000	0.8195934	0.7567464	0.7279012
	...	...	...	...	...
	...	...	...	...	...

(b) : Exponential request size distribution, first fit algorithm

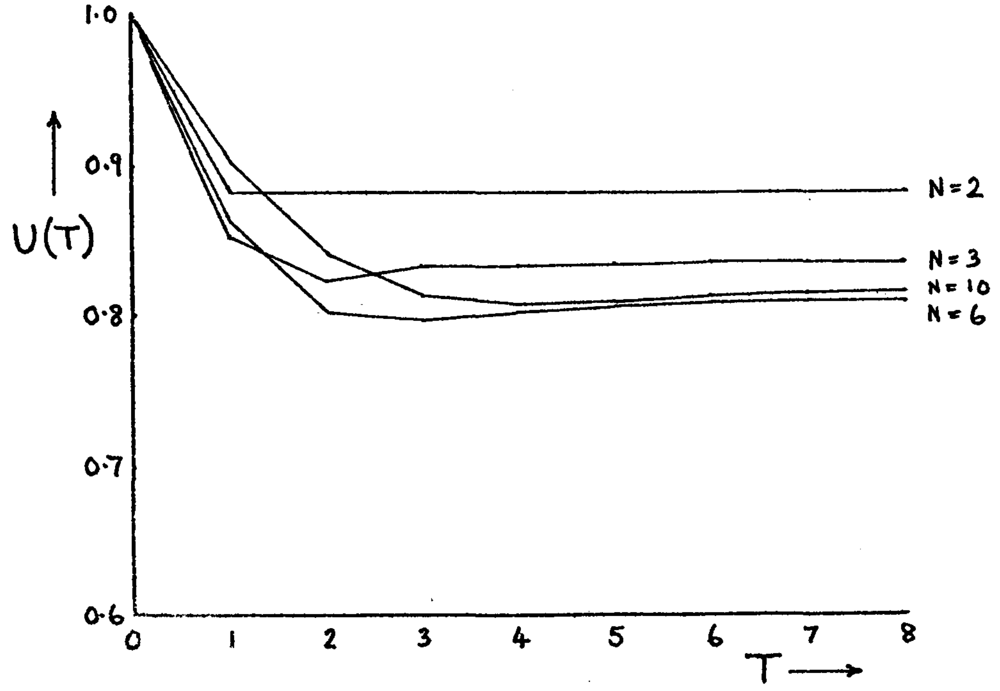
T	N=1	N=2	N=3	N=6	N=10
1	1.0	0.8819660	0.8518250	0.8627041	0.9027041
2	1.0	0.8819660	0.8226989	0.8017398	0.8414428
3	1.0	0.8819660	0.8331066	0.7963910	0.8134895
4	1.0	0.8819660	0.8338175	0.8022597	0.8074937
5	1.0	0.8819660	0.8337341	0.8054673	0.8094640
6	1.0	0.8819660	0.8350006	0.8070752	0.8121949
7	1.0	0.8819660	0.8350908	0.8079403	0.8140472
8	1.0	0.8819660	0.8351125	0.8084122	0.8152171
	...	...	...	...	...
	...	...	...	...	...

Table 6.3 Convergence of expected utilisation U(T) T=1,2,... (see figure 6.5)

effect being to achieve convergence as if from a slightly different starting vector. In practice there were no noticeable problems introduced by rounding in either implementation both of which used the double-length (64 bit) floating point arithmetic of the IBM 360/370 computers. After each iteration a check was made that the elements of each vector $\underline{\pi}(T)$ still summed to unity. Since they always did, to better than at least eight significant figures, the question of rounding errors was not investigated further.



(a) Uniform distribution of request sizes



(b) Exponential request size distribution (defined: equations 4.9,4.10)

Figure 6.5 Convergence behaviour : utilisation U(T) against event time T

6.5.2 The epsilon algorithm: predictions of eventual converged values

The epsilon sequence transformation was described in section 6.3, and it has been used in an attempt to predict the steady state utilisation U as soon as possible from the initial values U(T), T=0,1,2,... as T increases. Typical examples are shown in table 6.4. The results did not live up to expectation for the uniform distribution, and were indeed quite disappointing for the exponential request size distribution. It is just as well that the method is relatively simple, and cheap in its use of storage and execution time.

The pattern of table 6.1 is followed in presenting the values in table 6.4, except as follows. Because the upward sloping diagonals of table 6.1 become available on successive iterations, they are presented as successive rows in table 6.4; and also the intermediate even-numbered columns of table 6.1 have been left out since although they are part of the epsilon algorithm they contain values which are not approximations to the final converged value of U. It can be seen that for the uniform request distribution, the epsilon algorithm produces values for U to any given accuracy no sooner than after about half as many iterations as are necessary for the power method to produce convergence to the same accuracy. Thus although it is certainly not startling its use in this case might be said to be worthwhile. For the exponential distribution however it is hardly any quicker than the power method itself, being quite obviously fooled to begin with for instance by the initial downward swing of U below the eventual converged value. Since the amount of computation required to get a value of U increases by a factor of about three times for each extra word of memory size, using the epsilon algorithm might allow one further such point to be computed than might otherwise be obtained for the uniform distribution, but it is of no

T	U(T)	D(3,T)	D(5,T)	D(7,T)	D(9,T)	D(11,T)
0	1.0					
1	.75311671					
2	.72433592	.72053802				
3	.72585101	.72577524				
4	.72701662	.73090422	.72720626			
5	.72754715	.72799034	.72796530			
6	.72778966	.72799386	.72799385	.72799527		
7	.72790119	.72799616	.72800038	.72799633		
8	.72795386	.72800098	.72799166	.72799163	.72799500	
9	.72797943	.72800358	.72800634	.72796611	.72800850	
10	.72799215	.72800472	.72800555	.72800547	.72800540	.72800553
11	.72799859	.72800520	.72800552	...(quantities too nearly equal to divide by their difference)		
	...	...	...			
	...	...	...			

(a) N = 10, first fit algorithm, uniform request size distribution

T	U(T)	D(3,T)	D(5,T)	D(7,T)	D(9,T)	D(11,T)	D(13,T)
0	1.0						
1	.90270407						
2	.84144280	.73729468					
3	.81348950	.79003002					
4	.80749368	.80585643	.83039737				
5	.80946395	.80897664	.81377428				
6	.81219490	.80239030	.81392700	.81392278			
7	.81404717	.81795182	.81577345	.81375472			
8	.81521711	.81722308	.81715992	.81970436	.81593826		
9	.81598778	.81747540	.81740006	.81761859	.81716063		
10	.81651164	.81762349	.81778479	.81764484	.81764313	.81680571	
11	.81687232	.81766954	.81768769	.81768133	.81749357	.81767438	
12	.81712331	.81769767	.81773762	.81770948	.81702974	.81701286	.81770847
13	.81730013	.81772162	.81783213	.81792845	.81777642	.81753981	.81776523
	...	...	...	...	...	...	...
	...	...	...	...	...	...	...

(b) N = 10, first fit algorithm, exponential request size distribution

Table 6.4 <u>Epsilon sequence approximations for U from U(T): two cases</u>

Compare table 6.1. Also see table 6.3 and figure 6.5 for the successive values of the expected utilisation U(T) in these examples.

Each row in this table contains entries corresponding to alternate entries in the upward sloping diagonals of table 6.1. All the entries in each row T can be computed from the entries in the previous row (T-1), as soon as the value of U(T) becomes available.

use where it is really needed with the exponential distribution, which takes longer anyway to converge.

Thus, any success with this method is clearly limited in practice. One possible reason may lie in equation 6.7; as explained in section 6.3, the method is only guaranteed to work for sequences of this kind after enough iterations have been made in order to produce at least as many columns in table 6.4 as there are terms in the summation. In this case, there are up to S' terms in equation 6.7 corresponding to the number of eigenvalues, or size, of the matrix P so for any reasonable value of N this guaranteed state of affairs is unattainable. The original hope that the earlier columns might nevertheless produce good accuracy, is clearly at best only partly realised. This lack of success with the epsilon algorithm does not of course rule out the possibility that a different sequence transformation, or another method of extrapolation, might do better.

6.5.3 Computed values of the steady state utilisation and fragmentation

The values obtained by the power method for the expected utilisation U, the expected external fragmentation E[EF] and the expected internal fragmentation E[IF] are shown in table 6.5 and plotted in figures 6.6 and 6.7.

Figures 6.5(a),(b) show the values obtained for U when the request sizes are uniformly or exponentially distributed respectively, for the first fit, best fit and worst fit algorithms and also for the relocating model of chapter 4 section 4.4. (Thus figure 6.6 repeats the values in figure 4.8.) Not all of the models have been computed for N=12. For the first fit algorithm, uniform request distribution, the value of U = 0.7206 accurate to four decimal places was arrived at after 11 iterations, taking 7 hours

		Uniform request distribution (equations 4.7, 4.8)				Exponential distribution (equations 4.9, 4.10, 4.11)			
N	S(N)	U	E[EF]	E[IF]	Both	U	E[EF]	E[IF]	Both
1	2	1.0	0.0	0.5	0.5	1.0	0.0	0.5	0.5
2	5	0.8750	0.1250	0.3125	0.4375	0.8820	0.1180	0.3455	0.4635
3	13	0.8196	0.1804	0.2256	0.4060	0.8351	0.1649	0.2896	0.4545
4	34	0.7901	0.2099	0.1768	0.3867	0.8192	0.1808	0.2646	0.4454
5	89	0.7703	0.2297	0.1450	0.3747	0.8102	0.1898	0.2494	0.4392
6	233	0.7568	0.2432	0.1229	0.3661	0.8090	0.1910	0.2410	0.4320
7	610	0.7467	0.2533	0.1066	0.3599	0.8095	0.1905	0.2353	0.4258
8	1597	0.7391	0.2609	0.0942	0.3551	0.8120	0.1880	0.2318	0.4198
9	4181	0.7329	0.2671	0.0843	0.3514	0.8146	0.1854	0.2293	0.4147
10	10946	0.7280	0.2720	0.0763	0.3483	0.8177	0.1823	0.2276	0.4099
11	28657	0.7239	0.2761	0.0696	0.3457	0.8205	0.1795	0.2263	0.4058
12	75025	0.7206	0.2794	0.0641	0.3435				

(a) : First fit allocation algorithm

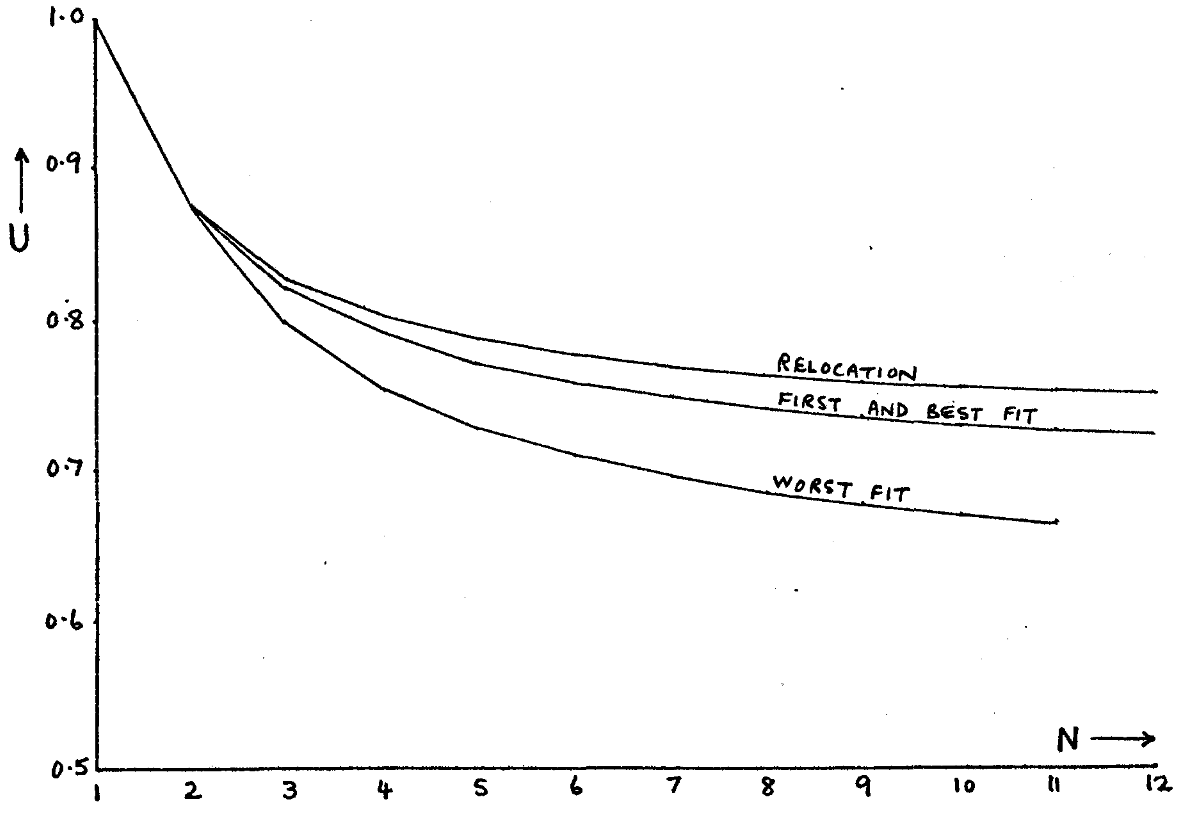
N	S(N)	U	E[EF]	E[IF]	Both	U	E[EF]	E[IF]	Both
1	2	1.0	0.0	0.5	0.5	1.0	0.0	0.5	0.5
2	5	0.8750	0.1250	0.3125	0.4375	0.8820	0.1180	0.3455	0.4635
3	13	0.8196	0.1804	0.2256	0.4060	0.8351	0.1649	0.2896	0.4545
4	34	0.7901	0.2099	0.1768	0.3867	0.8192	0.1808	0.2646	0.4454
5	89	0.7703	0.2297	0.1450	0.3747	0.8102	0.1898	0.2494	0.4392
6	233	0.7569	0.2431	0.1230	0.3661	0.8098	0.1902	0.2412	0.4314
7	610	0.7469	0.2531	0.1067	0.3598	0.8110	0.1890	0.2358	0.4248
8	1597	0.7393	0.2607	0.0942	0.3549	0.8141	0.1859	0.2325	0.4184
9	4181	0.7332	0.2668	0.0843	0.3511	0.8173	0.1827	0.2301	0.4128
10	10946	0.7284	0.2716	0.0763	0.3479	0.8210	0.1790	0.2285	0.4075
11	28657	0.7243	0.2757	0.0697	0.3454	0.8242	0.1758	0.2272	0.4030
12	75025								

(b) : Best fit allocation algorithm

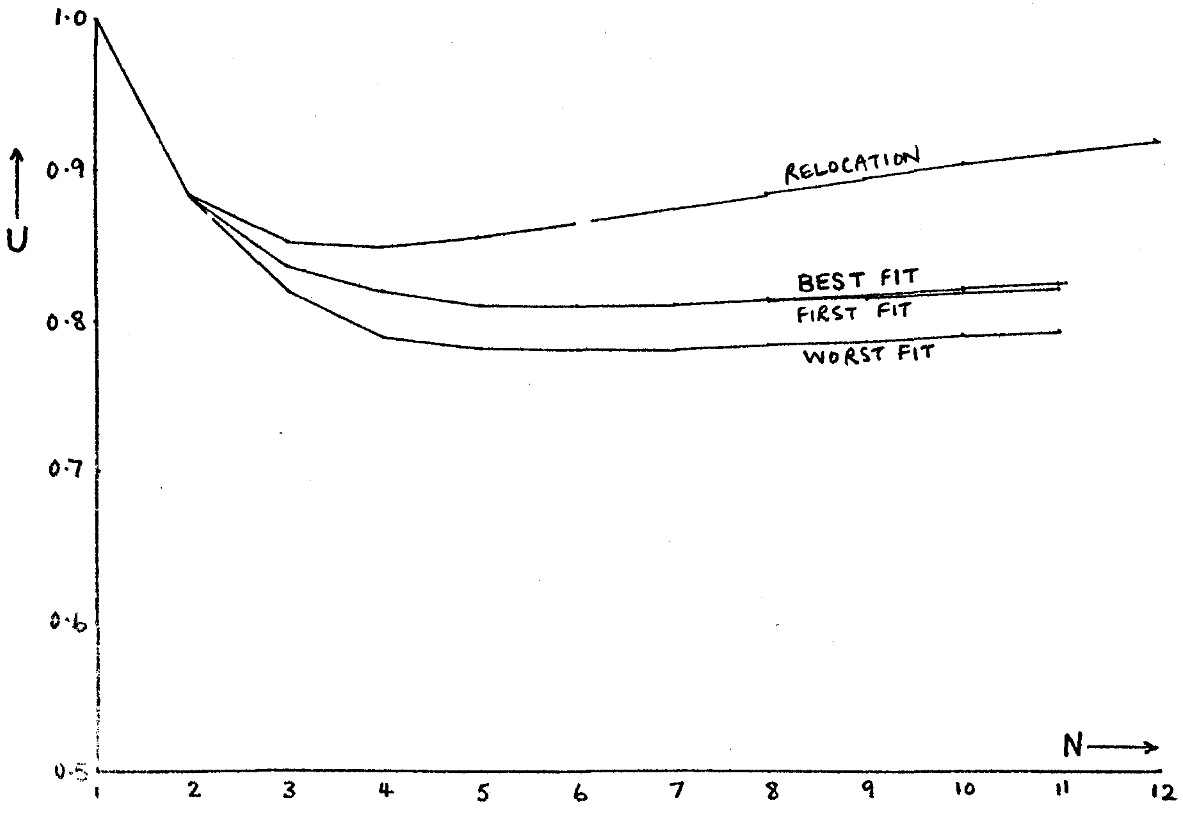
N	S(N)	U	E[EF]	E[IF]	Both	U	E[EF]	E[IF]	Both
1	2	1.0	0.0	0.5	0.5	1.0	0.0	0.5	0.5
2	5	0.8750	0.1250	0.3125	0.4375	0.8820	0.1180	0.3455	0.4635
3	13	0.7981	0.2019	0.2178	0.4197	0.8203	0.1797	0.2838	0.4635
4	34	0.7522	0.2478	0.1667	0.4145	0.7901	0.2099	0.2558	0.4657
5	89	0.7261	0.2739	0.1353	0.4092	0.7818	0.2182	0.2424	0.4606
6	233	0.7077	0.2923	0.1137	0.4060	0.7803	0.2197	0.2344	0.4541
7	610	0.6942	0.3058	0.0980	0.4038	0.7812	0.2188	0.2291	0.4479
8	1597	0.6838	0.3162	0.0861	0.4023	0.7840	0.2160	0.2256	0.4416
9	4181	0.6757	0.3243	0.0768	0.4011	0.7870	0.2130	0.2232	0.4362
10	10946	0.6691	0.3309	0.0693	0.4002	0.7902	0.2098	0.2214	0.4312
11	28657	0.6637	0.3363	0.0631	0.3994	0.7932	0.2068	0.2200	0.4268
12	75025								

(c) : Worst fit allocation algorithm

Table 6.5 Average steady state utilisation and fragmentation: Results

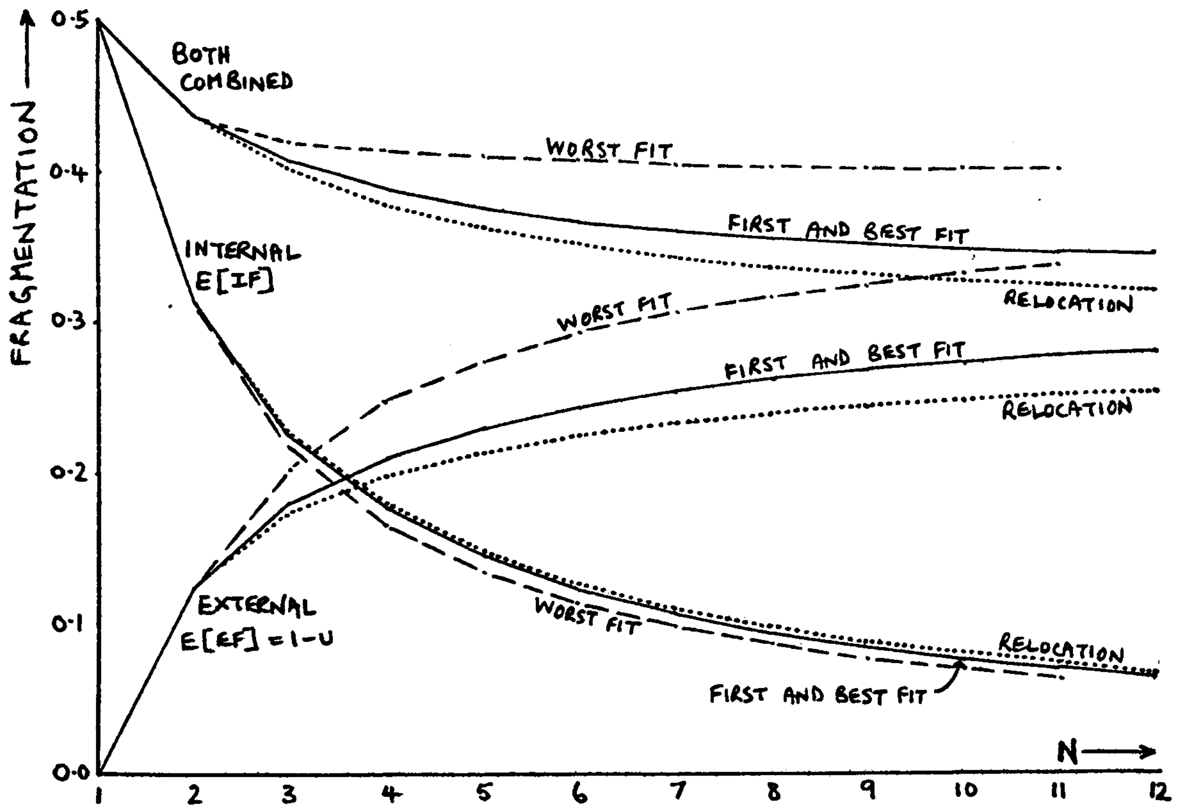


(a) Uniform request size distribution

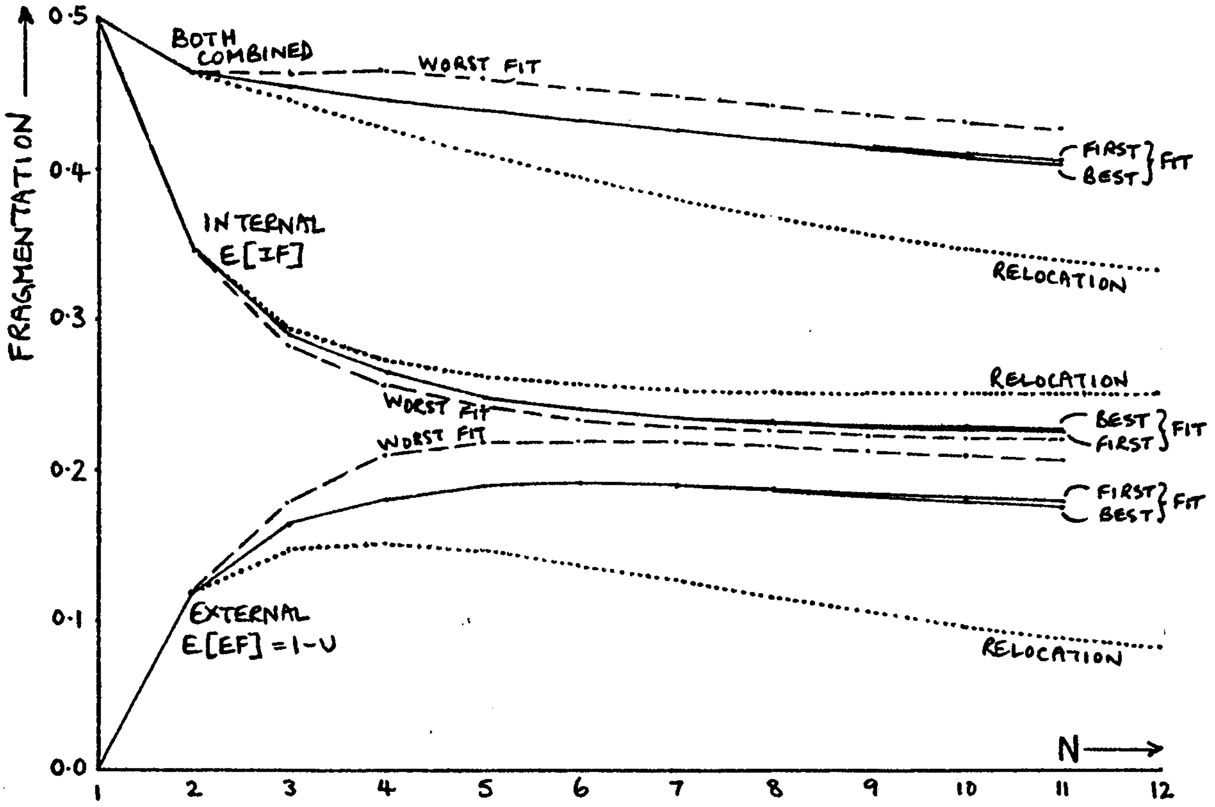


(b) Exponential request size distribution

Figure 6.6 Steady state utilisation U against memory size N : computed values
Values for the best, first, worst and relocation fit algorithms are shown.



(a) Uniform request size distribution



(b) Exponential request size distribution

Figure 6.7 External E[EF], internal E[IF] and combined fragmentation against N

Values for the best, first, worst and relocation fit algorithms are shown.

50 minutes of computation time on the 370/168 computer with the second more efficient implementation described above. In case this seems excessive (and it certainly is a lot of computation) it should be remembered that the calculation had to obtain the principal (dominant) eigenvector of a sparse but large square matrix of size 75024×75024. (To the same scale as most of the matrix diagrams in this thesis, this matrix is over 1080 feet square. It has not, needless to say, been included as a figure.)

As expected, the utilisation in the relocating model is always greater than when the memory may become fragmented. The first fit and best fit algorithms perform very similarly for these small memory sizes as can be seen both from figure 6.6 and table 6.5. Even the so-called worst fit algorithm still achieves a utilisation not much more than 10 per cent less than the maximum possible with relocation. However, small as the difference is, there is a steady divergence between the algorithms as N increases and it is interesting to guess or extrapolate from these figures to values of N one or more orders of magnitude greater than this. The known closed forms of equations 4.20, 4.23 for U in the relocating model should presumably guide the choice of function to be used to fit the values of U for the other algorithms. Judging from figure 6.6 the indication seems to be that as N increases, U will always approach some constant limit asymptotically just as it does in the relocating model. So far however, the author has had no success with this approach. The extrapolation must be made to fit as accurately as possible to the existing data if it is to be of any use, since for instance the first and best fit algorithms may reasonably be expected to differ appreciably in their performance for large values of N whereas the differences so far are still very small.

It will be noticed that, contrary to the results reported by Knuth (1968) mentioned in chapter 2 section 2.2, that best fit always performs at least as well as first fit for these two distributions, and increasingly better as soon as a difference occurs. This therefore generally supports the results of Shore (1975) who was also comparing the performance of these two algorithms for other request distributions as well; his simulation results indicated that first fit was sometimes better, for some of the other distributions. As already mentioned in chapter 2 however, Knuth's model did not use a saturated request queue, and this possibility will be examined in the next chapter.

Figure 6.7 and table 6.5 show the expected fragmentation for these algorithms, again for the uniform and exponential request size distributions. The expected external fragmentation E[EF] is, as explained in chapter 2 and chapter 4 section 4.4.3, simply defined to be the space not occupied by the allocated blocks; E[EF] = 1-U. Internal fragmentation is introduced by adopting the modifications to the discrete request size distributions described in section 4.4.4 and by assuming that requests are always rounded up to the nearest integer. Assuming that the average space wasted by rounding is then half a word, equation 4.24 can be used to find the expected wastage due to rounding. When the probability π_i of the occurrence of each configuration or state i in equilibrium is known, it is relatively simple to scan this vector $\underline{\pi}$ to form the sum:

$$E[IF] = \sum_{i=1}^{S'} (\pi_i/2N) \times (\text{number of allocated blocks in state } i) \qquad \text{.... } 6.10$$

and this has been done with the results shown.

The values of E[IF] turn out to be even closer together for the different algorithms than the values of E[EF]. Clearly, E[IF] for the relocating model is never less than in any fragmented model since it is always possible to get at least as many blocks into memory in a relocation scheme as it is in the corresponding scheme without relocation. Hence for the uniform distribution, since it is already known from equation 4.27 that E[IF](uniform) tends to zero as N increases, the expected internal fragmentation in any allocation scheme must similarly approach zero at least as quickly. For example, the maximum difference in internal fragmentation between the relocating and worst fit algorithms occurs as early as N=4 (0.1802 - 0.1667 = 0.0135) or N=5 (0.1488 - 0.1353 = 0.0135), if it can be assumed from figure 6.7(a) that nothing unusual happens beyond N=11. For the exponential distribution, the relocation asymptote of E[IF] is 0.25, so that in any fragmented model any eventual limit can be no greater than this.

As noted in chapter 2 section 2.1, having introduced internal fragmentation by modifying the request distribution so that the distribution of rounded up request sizes remains unchanged, the idea of "utilisation" has to be similarly modified. The quantity U is, as before, the average space allocated to (rounded up) blocks, so that U + E[EF] = 1 as before, but the value of U now contains the space wasted by rounding as well as the space originally requested. This effect can be studied by considering the quantity:

$$\text{total fragmentation} = E[EF] + E[IF] ,$$

the true or proper utilisation (equation 2.2) then being the complement of this. The total fragmentation is included in table 6.5 (where it is labelled as "Both") and is plotted in figure 6.7.

6.5.4 Randell's observation on the effects of rounding on fragmentation

The behaviour of the total fragmentation is made even clearer by following the presentation given by Randell (1969). Figure 6.8 is extracted

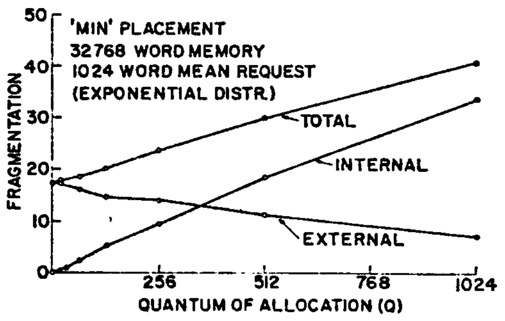
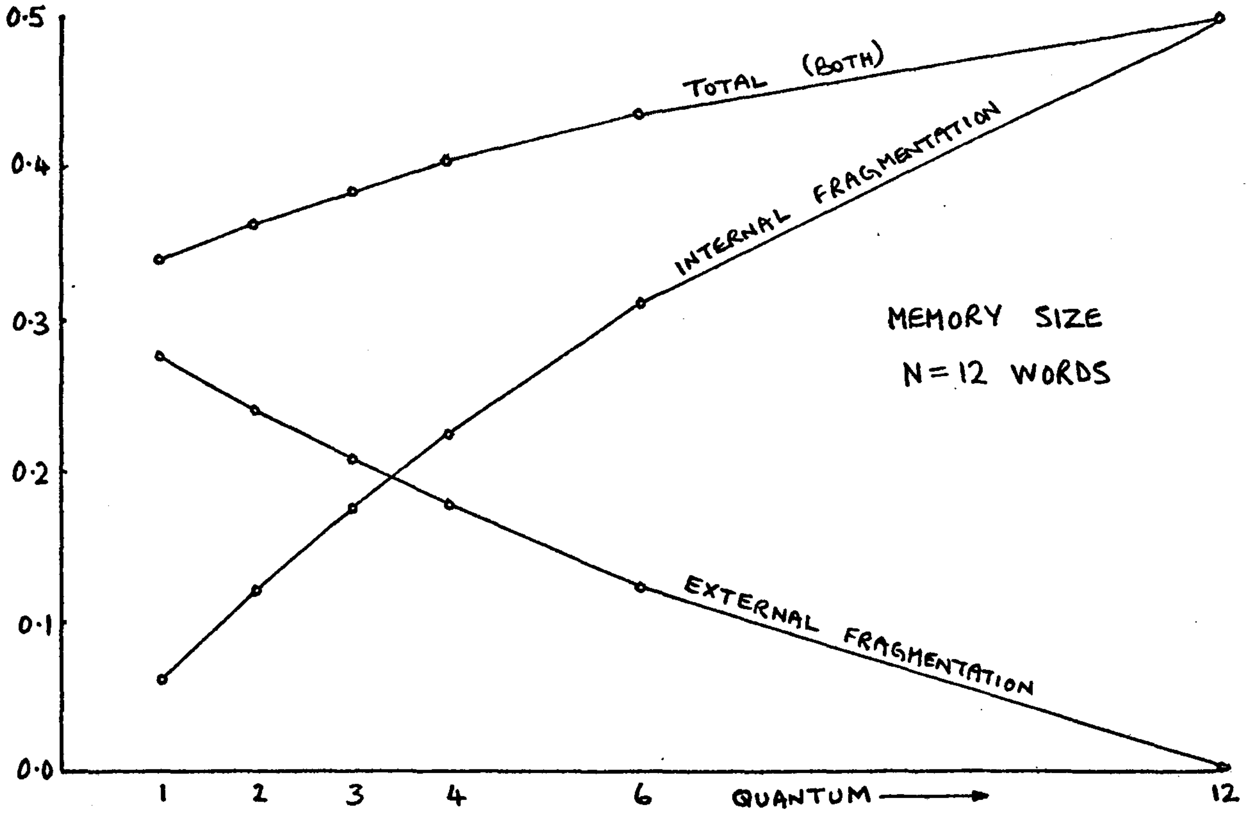


FIG. 2. The effects of rounding up storage requests (exponential distribution)

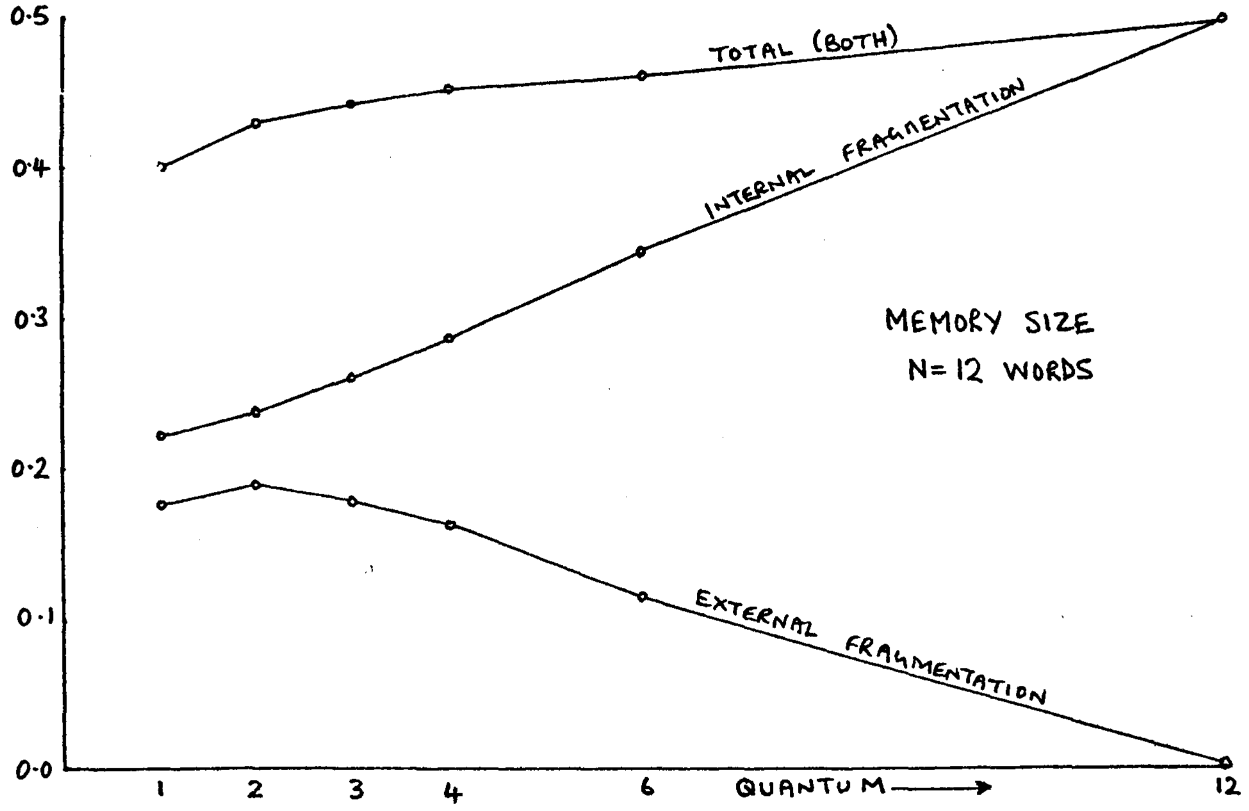
All the results show the unexpected (to the author at least) fact that as Q increases the *loss of utilization due to increased internal fragmentation distinctly outweighs the gain due to decreased external fragmentation.* This is a very interesting result, though of course analytical confirmation, as well as determination of the region of validity of the result, is sorely lacking. In fact very little analytical work has been done in this area at all.

Figure 6.8 Extract from Randell's 1969 paper on storage fragmentation

from his paper. In his simulation, the memory size was 2^{15} = 32768 words, and the rounding quantum sizes were 64, 128, 256, 512, 1024 words as shown. Also, his definition of external fragmentation was the extra space wasted between allocated blocks compared with the inevitable amount left over in the relocation model due to requests not always adding exactly to N. The data plotted in figure 6.8 were obtained by simulations of the storage allocation model. They can be compared with the computed theoretical values in figure 6.7 rearranged to produce the data plotted in figure 6.9. Here, the memory size is taken to be N=12 words, since Randell's choice of N is still (even after allowing for his quantum sizes by dividing by the largest of them) beyond the computing capacity of the present implementation. Following the spirit of Randell's presentation, the quantum sizes which are now integral numbers of words, are chosen to be 1,2,3,4,6 and 12 words. (That is, 12 quanta of 1 word each, or 6 quanta of 2 words each, and so on.) Then since the behaviour of a 12 word memory in which the allocation quantum is, for



(a) Uniform request size distribution



(b) Exponential request size distribution

Figure 6.9 Rearrangement of figure 6.7 to match the presentation in figure 6.8

example, 2 words is the same with the uniform request distribution as in a 6 word memory with a quantum of 1 word, the values of E[EF] and E[IF] for N=6 in figure 6.7 can be transferred to figure 6.9 to correspond to an allocation quantum of size 2 in a memory of 12 words. The other values are made to correspond similarly. In figure 6.8 Randell was using the best fit algorithm (which he called "MIN") with the exponential distribution but in figure 6.9 the (almost equal) first fit algorithm has been used, with both the uniform and the exponential distributions, because that is the algorithm for which it happened that a value with N=12 was computed. The correspondence between the curves for the experimental data in figure 6.8, and the theoretically exact data in figure 6.9, is good, and goes at least some way towards the "analytical confirmation" called for. However, this good agreement says almost nothing about the "region of validity of the result" which will, except for one small point which may be noted here, have to be left to the further analysis of the work which is founded in chapter 5. The small point of interest comes from figure 6.7(b), and it is that, contrary to the general trend, in one case at least it <u>is</u> possible with constant quantum size and increasing memory size for the total fragmentation to increase (or in Randell's terms, for the total fragmentation to decrease with constant memory size and increasing quantum size, as he originally intuitively expected). This is demonstrated by the worst fit algorithm with the exponential distribution, between the values N=2 to N=4. The existence of this possibility makes it indeed interesting to know if more reasonable algorithms can also produce the same behaviour, and under what conditions.

6.6 Computing exact allocation models for small memories: a summary

The simple approach to solving the steady state equation 3.11: $\underline{\pi} = \underline{\pi}P$ of just working out all the possible states, finding all the possible transitions between them with their probabilities and storing these as the transition matrix P, and then applying standard matrix techniques for obtaining the eigenvector $\underline{\pi}$, fails for all but the smallest values of N. Beyond about N=6 words (or S'(N=6) = 232 states), the storage time and computation requirements required for general methods which take no account of the structure of P quickly become excessive as the number S' of states increases exponentially.

That anything at all can be done about this, depends entirely on the particular properties of the transition matrix and what is required from it. Because it is sparse, a storage representation can be chosen to take advantage of this so that by leaving out the zeroes larger matrices can be stored in the same space. That it is also stochastic and usually ergodic as well, plus the fact that at most only the eigenvector of the dominant unit eigenvalue is needed, allows the power iteration method to be considered for obtaining it. Amongst other advantages, this method is able to make good use of the sparse representation of the matrix since the operation to be performed is only that of multiplying a vector by the matrix. Further, this can be managed reasonably efficiently by almost entirely sequential operations so that both the matrix and the vector can recede from main storage which is usually fairly restricted in size, into the comparatively vast background of secondary (disc) storage without (because of the sequentiality) too much of a penalty in increased access time.

The power method allows the possibility of further refinement in at least three ways, although as already seen for one and as will be seen in the next chapter for another, two of these have not yet come to much. Successive approximations to finally converged values produced by each iteration bring the possibility of extrapolating directly to the limit, although the epsilon algorithm which was tried was not particularly successful at doing this. Recent numerical techniques such as simultaneous iteration are based on the power method and can in some cases produce big savings in computation time, but in section 7.1 this is shown to be uncertain for the present models although not every possibility has yet been examined. A third trick which has been used in the present computations but not as yet to its full potential, is to make as good an initial guess as possible at the final converged vector, and to use this as the starting point to reduce the number of iterations required. For example, since best fit and first fit appear to behave so similarly for these low memory sizes, having performed a computation for first fit the finally converged eigenvector of state probabilities can be used to start the computation for the best fit model with the same memory size and request distribution, instead of the rather arbitrary state E which for want of anything better has generally been used. This has been done for some of the computed values in table 6.5. There is also the possibility of using the converged eigenvector from a given memory size N to construct a good guess to be used as the initial vector for the next larger size (N+1) words. If this can be done sufficiently well, then given that the extra storage space is available it may just be possible for instance to produce the converged eigenvector for N=13 in no more time and many fewer iterations than were needed for N=12 starting from the arbitrary state E, the largest memory computed so far. This idea has occurred only recently to the author despite its obviousness and has not yet been properly investigated.

In addition to the sparsity and the various possible ways of speeding up and enlarging on the basic power method, the fact that the transition matrix has a regular recursive structure has been used to advantage to avoid storing it altogether, and this has been implemented with a considerable saving in both storage space for the no longer needed matrix and access time to retrieve it. The detailed knowledge available from chapter 5 of the structure allows further efficiencies in the resulting "multiplication by parts", by avoiding the computing of whole groups of numbers which are either known in advance to be all zero or else which are subsequently unused or are multiplied by zeroes.

One result of all these improvements was the remarkable experience, albeit at the expense of a great deal of machine time, of computing the eigenvector of an (admittedly sparse) matrix of order more than 75000 elements square, and of doing this several times for matrices of order more than 28000. In the end however, none of these techniques affect the exponential nature of the growth of the size of the transition matrix, and so they are all limited much too quickly relative to the effort needed to implement them. An improvement in efficiency by what are in practice large factors of about three, or ten times, resulted each time in only one or two extra points along the graph of increasing N. Doubling the efficiency, normally praiseworthy, was not enough to gain even one extra point in the same computation time and so a number of minor improvements which could have been made (for instance to recode certain heavily used sections of the program more efficiently than is possible in Algol W) were just not worthwhile. It remains to be seen whether any fundamental improvements as yet undiscovered are still possible which will reduce the exponential growth with increasing N, but anything less drastic will probably not be sufficient to justify much further effort than has already been spent.

As for the results which have been obtained, they are already quite valuable in at least three ways. First, they indicate the likely behaviour of models with much larger and more reasonable memory sizes. Although attempts to extrapolate to large values of N from the limited data available have not so far been successful, it probably is possible to do much better. Second, they provide an (admittedly very limited) nursery ground for comparing different algorithms and request distributions. These comparisons can also be made for other similar models which have slightly different rules (some of these are described in the next chapter). Given the limitations set by the number of states on the size of the problem, there is no reason in principle why comparable results should not be possible for these other models. The confirmation of Randell's findings, that rounding up requests to large quantum sizes does not in general make more efficient use of the memory, is a good example of what can be achieved with models of memory size no greater than N=12. Thirdly and probably most importantly, in the usual way that practical experience always improves the use of and complements theory, actually computing and seeing such quantities as the transition matrix probabilities and their relationships to each other in a particular model eventually forces insights into their structure and composition to be noticed which might not otherwise be seen. It is unfortunate that these observations have nearly always seemed to be obvious after they were made so that they should perhaps have been noticed sooner. However it is also true that some of the work in chapter 4 and more especially in chapter 5 that probably would have passed unnoticed otherwise was first realised in this practical way.

Chapter 7 : Further possibilities for analysis and investigation

The investigation presented in this thesis may be extended in several different ways. Besides pursuing the analysis and computation of chapters 5 and 6 further to learn more about the storage allocation model as defined in chapter 3, either the definition of the model or the assumptions on its parameters or both can be altered to obtain related models, and the analytical and numerical methods presented in chapters 4, 5 and 6 can similarly be applied to investigate their behaviour. The author is currently following these lines of enquiry, some of which appear straightforward and promising of results, and will report on their progress in due course.

7.1 Improving the numerical convergence of the computations of chapter 6

The power method used in the last chapter is perhaps the simplest iterative technique that can be applied to obtain the eigenvector $\underline{\pi}$ of steady state probabilities. The successive vectors $\underline{\pi}(T) = \underline{\pi}(T-1)P$ are calculated until sufficient convergence is obtained. This method was used in practice in the first instance not only because it is simple, but because it also indicates clearly the transient behaviour. The successive vectors $\underline{\pi}(T)$ which are obtained and which lead to $\underline{\pi}$ in the numerical computation, happen naturally to be the vectors of probabilities of being in any of the possible states after T events have occurred, starting from some initial state (or probability vector $\underline{\pi}(0)$) at T = 0, and so they are of interest in their own right. The power method can of course suffer from an excessive number of iterations required before sufficient convergence is obtained, however, and this depends upon the magnitudes of the eigenvalues of P nearest to the dominant unit eigenvalue. If μ is an eigenvalue of P having absolute magnitude closest to 1 of the remaining eigenvalues, then $\underline{\pi}(T)$ converges to $\underline{\pi}$ as rapidly as $|\mu|^T \to 0$, so that the closer that $|\mu|$ is to 1, the slower will be the convergence.

There are other iterative methods of finding the dominant eigenvector $\underline{\pi}$ besides the power method, and amongst these is "simultaneous iteration", due originally to Bauer (1958) although he did not give it this name, and particularly its extension to "lop-sided iteration" by Jennings and Stewart (1975). This technique simultaneously calculates the first n dominant left (or right) eigenvectors of a general real unsymmetric matrix, where n may be chosen at will to suit the particular problem. It requires n times as much storage space, and rather more than n times as much computation time per iteration, but as in the power method, the rate of convergence is governed by the relative ratio between two eigenvalues, in this case the first and the (n+1)-th (ordered in decreasing magnitude) instead of the first and second, that is, the relative ratio of the first eigenvalue for which the eigenvector is not being calculated, to the dominant eigenvalue. Stewart (1978) has compared the running times of the power method and lop-sided iteration, and points out that lop-sided iteration can be very much faster than the power method if, for whatever reason, there is a "gap" or sudden sizeable drop in the spectrum of eigenvalue magnitudes whereas the second eigenvalue happens to be close in magnitude to the first. If n is chosen so that this drop occurs between the n-th and (n+1)-th eigenvalues, then lopsided iteration can be much faster even allowing for the extra computation of the (intrinsically unwanted) (n-1) eigenvectors after the first.

Tables 7.1, 7.2, 7.3 show the complete spectra of eigenvalues of the transition matrix P for N = 2,3,4 respectively for the first fit algorithm and

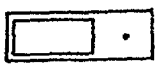
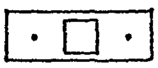
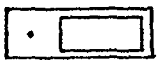
Uniform distribution		Exponential distribution	
Eigenvalues,	Moduli	Eigenvalues,	Moduli
1.0	1.0	1.0	1.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Table 7.1 Transition matrix eigenvalues for N=2, first fit algorithm

Uniform distribution		Exponential distribution	
Eigenvalues,	Moduli	Eigenvalues,	Moduli
1.0	1.0	1.0	1.0
0.316+0.078i	0.325	0.372+0.089i	0.383
0.316-0.078i	0.325	0.372-0.089i	0.383
0.167	0.167	0.272	0.272
-0.066+0.024i	0.070	-0.111	0.111
-0.066-0.024i	0.070	-0.066	0.066
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Table 7.2 Transition matrix eigenvalues for N=3, first fit algorithm

the uniform and exponential distributions used in chapters 4 and 6. The bottom row and right hand column of P have been excluded as they are identically zero, so there are S'(N) = 4, 12, 33 values in each case. There does not appear to be any physical significance to attach to the eigenvalues after the first. Approximately half of the values are zero; some zero values are certainly to be expected from each matrix since certain of the states must have the same set of transitions with equal probabilities, giving rise to equal rows of the transition matrix. For N=3, for example, the states

 7  10  11

Uniform distribution		Exponential distribution	
Eigenvalues,	Moduli	Eigenvalues,	Moduli
1.0	1.0	1.0	1.0
0.390+0.030i	0.391	0.551	0.551
0.390-0.030i	0.391	0.500	0.500
0.281	0.281	0.449	0.449
0.238	0.238	0.325	0.325
0.163+0.039i	0.168	0.289+0.090i	0.302
0.163-0.039i	0.168	0.289-0.090i	0.302
0.139	0.139	0.189	0.189
0.016+0.098i	0.099	0.013+0.131i	0.132
0.016-0.098i	0.099	0.013-0.131i	0.132
-0.093	0.093	-0.120+0.036i	0.124
0.054+0.051i	0.075	-0.120-0.036i	0.124
0.054-0.051i	0.075	0.063+0.063i	0.089
-0.064	0.064	0.063-0.063i	0.089
-0.016+0.043i	0.046	-0.020+0.057i	0.060
-0.016-0.043i	0.046	-0.020-0.057i	0.060
0.034	0.034	0.038	0.038
0.0	0.0	0.0	0.0
⋮ (16 zeroes)	⋮	⋮ (16 zeroes)	⋮
0.0	0.0	0.0	0.0

Table 7.3 Transition matrix eigenvalues for N=4, first fit algorithm

(see figure 4.3) have all the same transitions, since the first action to occur will empty the memory and (since the maximum gap in each state is just one word) leave an identically distributed queue to fill it. Apart from the drop to the first of these zero eigenvalues, there does not seem to be any suitably large sudden drop in modulus in any of these early spectra for small values of N, which would be suitable for simultaneous iteration. Indeed, the biggest drop for these early matrices is the first one, so that the power method does not look at all bad, and may be the best of the iterative methods after all.

In another recent paper by Stewart (1977), he points out that simultaneous iteration can also be applied to a transformed version of the steady state equation, for instance by writing

$$P = L + D + U \qquad \text{.... 7.1}$$

where P has been separated into L and U which are lower and upper triangular, and D is diagonal. Then equation 3.11: $\underline{\pi} = \underline{\pi}P$ becomes $\underline{\pi} = \underline{\pi}(L+D+U)$, which can be rewritten for example as

$$\begin{aligned} & \underline{\pi} = \underline{\pi}U(I-L-D)^{-1} \\ \text{or} \quad & \underline{\pi} = \underline{\pi}D(I-L-U)^{-1} \\ \text{or} \quad & \underline{\pi} = \underline{\pi}L(I-D-U)^{-1} \\ \text{or} \quad & \underline{\pi} = \underline{\pi}(L+U)(I-D)^{-1} \quad \text{, and so on} \end{aligned} \qquad \text{.... 7.2}$$

Quoting Stewart, "it is desirable to apply the iterative methods (simultaneous iteration, etc.) to that matrix which yields convergence in the smallest number of iterations, i.e. the matrix whose subdominant eigenvalues are, in modulus, furthest from unity". Thus it is possible that such a rearrangement could result in faster convergence even where simultaneous iteration applied directly to P would produce little or no advantage, though this has yet to be investigated.

It was disappointing that the epsilon algorithm extrapolation technique which was tried in order to short cut the iterations by predicting the converged values in advance after the first few iterations, did not in practice work very effectively. If it had done so reliably, then almost certainly models of memory size at least one or two words larger could have been computed in the same time by only producing the first few iterations for T=1,2,3,... and then relying on the values extrapolated from there for large T. More effort put into discovering a better technique may well prove worthwhile; perhaps the epsilon algorithm may do a little better from a more suitable starting state, though this does not seem very likely as in effect any of the successive approximations U(T), T=1,2,3,... can be considered as starting positions (although of course they are all related to each other because they all derive from the initial probability vector $\underline{\pi}(0)$ so that as far as the epsilon algorithm is concerned they may all be tarred with the same brush), just by excluding those parts of table 6.1 (the first few downward sloping diagonals) which depend on the earlier approximations U(0),U(1),...,U(T-1). The extrapolation need not be with the values U(T) but might for instance be applied with some extra effort to the successive vectors $\underline{\pi}(T)$ themselves. Of course computing better estimates for $\underline{\pi}(T)$ is in a sense what simultaneous iteration does.

7.2 Continued algebraic analysis of the reduced steady state equations

The complete surprise of finding the reduction of the steady state equation 3.11 to the simple form of equation 5.20, and the subsequent reduction which was found to be possible by state aggregation to the equations 5.23 constraining the allocation algorithm, lends hope to the possibility that there is further structure in these equations which has not yet been noticed and which could be exploited to understand and predict the model's behaviour. The structure of the transition matrix P as an expansion in terms of simpler matrices which was explained in section 5.1, now seems obvious (to the author) with hindsight and perhaps should have been noticed much earlier than it actually was, in principle if not in detail. However it was not in fact appreciated until the first direct implementation of the power method described in chapter 6 which first calculated and stored the entire transition matrix, forcing attention to be drawn to the repetition occurring not only in P but in its component parts (which this first implementation was in essence already calculating). It seems likely that there is more to be gained from a study of these equations, although the relationships between the various components of the K and K_n matrices for example appears to be certainly no simpler than has been shown. As mentioned in chapter 5, equations 5.23 are in a form where they might be formulated as part of a linear programming problem, for instance to maximise the storage utilisation $U = \underline{\pi} . \underline{u}'$, equation 3.3, by varying the allocation algorithm and this may lead to further simplification.

It might be worth bearing in mind that it is by no means necessary to know all the individual elements of $\underline{\pi}$ to form the product $U = \underline{\pi} . \underline{u}'$, just the sum of all those for each state with the same number of allocated words and hence the same entry in the vector $\underline{u}$. This kind of study may well be productive also for the modified but related versions of the present model described below in section 7.5. Certainly any analysis which allowed one to determine for example which allocation algorithm gives the best storage utilisation with a given request distribution, or even just to compare one algorithm with another to say which was the better, would be most useful.

This latter problem is just the one that is addressed by dynamic programming techniques, for which the states are not grouped to nullify the possible allocation choices but are left separate so that the choices can be compared with each other. It will be interesting to see if comparative statements like the above will be possible by using the known algebraic structure of the allocation and the other matrices to either simplify or extend the dynamic programming analysis.

7.3 Infinite extensions of the storage allocation model

The apparent underlying structure of the transition matrix P looks much the same, whatever the value of memory size N, and this suggests that extending the memory size to infinity in some way might prove worthwhile by simplifying the analysis or providing an alternative viewpoint. Two possibilities have been considered in a previous memorandum (Betteridge (1974a)), either extending the memory indefinitely somehow to the set of all positive integers, or alternatively mapping it onto the real interval [0,1]. They are both presented informally below.

Consider figures 7.1(a,b,c) which for simplicity show the deallocation matrix D defined in chapter 5 instead of the full transition matrix P, for memory sizes N = 2, 3, 4. These same deallocation matrices appear in figures 7.2(a,b,c) also, but the scales of drawing these figures have been altered so that each matrix is the same size. As N increases, the "resolution" or "definition of detail" also increases. Each is still a discrete matrix, the sequence growing exponentially in size with N, compare figures 4.1-4.4, 4.6, 4.7. The constant recursive structure described in chapter 5 differing between the matrices only in the fine detail, is indicated and prompts the question of what could happen to this sequence of matrices in the limit as N increases indefinitely.

One way to consider this follows figures 7.1(a,b,c), and parallels the finite case closely. The memory is infinite (or indefinitely large) and discrete, each word having a positive integer address. Requests are still for a finite whole number of consecutive words, successive sizes being chosen independently at random from some fixed distribution. The range of possible request sizes is in general unlimited in the same way that the memory size is unlimited. The rate at which events are occurring is problematical however;

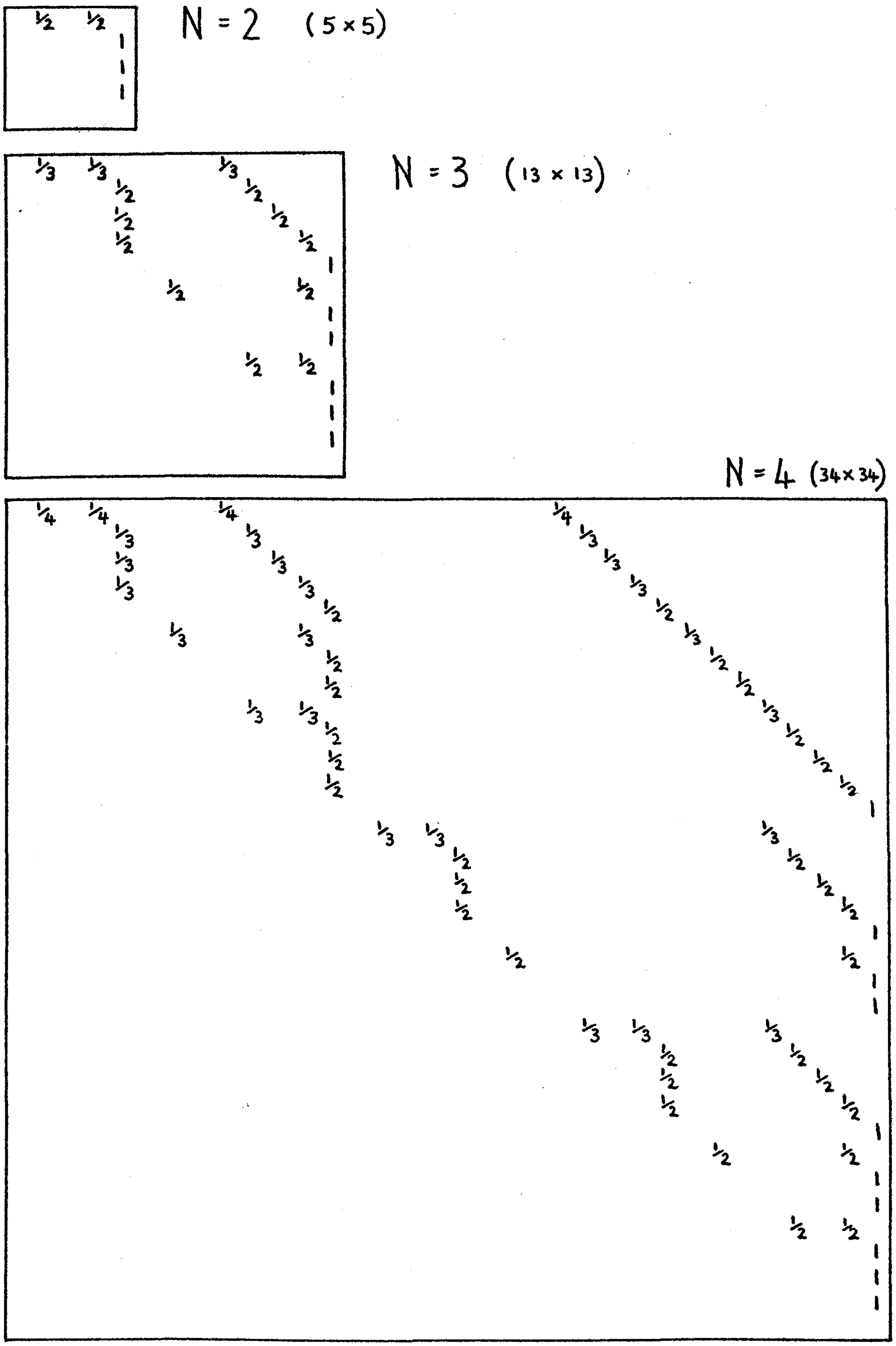


Figure 7.1 Deallocation matrices D for N=2,3,4 (compare figure 5.1)

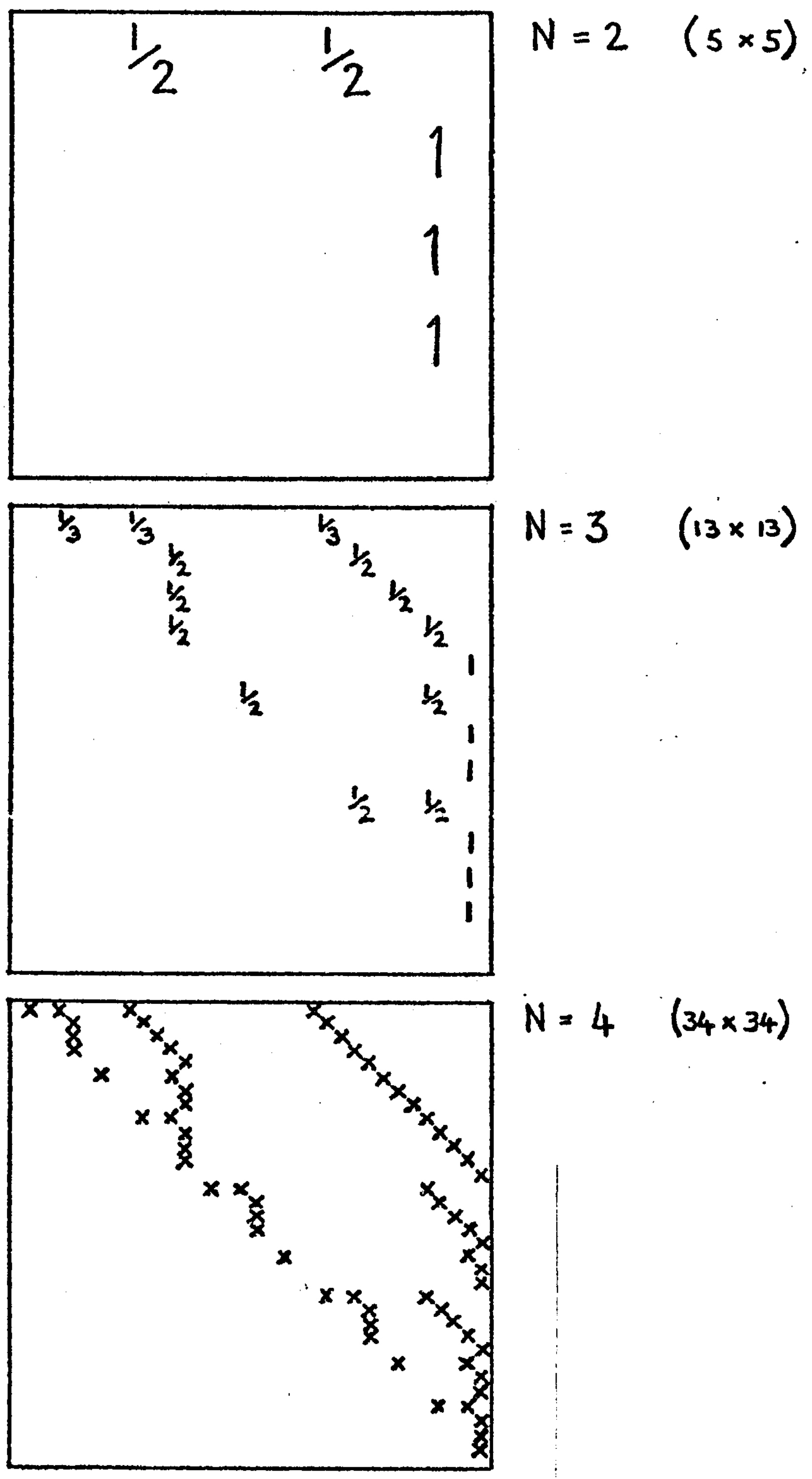


Figure 7.2 Deallocation matrices D for N=2,3,4 (altered scale from figure 7.1)

it has to be speeded up, or else the amount of change in a finite time throughout memory as a whole is indefinitely small. For otherwise if only one block is to be deallocated at a time, then since there are in general infinitely many blocks in memory, the "average" block is assured of an infinitely long tenancy. Also, after such a deallocation has occurred (of a block at a location whose address on average is halfway towards infinity), there is the possibility although with vanishingly small probability, of an unlimited sequence of allocations for requests after the first in the queue all of which can fit into the small gaps already present in memory before the deallocation. These indefinitely long sequences of allocations will presumably be represented by whatever is going to correspond to the matrix product $(I-A)^{-1} = I+A+A^2+\ldots$ in the finite model.

This viewpoint, which was called the "worm's eye view" in the previous memorandum (1974a), is not very satisfactory as there is only a vague idea of how the infinite model works or will be represented, and also no idea of how different it would be from the large, discrete but finite models which are the actual objects of study. However, Reeves (1979, 1980) has considered a similar generalisation with success, the use of generating functions being the key to his analysis.

An alternative proposal for a model to represent the limiting behaviour of the finite models as memory size N tends to infinity is related quite closely to the studies by Renyi, Palasti and others of the random intervals on a line in one and two dimensions, described in chapter 2. It follows the sequence of figures 7.2(a,b,c) and is illustrated by figures 7.3 and 7.4. Instead of a discrete transition matrix $P = (p_{ij})$ of some large but finite order there is now a real-valued function $P_r = P_r(x,y)$ of two variables x, y defined on the unit square $0 \leq x,y \leq 1$. For simplicity in illustrating the

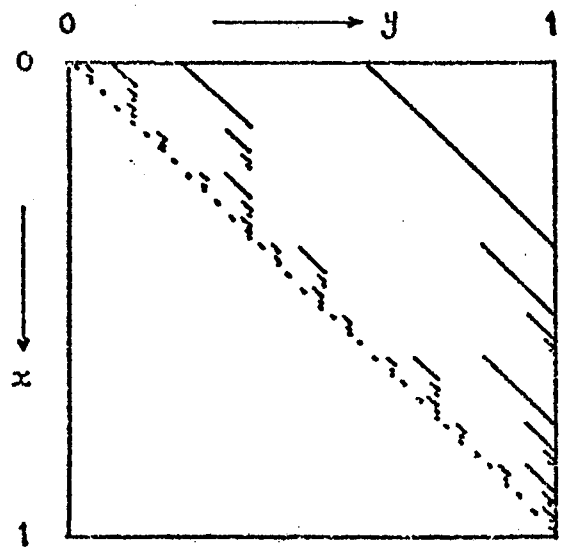


Figure 7.3 Real-valued function $D_r = D_r(x,y)$ defined on the unit square "Bird's eye view"; D_r is zero "everywhere except on the diagonals shown".

principle, the deallocation matrix D appears instead of the full transition matrix P in figures 7.1 and 7.2, and an attempt has been made in figure 7.3 to illustrate the limit of the progression in figures 7.2(a,b,c) as N increases indefinitely. The concept of a discrete word of memory has disappeared and an allocated block is instead specified by its beginning and ending endpoints, the interval [b,e] say, where $0 \leq b < e \leq 1$. Requests for memory specify a size which is a real value, positive and ≤ 1 instead of a positive integer number of words as previously. This is initially promising, for if one were to stand sufficiently far away from a large but finite discrete model so that the fine details become merged, the effect should look the same as this. This proposal was consequently called the "bird's eye view" in the previous

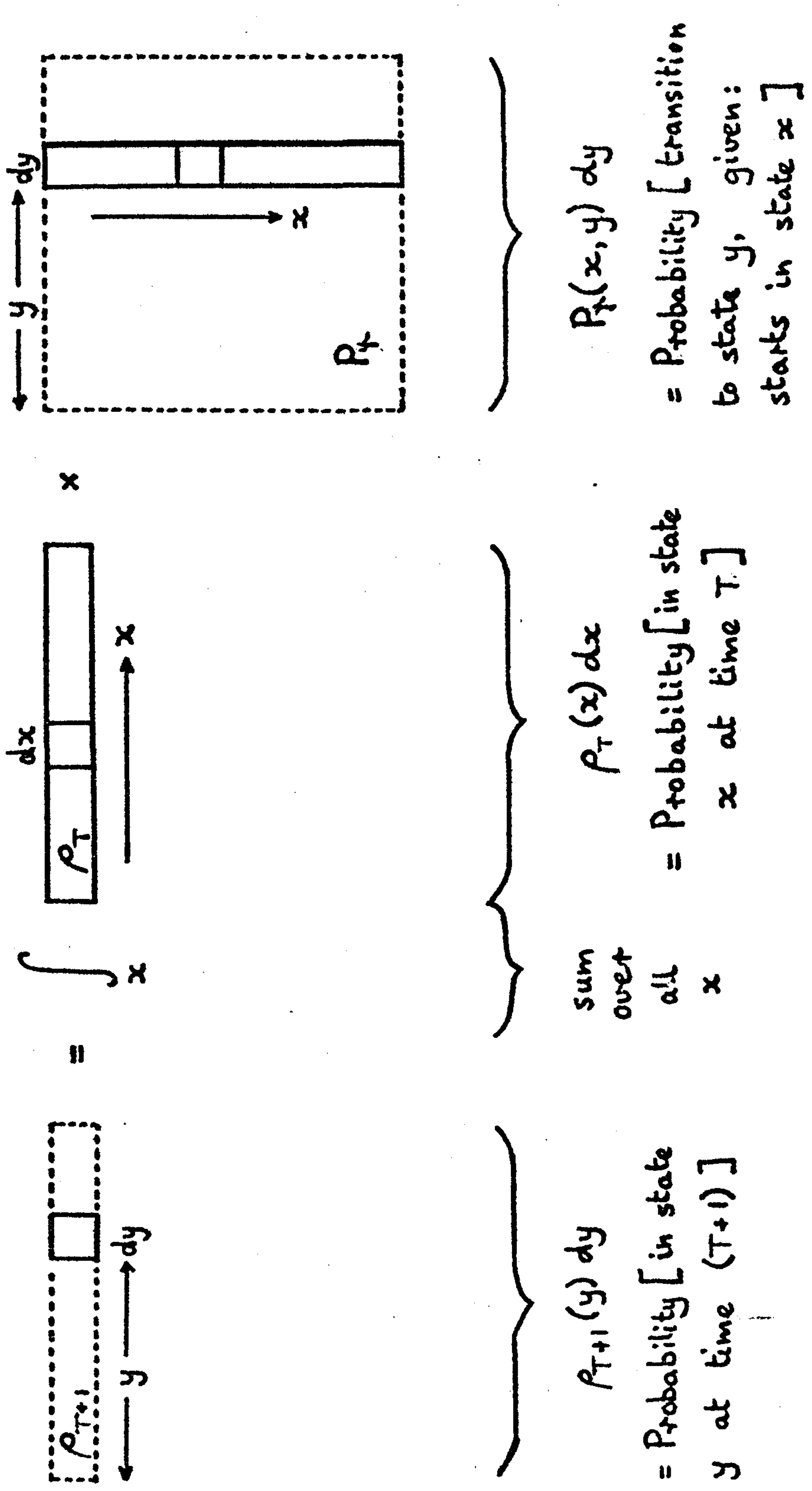

ρ_{T+1}
y
dy
=
∫ x
ρ_T
dx
x
×
P_r
y
dy
x
ρ_{T+1}(y) dy
= Probability [in state y at time (T+1)]
sum over all x
ρ_T(x) dx
= Probability [in state x at time T]
P_r(x,y) dy
= Probability [transition to state y, given: starts in state x]

Figure 7.4 Illustration of the transition equation 7.4, continuous memory

("Bird's eye view"): $\rho_{T+1}(y)dy = \int_{x=0}^{x=1} \rho_T(x)dx . P_r(x,y)dy$ 7.4

memorandum (1974a) to contrast it with the "worm's eye view". It is possible up to a point to write down the relations between state and transition probabilities corresponding to the finite model, as follows, and this possibility is illustrated by figure 7.4.

In the discrete transition matrix P, the value of the (i,j)-th element p_{ij} is the probability that, starting from state i, an arbitrary transition will be to state j. In the limiting case shown in figures 7.3, 7.4, if the model is in the state indexed by x, then the probability that an arbitrary transition will be to state y is $P_r(x,y)dy$. That is, P_r is a probability density function when viewed along a line of constant starting state x.

In place of the discrete stochastic condition which says that for any starting state i the model must make a transition somewhere with probability 1,

$$\sum_j p_{ij} = 1 \text{ for any } i,$$

there corresponds:

$$\int_0^1 P_r(x,y)\,dy = 1, \qquad \text{any } 0 \leq x \leq 1. \qquad \ldots\ldots 7.3$$

Similarly, in place of the transition equation (the "equation of motion") of the system

$$\pi_j(T+1) = \sum_i \pi_i(T)p_{ij} \qquad \text{for all } j$$

or in matrix notation,

$$\underline{\pi}(T+1) = \underline{\pi}(T) \cdot P$$

there corresponds (see figure 7.4):

$$\rho_{T+1}(y)dy = \int_{x=0}^{x=1} \rho_T(x)dx.P_r(x,y)dy \qquad \text{.... 7.4}$$

or $$\rho_{T+1}(y) = \int_0^1 \rho_T(x)P_r(x,y)dx$$

where $\rho_T(x)$, $\rho_{T+1}(x)$ are the probability density functions at times T, T+1 defined on $0 \leq x \leq 1$:

$$\rho_T(x)dx = \text{Probability[model is in state } x \text{ at time } T] \qquad \text{.... 7.5}$$

Thus, the limiting probability density function $\rho(x)$ of the states of the model as $T \to \infty$ satisfies

$$\rho(y) = \int_0^1 \rho(x)P_r(x,y)dx \qquad \text{.... 7.6}$$

which is a linear homogeneous integral equation of the second kind, as discussed in for example Smithies (1958).

Where this approach has so far failed, is not in the specification of a single state as a set of intervals in [0,1] but in making the correspondence between the set of all possible states and the same real interval. The progression of figures 7.2(a,b,c) to an apparent limit appears to be deceptive. There does not seem to be any simple way of making a correspondence which would allow a real value x in [0,1] to represent such a state, which would include all the possible states as x varies between 0 and 1. In other words, the set [0,1] does not seem to be a suitable index set for the collection of all possible configurations of disjoint intervals of [0,1]. It is possible that some alternative index set may be found to overcome this difficulty and allow such a one-to-one correspondence to be made, but this does not seem very likely at the moment. There does not seem to be a way in which the theory of real-valued functions suggested by equation 7.6 could be brought to bear on the problem without such a representation of all the possible configurations.

7.4 Other ways of specifying time behaviour, and consequent Markov theory

There are at least two ways in which the passage of time in the storage allocation system can be modelled differently from the conditions assumed in chapter 3. The constant time-independent behaviour which results from the assumed negative exponential distribution of inter-event time, can be relaxed and generalised if more information is available. The model will not in general lose its memoryless Markov property if this is done, but still remains a semi-Markov chain. Transitions occur as before but the time between them becomes dependent on the particular states and transitions. A useful application of this, and a good reason therefore for considering it, is to consider a "multiple processing" storage allocation system, instead of the processor sharing model assumed in chapter 3. As for any semi-Markov chain, the behaviour of this multiple processing model can be derived from the underlying full Markov chain (processor sharing) model with the extra knowledge of how the inter-event time is specified. This of course is one reason why the processor sharing model should be examined in detail first.

A second approach is to specify that events (block deallocations) occur randomly at given rates in real time, more likely events having greater rates of occurrence, and then deriving a matrix of transition rates between states from this, to govern the behaviour of the model. This leads to the storage allocation system being modelled as a continuous time Markov chain. This is again equivalent to the discrete time formulation adopted in the earlier chapters in which events are merely numbered successively as they occur, the main difference being that the time between successive events is no longer derived separately from the statement (in the transition matrix P) of the relative probabilities of alternative transitions that may occur in any state.

Each of these two alternatives, semi-Markov chain and continuous time Markov chain, are considered in turn.

7.4.1 Generalisation to semi-Markov chains; multiple processing model

One of the original papers which introduced and described semi-Markov chains is that of W. L. Smith (1955). Transitions occur according to a transition probability matrix just as in a full Markov chain, but the time between events or transitions is an arbitrary random variable which in general depends upon which state the model is in and which transition it makes from that state. By ignoring the variability of time, a semi-Markov chain becomes an ordinary (full) Markov chain; suppose its transition probability matrix is $P = (p_{ij})$ as before, with steady state probability vector $\underline{\pi} = (\pi_i)$. When the state- and transition-dependent behaviour is introduced, the steady state probability vector is modified and becomes (see Smith (1955) section 4.2, page 20):

$$\underline{\pi}' = (\pi_i'\ ) , \quad \text{where } \pi_i' = \pi_i t_i / (\sum_{j=1}^{S} \pi_j t_j) \qquad \text{.... 7.7}$$

where t_i is the average time, or holding time, spent in state i before a transition occurs.

Equation 7.7 expresses in precise terms, what one might expect: that in equilibrium the model is more or less likely to be found at random in a particular state than in the discrete time model, according to whether its holding time is relatively longer or shorter, compared with the other holding times.

An example where this idea is useful, is if the condition of processor sharing assumed in chapter 3 is altered. In processor sharing, each allocated block in memory "ages", or is serviced, at a rate inversely proportional to the total number of allocated blocks, and so the expected time to the next deallocation consequently remains constant however many blocks there are. Contrasting with this, a different condition would be that all the allocated blocks in memory continue to age each at the same rate irrespective of however many there are, so that when there are more blocks, transitions occur more quickly, and conversely. This would correspond to a multiple processing system, in which each allocated block has its own separate process or processor. In this case the holding time of any particular state is inversely proportional to the number of allocated blocks. To solve such a system, one could as before first apply the processor-sharing theory of the full Markov chain and then use the weighting of equation 7.7 to get the steady state probabilities in the semi-Markov multiple processor case. Equation 7.7 extends to the fully general case where the average conditional holding times (t_{ij}), or the average time spent in state i before a transition to state j occurs given that this transition will occur, can all be different. The unconditional holding times (t_i) are first obtained from the (t_{ij}):

$$t_i = \sum_{j=1}^{S} p_{ij} t_{ij} \qquad \text{.... 7.8}$$

and the (t_i) can then be used in equation 7.7.

7.4.2 Equivalent continuous time Markov chain formulation

Another way of considering the time behaviour of the storage allocation system is to model it as a continuous time Markov chain, still with a finite discrete set of states. This is really only an alternative viewpoint as there is no fundamental change to the operation of the model. Elapsed time is measured by the continuous real variable t. The definition of states of the chain as configurations of the memory, is unaltered so that there are still $S = f_{2N}$ states altogether including the empty state, but instead of an $S' \times S'$ matrix of transition probabilities $P = (p_{ij})$ there is now an $S' \times S'$ matrix of constant transition rates $R = (r_{ij})$ where

$$r_{ij} = \begin{bmatrix}\text{rate of transition from state i to state j,} \\ \text{given that the system is in state i}\end{bmatrix} \quad ,i\neq j \qquad \text{.... 7.9}$$

that is, for $i\neq j$,

$$r_{ij}dt = \text{Probability}\begin{bmatrix}\text{transition to state j will occur} \\ \text{during } (t,t+dt)\text{, given that the} \\ \text{model is in state i at time t}\end{bmatrix} \qquad \text{.... 7.10}$$

If $pr_i(t)$ is defined as the probability that the model is in state i at time t, then it follows that, for $i = 1,\ldots,S'$,

$$pr_i(t+dt) = pr_i(t)(1-\sum_{j\neq i}^{S'} r_{ij}dt) + \sum_{j\neq i}^{S'} r_{ji}pr_j(t) + o(dt) \qquad \text{.... 7.11}$$

If r_{ii} is defined:

$$r_{ii} = -\sum_{j\neq i}^{S'} r_{ij} \quad , \quad i = 1,\ldots,S' \;, \qquad \text{.... 7.12}$$

then 7.11 reduces to

$$pr_i(t+dt) = pr_i(t) + (\sum_{j=1}^{S'} r_{ji}pr_j(t))dt + o(dt) \quad \text{, or} \qquad \text{.... 7.13}$$

$$\frac{d}{dt}(pr_i(t)) = \sum_{j=1}^{S'} r_{ji} pr_j(t) \quad \text{for } i = 1,\dots,S' \qquad \dots. \ 7.14$$

which can be expressed as a matrix multiplication:

$$\underline{pr}'(t) = \underline{pr}(t)\, R \qquad \dots. \ 7.15$$

where $\underline{pr}(t)$ is the row vector $\underline{pr}(t) = (pr_i(t))$ and $\underline{pr}'(t)$ is its derivative with respect to t. If equilibrium is approached in the limit as t increases, then $\underline{pr}'(t) \rightarrow \underline{0}$ and so

$$\underline{pr}\ R = \underline{0} \qquad \dots. \ 7.16$$

where $\underline{pr}(t) \rightarrow \underline{pr}$ as t increases. $\underline{pr}$ is the stationary probability vector of the continuous time Markov chain, and it can be obtained by solving equation 7.16.

To obtain the transition rates $R = (r_{ij})$, it is necessary only to decide on the rates at which the deallocations of individual blocks can occur in any given state. For processor sharing, the rate at which any single block in a given state is likely to be deallocated is as before inversely proportional to the number of blocks in memory in that state. For multiple processing, the rate of service or deallocation of any block is constant for all blocks in all states. Having determined these deallocation rates, the rates of transitions (r_{ij}) are then fixed, as when a deallocation occurs the subsequent and immediate chain of allocations from the queue takes place in the same way as in the discrete time model. Given that a particular deallocation occurring at a rate x will lead if it does occur from state i to state j, $i \neq j$, with probability y as determined by the distributions of blocks and gaps in state i and the allocation algorithm, then the contribution to r_{ij} is (xy).

To see how the continuous time Markov chain is related to the discrete time chain of earlier chapters, choose any arbitrary dt > 0. Then from equation 7.16,

$$\underline{pr}\,(Rdt) + \underline{pr} = \underline{pr} \quad \text{, so that} \quad \underline{pr}(Rdt + I) = \underline{pr} \qquad \text{.... 7.17}$$

In the processor sharing model let the (constant) rate at which deallocations are occurring in any state be r, so that for $i \neq j$, $r_{ij} = r/n$ if there are n blocks in state i. Then if dt is chosen so that rdt = 1 the matrix (Rdt + I) is just the transition probability matrix P of the discrete time chain, and the equilibrium vector $\underline{pr}$ must therefore be $\underline{\pi}$. Equation 7.17 then becomes just the steady state equation 3.11, $\underline{\pi} = \underline{\pi}P$, and the continuous time Markov formulation is equivalent to the discrete time model already considered.

7.5 Other rules for allocating space in the storage allocation model

There are a number of ways in which the allocation and management of storage could be modelled differently from the scheme described in chapter 3. This scheme is simple in many respects and deliberately so, as the variations which are possible introduce extra complication which it was judged best to leave out at least to begin with. This complication usually means that extra information is required to be kept to decide the probabilities of transitions from one state to the next, and if there is no other way, this information has to be absorbed into the state specification causing the total number of states to go up and the relationships between individual states to become usually more complicated as well.

7.5.1 Removal of first queue request size, q, from state specification

A good example of how this extra information need not always be included as part of the specification of a state, but can be left out instead at the expense of some extra computation in working out the transitions among the consequently unexpanded set of states, is provided by an earlier version of the model which was considered and experimented with before the present version described in chapter 3. In this earlier version a state of the model, (or Markov chain as it is also possible to prove), is specified not only by fixing on the sizes and positions of the allocated blocks but also by giving the size q of the request waiting at the head of the queue. Unlike the present version, situations with exactly the same configuration of allocated blocks and gaps in the memory, but with a different value for q, are represented by different states in this earlier model. Obviously some states are inadmissible, except (as with the empty configuration in the present model) as intermediate stages occurring momentarily during a complete transition. They are precisely the cases where the queue request is less than or equal to the size of at least one gap in the memory. With a non-zero request distribution (r_n), $r_n > 0$ for all $n = 1,\ldots,N$, all the remaining states can be reached; this can be shown in the same way as the given proof in section 3.3.3 for the model that has been studied here. In general therefore, the number of possible states in this earlier model is larger than $S = f_{2N}$, and will be a significant fraction of the total number $N \times f_{2N}$ of all the (admissible and inadmissible) states. As shown in the preceding chapters, it was found possible to remove the first queue request size q from the state specification because the relative probabilities $(q_n^{(i)})$ of its possible sizes n in any given state i could be calculated from the request distribution (r_n) (section 3.2.4), and these were sufficient to allow the transition probabilities $P = (p_{ij})$ to be calculated. This exclusion of q however introduced the extra complication of the Q_n matrices (section 5.1.5) into the composition of the transition probability matrix P.

7.5.2 Delayed collapsing of free gaps, and garbage collection

Usually, any extra rules or information describing how the model works can only obviously be managed by making the state definition more precise, thus increasing the total number of states. (A notable exception however is the buddy system, see the next section 7.5.3 below.) An example is provided by the possibility of changing the collapsing rule which in the present model requires that a gap created by the deallocation of a block is immediately merged with its neighbours. One alternative way that collapsing is sometimes implemented is to only do garbage collection, as it is known, as a special operation when a request is tried which cannot be allocated entirely within any of the existing gaps as defined in the free storage list maintained by the allocation routine. When this happens, the special garbage collection process merges any adjacent free gaps to see if the waiting request can then be allocated into any of the larger gaps so formed. The idea is to save time by doing the extra housekeeping of collapsing less often. However, it is known from simulation results such as those of Nielsen (1977) for example, that as might be expected the storage utilisation of such a scheme is lower and memory becomes more fragmented than in a scheme which automatically merges adjacent free gaps together.

7.5.3 Buddy systems; no extra state information needed

Another well known example of a class of storage allocation systems which delay the collapsing of adjacent free blocks are the various buddy systems, outlined in chapter 2 section 2.2. Adjacent free gaps will only be combined if they and all their buddies are free, and in systems of the buddy type it is perfectly possible for there to be adjacent free gaps which can not be combined together. This apparently extra information is in fact redundant as it can be constructed from a knowledge of the allocated blocks' positions and sizes. The model as defined in chapter 3 and its algebraic representation in chapter 5 is therefore sufficient for buddy-type systems, needing only a set of correctly defined allocation matrices A_n.

As an example, figure 7.5 shows the allocation incidence matrix for N = 4, for the original binary buddy system of Knowlton (1965) (compare figure 5.7, the generalised allocation incidence matrix for N=4). Many of the total $S = f_{2N}$ configurations will be inadmissible in a buddy-type system, all those containing blocks which straddle a boundary between buddies without completely occupying both of them, and the allocation matrix will avoid these states which therefore can never occur. In figure 7.5 for example, states 6, 7, 8, 19, 20, 27, 28, and 29 cannot occur in the binary buddy system because they all contain blocks which lie across the boundary between the buddies formed by words (1,2) and (3,4) without occupying all of both of them, as in state 21 which is a possible state. There will also be an implementation difference in the choice of which of alternative equal sized buddies should be used to allocate a request. The model of chapter 3 cannot imitate the usual implementation which uses whichever is currently the first in a last in, first out (that is, a stack) list. Since all buddy systems organise the memory into a tree-like structure so that there can be no interaction between blocks

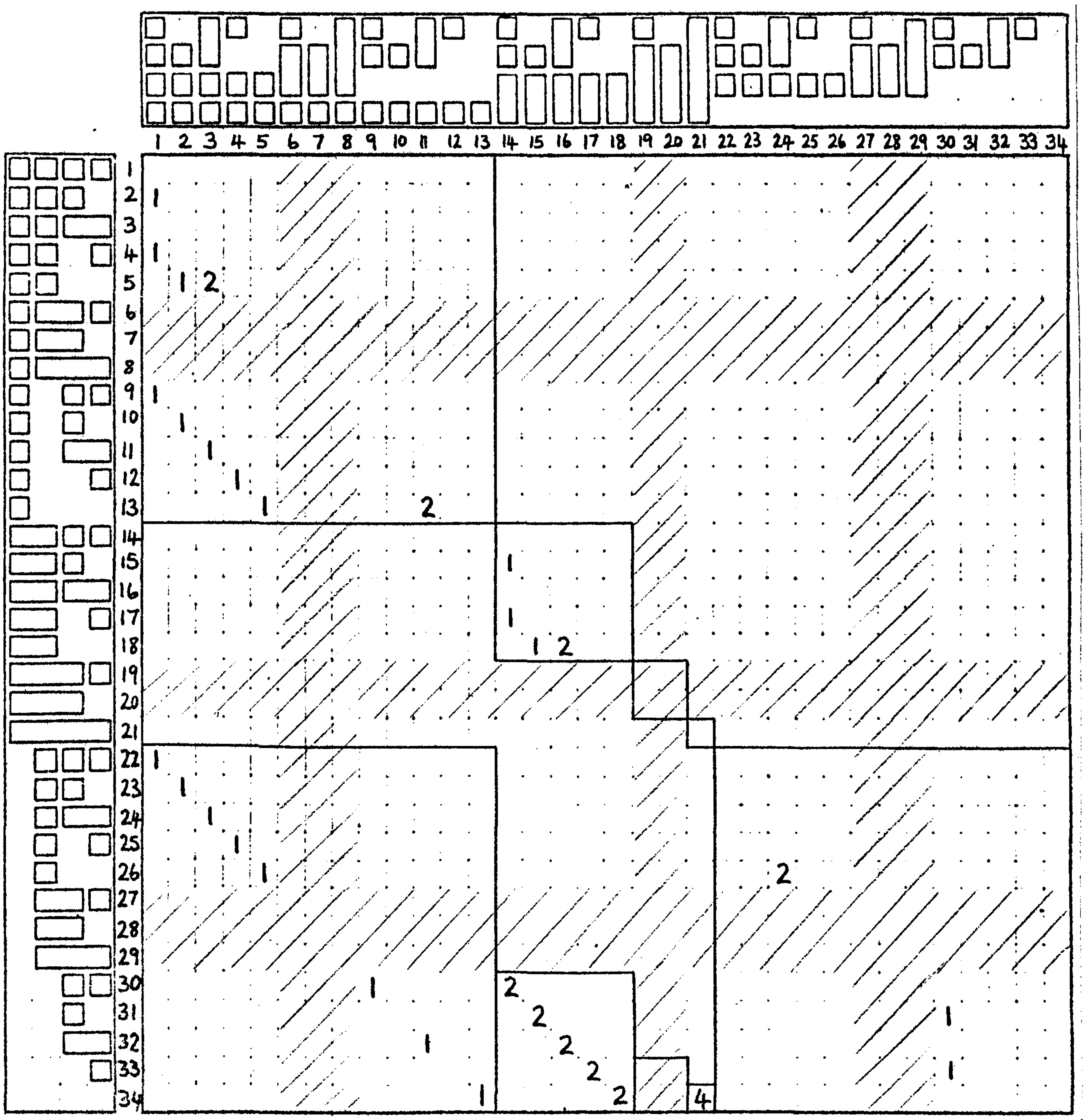


Figure 7.5 Allocation incidence matrix for the binary buddy system, N=4

Compare this figure with figure 5.7. All possible allocations in the binary buddy system are shown. The entries are not probabilities, but indicate where non-zero allocation transition probabilities may occur. Each entry (1,2,3 or 4) is the size of request for the corresponding allocation.

contained in separate buddies until those buddies are recombined, this will make no difference to such measures of storage utilisation or fragmentation as for instance the counts of how many blocks there are of a given size, but only to where the blocks actually happen to be. Because so many of the configurations of the general model are inadmissible in this well defined way, it may be possible to simplify the set of states to some advantage and therefore the algebra to automatically exclude them when analysing a buddy type of system.

7.5.4 Circular instead of linear storage

A device sometimes adopted to simplify analysis by avoiding end effects, and also because computer addressing modulo arithmetic can allow its implementation (especially perhaps in a virtual memory environment), is to assume that the memory is circular so that if the words of memory are numbered in order from 1 to N, then word 1 follows word N without any boundary. A good example of an analysis where this is assumed is that of Reeves (1979, 1980). It should be noted in passing that the end effects so avoided can be claimed to be both important and interesting, see for example Page (1959) who however was studying a related problem in which blocks were only put into storage, and not (at least during the first stage of his problem) subsequently taken out again.

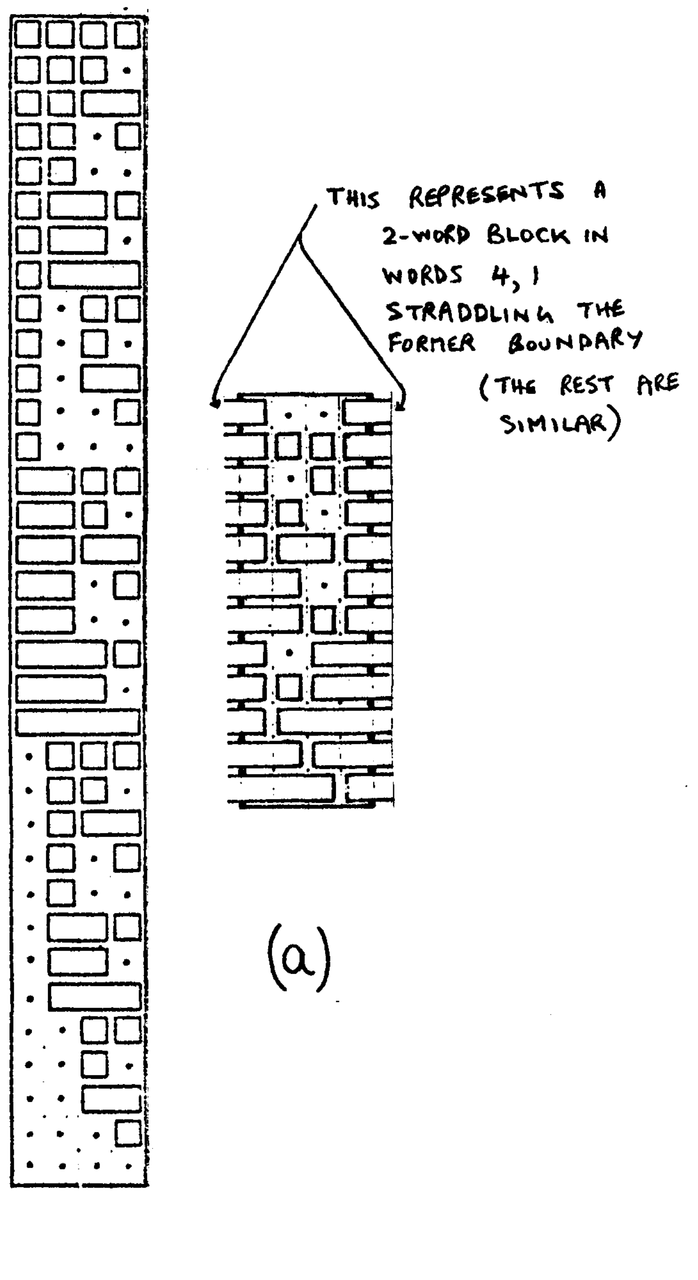
In the present model, the set of states or memory configurations must be extended to include all the extra cases in which blocks are allowed to straddle the former (N,1) boundary of the linear memory. The reorganisation of this expanded set of states into some sort of order corresponding to that defined in section 4.3 for example, seems to be tantalisingly difficult.

Ideally, such a representation of the states should avoid repeating states which are merely rotations of each other. In a circular memory such states are equivalent for the purpose of allocating a new request and modelling the subsequent behaviour, for all allocation algorithms except those which take account of and use a fixed numbering of the words of memory, for instance those which have an arbitrary fixed starting point.

Figure 7.6(a) shows the $S = 34$ possible configurations of an $N = 4$ word memory, together with the extra 12 cases introduced by circularity in no very good order, making 46 altogether. In part (b) of this figure the 32 of these states which are merely rotations of an already listed state have been excluded to leave only 14. Figure 7.7 shows the deallocation and generalised allocation incidence matrices for this reduced set of states when rotations are considered equivalent to each other (compare figures 5.1, 5.7).

7.5.5 Unsaturated request queue, and resulting modifications to the algebra

A very interesting change which can be made to the model as defined in chapter 3, is to modify the arrival rate of requests in the queue relative to the service rate as represented by their allocation in the memory so that the queue is no longer saturated and may become empty. If this is done, then the behaviour of an allocation scheme with an unsaturated request queue can be investigated, and in some respects this turns out to be easier to do. Most previous analyses of storage fragmentation have assumed a completely unsaturated request queue, requests being serviced (allocated) as soon as they arrive. Although storage utilisation is of great interest in a situation where the queue of requests is saturated or nearly so and the throughput is desired to be as high as possible, it is still relevant to enquire about the


THIS REPRESENTS A 2-WORD BLOCK IN WORDS 4, 1 STRADDLING THE FORMER BOUNDARY
(THE REST ARE SIMILAR)
(a)

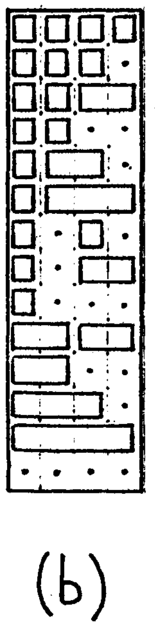
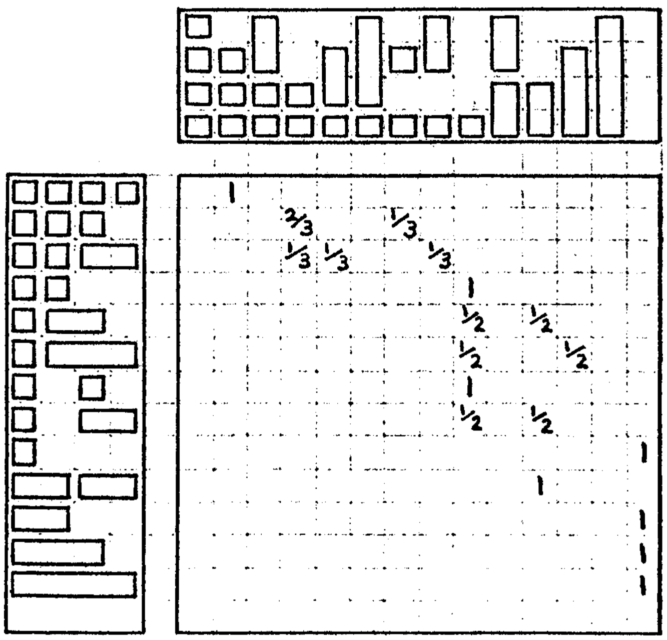


Figure 7.6 Extra states included for a circular memory, N=4

(a) The original 34 and the extra 12, (in no very good order).
(b) 32 of the 46 can be excluded if rotations are considered equivalent.

(a): DEALLOCATION



(b): ALLOCATION INCIDENCE

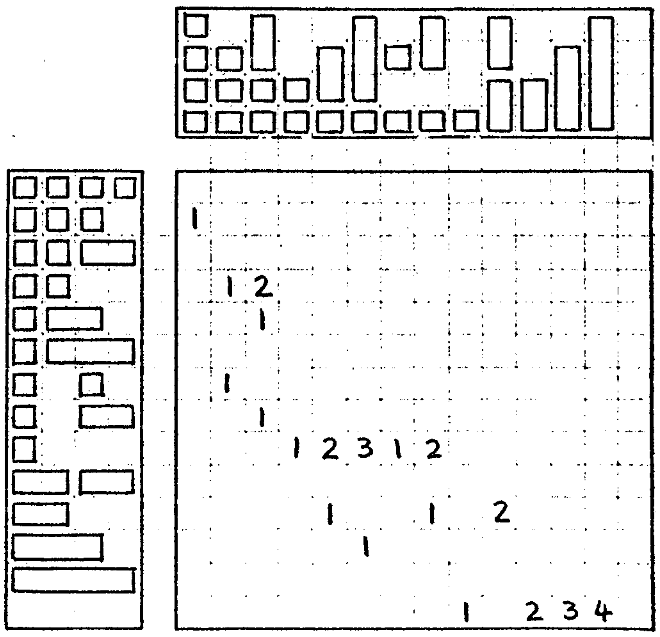


Figure 7.7 Deallocation and allocation incidence matrices, circular memory

Memory size N=4. States which are rotations are considered equivalent, see figure 7.6. Compare figures 5.1, 5.7 for the linear memory matrices.

behaviour of a storage allocation scheme in which each request can be serviced or allocated as soon as it arrives, for instance so as to know how much memory is needed in order for this situation to hold.

If the request queue is to be unsaturated, simple queueing theory (as well as common sense) dictates that its average rate of arrivals must not exceed the average service rate, in other words that the traffic intensity

$$\rho = \text{arrival rate / service rate}$$

must be no greater than 1. Once again, a simple first approach is to assume that processor sharing is in force, so that the single server (the combination of processor and memory) works at a constant rate independent of the number of allocated blocks as long as there is at least one, and to assume that both the service and arrival times are independent random variables with negative exponential distributions, so that the storage allocation scheme becomes an M/M/1 queue.

7.5.5.1 Comparison of the empty queue, finite queue and infinite queue models

As a first simplified approximation to this unsaturated request queue model, if the traffic intensity ρ is not too close to 1 and the memory size N is reasonably large compared with the average request size, then the possibility of overflow (no room for a request in memory) can be ignored. One way to realise this, is just to discard those requests which arrive and cannot be immediately allocated due to insufficient available space. A queue of requests is then never allowed to form. It should be stressed again that this empty queue simplification is to allow a first attempt at this unsaturated model to concentrate on the memory space allocation and fragmentation behaviour alone, although of course the empty (or immediately served) queue model is the one most often used in storage allocation studies. The action of the storage allocation model is different when the queue is empty, when it contains a finite number of requests, and when it can be assumed infinite, and the behaviour of the queue varying between these cases can be added after the empty queue model has been studied.

The empty queue model differs in three ways from the saturated or infinite queue model. First, the probability distribution of the size q of the request at the head of the queue is no longer conditional on the maximum size of gap in memory, and is in fact unchanged from the original request distribution (r_n). This is a considerable simplification of the algebra since there are no Q-matrices. Second, events in the model are identified with deallocations as before, but also with allocations as well. Third, only one allocation is performed at each allocation event, instead of the chain of allocations which stop only when a request is reached which will not fit, and this also is a considerable simplification of the algebra in chapter 5.

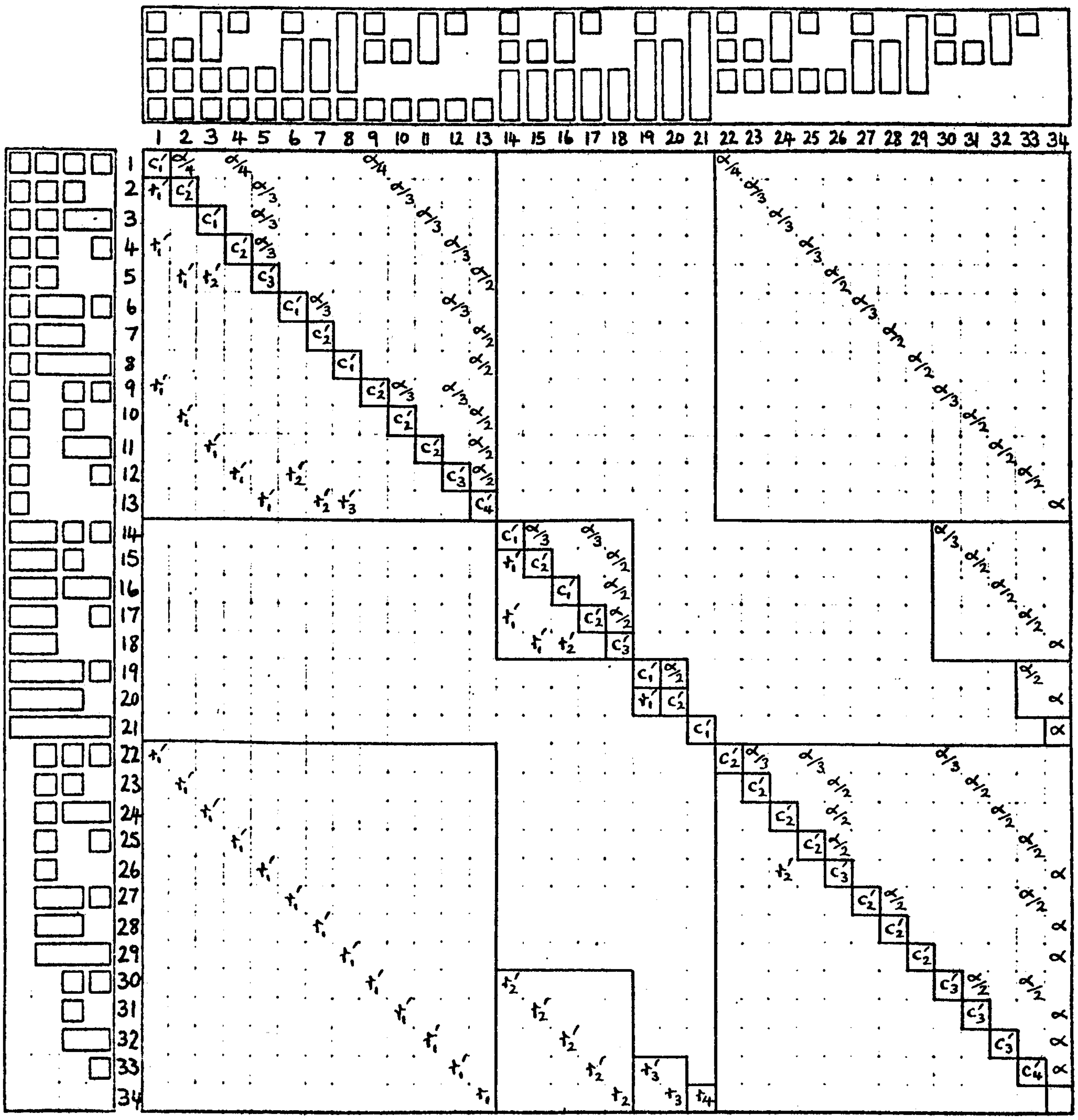
In the finite queue model, when the queue is not empty but cannot be assumed infinite, the waiting request queue size q will have the $q_n^{(i)}$ distribution (equation 3.8) as in the infinite queue model, and events only occur at deallocations, as in the infinite queue model, but after a deallocation, the chain of allocations has to stop if and when the existing queue has been exhausted. The general, varying queue length model is a combination of the empty and finite queue models, with events occurring both as new requests arrive at the queue and whenever a block is deallocated from the memory.

7.5.5.2 The empty queue model

The transition probability matrix of the empty queue model is much simpler than in the infinite queue model and it can be written down directly in terms of the basic allocation and deallocation matrices of chapter 5. For traffic intensity ρ, except when the memory is empty the probability that a randomly chosen event is a deallocation (service) event is $1/(1+\rho)$, and that it is an allocation (arrival) is $\rho/(1+\rho)$. With the allocation matrix A, deallocation matrix D and allocation termination matrix T as defined in chapter 5, (equations 5.9 and 5.1) the transition probability matrix P_e of the empty queue model can be expressed:

$$P_e = 1/(1+\rho) \times D \ + \ \rho/(1+\rho) \times (A+T) \qquad \ldots. \ 7.18$$

everywhere except for the row corresponding to the empty state. This last row of P_e is equal to the corresponding row of the allocation matrix A. Since D and T are zero in this row anyway, the only alteration necessary to make equation 7.18 completely correct is to replace the coefficient $\rho/(1+\rho)$ of A by the value unity for this row only. P_e is thus much easier to write down than its counterpart P is in the infinite queue case, as can be seen by comparison with equations 5.12 or 5.13. As an example, figure 7.8 shows P_e for N = 4, with the first fit allocation algorithm. The addition of the three matrices D, A, T in equation 7.18 is a simple affair as D is strictly upper triangular, A is strictly lower triangular and T is diagonal.



$$\alpha = \frac{1}{1+\rho} \; ; \qquad t_i' = \frac{\rho}{1+\rho}\, t_i \; , \qquad c_i' = \frac{\rho}{1+\rho}\, c_i \qquad i = 1,2,3,4 \; .$$

$(\rho = \text{TRAFFIC INTENSITY})$

Figure 7.8 Transition probability matrix P_e for the empty queue model, N=4

This matrix is easily formed according to equation 7.18 merely by superimposing D (upper triangle), A (lower triangle) and T (diagonal) with the appropriate coefficients; compare figures 5.1, 5.14 (first fit algorithm).

7.5.5.3 The finite queue model

The variation of this model's behaviour from the infinite queue model when there is an non-empty but finite queue is in one respect only, that after a deallocation a maximum of only as many allocations as there are requests in the queue is possible. The transition probability matrix P_f thus resembles P of the infinite queue model closely (compare equations 5.12, 5.13):

$$\begin{aligned} P_f &= \Sigma Q_n DT_n + (\Sigma Q_n DA_n)(T+A(T+\ldots+A(T+A)\ldots)) \quad \text{(L-1 factors)} \\ &= \Sigma Q_n DT_n + (\Sigma Q_n DA_n)((I+A+A^2+\ldots+A^{L-2})T+A^{L-1}) \quad \ldots.\ 7.19 \end{aligned}$$

where the queue length is L requests. If L is not less than the memory size, $L \geq N$, then $P_f = P$.

7.5.5.4 Unsaturated model as a combination of empty and finite queue models

As already mentioned, the transition probability matrix of this model will have to combine the algebraic statements of P_e and P_f, equations 7.18 and 7.19 above, and also to keep track of changes of the number of elements in the queue. The latter may be possible separately, for example just by using the probability $(1-\rho)\rho^L$ of finding L requests in the M/M/1 queue. It will be interesting to develop both this model and especially that of the empty queue in their own right, to see if and how far the simpler algebra can be analysed and compared with the existing results of others in the published literature.

7.6 Summary of further analysis and extension of storage allocation models

The study and application of the storage allocation model so far developed can be continued and extended in a number of useful ways. More powerful numerical methods, perhaps those recently suggested by Stewart (1977) may allow the convergence of the computations described in chapter 6 to be accelerated. This in turn should allow computations for slightly larger memory sizes, although the exponentially increasing amount of information required in these calculations is still a limiting factor. There are reasonable prospects of being able to learn more of the model's structure by continuing the algebraic analysis presented in chapter 5. There is also still a possibility that a useful way of defining an analogous infinite model amenable to analysis may be found, either by extending memory size N to infinity in the discrete case or by "smearing" the discreteness to the real continuum. The model may be made more general, either in its time behaviour as a semi-Markov chain or stochastic renewal process, or else by introducing extra rules for allocating storage, for example to study systems in which the treatment of gaps is more general, such as those including garbage collection. Buddy systems can already be represented by the present model, although their study may be helped by rearranging the set of states or memory configurations into an order which more closely represents their natural interrelationships in any particular system. Models of circular memory are sometimes studied to avoid awkward end effects, and rearranging the states will be necessary in this case also. The extension of the model to the case where the queue of requests is not saturated but can become empty introduces alternative algebraic derivations of the transition probability matrix when the queue is empty or only finite, and these can be combined to produce a stochastic model for such an unsaturated queue.

A topical example of a useful way in which to extend and apply the analysis of the present model, would be to build an algebraic model corresponding to that studied by Reeves (1979, 1980). This has a circular storage (though this may not of course introduce significant differences for large memories, depending on the request distribution) and uses the empty queue described above, requests being satisfied immediately they arrive. It would be most interesting to compare the analysis of such a model with the results obtained by using generating functions, and it would not be surprising if at least one of the methods could benefit from results obtained by the other.

Chapter 8 : Conclusion

This thesis has presented a survey of the existing studies which have been made of the behaviour and performance of systems concerning the allocation and fragmentation of storage, and a method of analysing such an allocation scheme as a probability model. In the simplest cases, the model is a finite discrete-time Markov chain, and an example in which this is the case has been justified and studied. Quantities of interest such as the average amount of storage which is allocated to requests at any time, can be derived from the consequent vector of state probabilities. The transition probability matrix upon which these probabilities depend has been discovered to have a recursive structure. This arises from the simple recursive nature of certain more basic matrices of probabilities of which the transition matrix is composed. Advantage has been taken of this algebraic structure in a numerical implementation to calculate the state and eventually the steady state probabilities, of which the latter are important because they determine the average performance of the allocation scheme in equilibrium. This computation avoids storing the transition matrix and so uses less time and space. By this means, models with larger memories have been computed than would otherwise have been possible. Although the memory sizes are still small, the results can provide useful indications of the performance of larger but otherwise corresponding models. The algebraic composition of the transition matrix has also led to the discovery of an unexpected simplification of the steady state equations which determine the steady state probabilities. The dependence of these probabilities on the choice of allocation algorithm and the request distribution is expressed in a much more direct way in these simple equations than it is in the original version. The simplicity of the reduced form of these equations should make them amenable to further analysis in a number of ways besides those presented here, and some possible further uses and investigation of the algebraic structure which has been found, have been indicated.

It is conventional to make concluding remarks under the above heading, but in the present case the term is misleading as the most interesting work is yet to come. The value of the ideas and analysis which have been presented in the preceding chapters is mainly that they have begun the foundation of an analytical technique for studying dynamic storage allocation and fragmentation phenomena where, with few exceptions, none existed before. The benefits which lie ahead will come by further use of this technique on the wide range of storage allocation models to which it can be applied, to see where and how far the analysis will lead. The author is confident that the applications presented here represent only a beginning, and that there is still plenty of mileage left in the sturdy treads of this approach to the "storage fragmentation problem".

One of the above-mentioned exceptions is Reeves' generating function technique, the analytical approach of which is complementary to the present work. The general method presented here in this thesis is to describe the behaviour of a storage allocation model by means of a probability vector containing one element for each state and to rely on discovering and using the resulting discrete and recursive structure of the transition matrix to determine and compare the model's performance. By contrast, in Reeves' method other but similar probability variables (such as the probability in equilibrium that a randomly chosen gap or block is of a given size) which contain the information about what is going on in the model are considered and in fact they are made to be the coefficients in a generating function. The principle of this analytical technique then becomes simple enough; the allocation and deallocation actions of the model determine conditions which the generating functions so introduced must satisfy. The functions can then be determined from these and any similar conditions which arise from their definition, and the main purpose of the investigation is to extract properties of the model such as the performance statistics from the form of these functions. This simplicity of principle belies the considerable practical skill needed with this as with most theoretical model building, the ability to successfully compromise between what is analytically possible and the complexities of real life. The application of the algebraic technique developed here in the present work will bring plenty of opportunities to practise the art of bringing these two opposing factors together without losing the important features of any situation being considered.

References

Batson, A. P., Brundage, R. E. (1977):
Segment sizes and lifetimes in Algol 60 programs.
CACM 20,1 (January 1977), 36-44.

Batson, A. P., Ju, S.-M., Wood, D. C. (1970):
Measurements of segment size.
CACM 13,3 (March 1970), 155-159.

Bauer, F. L. (1958):
On modern matrix iteration processes of Bernoulli and Graeffe type.
JACM 5 (1958), 246-257.

Bays, C. (1977):
A comparison of next-fit, first-fit and best-fit.
CACM 20,3 (March 1977), 191-192.

Bellman, R. (1970):
Dynamic programming.
Princeton university press, Princeton N.J. (1957).
Markovian decision processes are formulated in chapter 11.

Bernal, J. D. (1959):
A geometrical approach to the structure of liquids.
Nature 183 (1959), 141-147.

Bernal, J. D. (1960):
Geometry of the structure of monatomic liquids.
Nature 185 (1960), 68-70.

Bernal, J. D., Mason, J. (1960a):
Co-ordination of randomly packed spheres.
Nature 188 (1960), 910-911.

Betteridge, T. (1971):
An analysis of relocating storage allocation fragmentation.
Newcastle University Computing Laboratory report MRM/10 (March 1971).

Betteridge, T. (1973):
An analytic storage allocation model.
Newcastle University Computing Laboratory technical report 43 (March 1973).

Betteridge, T. (1973a):
Structure of the transition probability matrix of the storage allocation problem.
Newcastle University Computing Laboratory report MRM/65 (October 1973).

Betteridge, T. (1974):
An analytic storage allocation model.
Acta Informatica 4, (1974), 101-122.

Betteridge, T. (1974a):
Infinite extensions of the storage allocation model.
Newcastle University Computing Laboratory report MRM/79 (December 1974).

Betteridge, T. (1977):
Towards exact solutions of the storage fragmentation problem.
Newcastle University Computing Laboratory report MRM/129
(September 1977).

Bodewig, E. (1956):
Matrix calculus.
North Holland, Amsterdam (1956).

Bromley, A. G. (1977):
Memory fragmentation in buddy methods for dynamic storage allocation.
University of Sydney Basser Department of Computer Science technical
report 121 (May 1977), (also submitted for publication to
Acta Informatica).

Bryan, G. E. (1967):
JOSS: 20,000 hours at a console - a statistical summary.
AFIPS Fall Joint Computer Conference Proceedings, volume 31 (1967),
769-777.

Burton, W. (1976):
A buddy system variation for disk storage allocation.
CACM 19,7 (July 1976), 416-417.

Campbell, J. A. (1971):
A note on an optimal-fit method for dynamic allocation of storage.
Computer Journal 14,1 (January 1971), 7-9.

Coffman, E. G. jr., Muntz, R. R., Trotter, H. (1970):
Waiting time distributions for processor-sharing systems.
JACM 17 (1970), 123-130.

Collins, G. O. jr. (1961):
Experience in automatic storage allocation.
CACM 4,10 (October 1961), 436-440.

Coxeter, H. S. M. (1958):
Close-packing and froth.
Illinois J. Math. 2 (1958), 746-758.

Cranston, B., Thomas, R. (1975):
A simplified recombination scheme for the Fibonacci buddy system.
CACM 18,6 (June 1975), 331-332.

Denning, P. J. (1970):
Virtual memory.
Computing Surveys 2,3 (September 1970), 153-189.

Domb, C. (1947):
The problem of random intervals on a line.
Proceedings of the Cambridge Philosophical Society 43 (1947), 329-341.

Dvoretzky,A., Robbins, H. (1964):
On the "parking problem".
Publication of the Mathematical Institute of the Hungarian Academy of Science 9 (1964), 209-226.

Dynkin, E. B., Yushkevich, A. A. (1969):
Markov processes: theorems and problems.
Plenum press, New York (1969).
Pages 87-98 describe the optimal stopping problem.

Feller, W. (1968):
Introduction to probability theory and its applications, volume 1.
Wiley, New York 3rd edition: (1968).

Fenton, J. S., Payne, D. W. (1974):
Dynamic storage allocation of arbitrary sized segments.
IFIP Congress 1974, Volume 2: Software. North Holland (1974), 344-348.

Gantmacher, F. R. (1959):
The theory of matrices, volume 2.
Chelsea publishing company, New York (1959).

Gelenbe, E. (1971):
The two-thirds rule for dynamic storage allocation under equilibrium.
Information Processing Letters 1,2 (July 1971), 59-60.

Gragg, W. B. (1972):
The Padé table and its relation to certain algorithms of numerical methods.
SIAM Review 14,1 (January 1972), 1-62.

Hinds, J. A. (1975):
An algorithm for locating adjacent storage blocks in the buddy system.
CACM 18,4 (April 1975), 221-222.

Hirschberg, D. S. (1973):
A class of dynamic memory allocation algorithms.
CACM 16,10 (October 1975), 615-618.

Howard, R. A. (1960):
Dynamic programming and Markov processes.
MIT Technology Press, and Wiley, New York (1960).

Isoda, S., Goto, E., Kimura, I. (1971):
An efficient bit-table technique for dynamic storage allocation of 2^N-word blocks.
CACM 14,9 (September 1971), 589-592.

Jennings, A., Stewart, W. J. (1975):
Simultaneous iteration for partial eigensolution of real matrices.
J. Inst. Math. Appl. 15 (1975), 351-361.

Kleinrock, L. (1976):
Queueing systems, volume 2: Computer applications.
Wiley, New York (1976).
Processor sharing is referred to on page 166.

Knowlton, K. C. (1965):
A programmer's description of LLLLLL, Bell Telephone Laboratories low-level list language.
Bell Telephone Laboratories, Inc., (March 1965).

Knowlton, K. C. (1965a):
A fast storage allocator.
CACM 8,10 (October 1965), 623-625.

Knuth, D. E. (1968,1973):
The art of computer programming, volume 1: Fundamental algorithms.
Addison Wesley, Reading Mass. (1968); 2nd edition: (1973).
Storage allocation is discussed in pages 435-455.

Krogdahl, S. (1973):
A dynamic storage allocation problem.
Information processing letters 2,4 (October 1973), 96-99.

Lehman, M. M., Rosenfeld, J. L. (1968):
Performance of a simulated multiprogramming system.
AFIPS Fall Joint Computer Conference 33 (1968), 1431-1442.

Lovell, A. C. B. (1968):
The story of Jodrell Bank.
Oxford University Press, London (1968), 139-140.

Maher, R. J. (1961):
Problems of storage allocation in a multiprocessor multiprogrammed system.
CACM 4,10 (October 1961), 421-422.

Mannion, D. (1964):
Random space-filling in one dimension.
Publication of the Mathematical Institute of the Hungarian Academy of Science 9 (1964), 143-154.

Margolin, B. H., Parmelee, R. P., Schatzoff, M. (1971):
Analysis of free storage algorithms.
IBM System Journal 10,4 (1971), 283-304.

Matzke, E. B. (1950):
In the twinkling of an eye.
Bulletin Torrey Botanical Club 77 (1950), 222-227.

Ney, P. E. (1962):
A random interval filling problem.
Annals of Math. Statistics 33 (1962), 702-718.

Nielsen, N. R. (1977):
Dynamic memory allocation in computer simulation.
CACM 20,11 (November 1977), 864-873.

Organick, E. I. (1973):
Computer system organization - the B5500/B6700 series.
Academic Press, New York (1973).

Page, E. S. (1959):
The distribution of vacancies on a line.
J. Royal Statistical Society B 21,2 (1959), 364-374.

Palasti, I. (1960):
On some random space filling problems.
Publication of the Mathematical Institute of the Hungarian Academy of Science 5 (1960), 353-360.

Peterson, J. L., Norman, T. A. (1977):
Buddy systems.
CACM 20,6 (June 1977), 421-431.

Purdom, P. W. jr., Stigler, S. M. (1970):
Statistical properties of the buddy system.
JACM 17,4 (October 1970), 683-697.

Purdom, P. W. jr., Stigler, S. M., Cheam, T.-O. (1971):
Statistical investigation of 3 storage allocation algorithms.
BIT 11 (1971), 187-195.

Randell, B. (1969):
A note on storage fragmentation and program segmentation.
CACM 12,7 (July 1969), 365-372.

Randell, B., Kuehner, C. J. (1968):
Dynamic storage allocation systems.
CACM 11,5 (May 1968), 297-306.

Reeves, C. M. (1979):
Free store distribution under random-fit allocation.
Computer Journal 22,4 (November 1979, to appear).

Reeves, C. M. (1980):
Free store distribution under random-fit allocation, part 2.
Computer Journal 23,2 (May 1980, to appear).

Renyi, A. (1958):
On a one-dimensional problem concerning random space-filling.
Publication of the Mathematical Institute of the Hungarian Academy of Science 3 (1958), 109-127.

Robbins, H. E. (1944):
On the measure of a random set.
Annals Math. Statistics 15 (1944), 70-74.

Robson, J. M. (1971):
An estimate of the store size necessary for dynamic storage allocation.
JACM 18 (1971), 416-423.

Robson, J. M. (1974):
Bounds for some functions concerning dynamic storage allocation.
JACM 21,3 (July 1974), 491-499.

Robson, J. M. (1977):
Worst case fragmentation of first fit and best fit storage allocation strategies.
Computer Journal 20,3 (August 1977), 242-244.

Russell, D. L. (1977):
Internal fragmentation in a class of buddy systems.
SIAM J. Computing 6,4 (December 1977), 607-621.

Scherr, A. (1967):
An analysis of time-shared computer systems.
MIT Technology Press (1967).

Scott, G. D. (1960):
Packing of spheres.
Nature 188 (1960), 908-909.

Scott, G. D. (1962):
Radial distribution of the random close packing of equal spheres.
Nature 194 (1962), 956-958.

Seneta, E. (1973):
Non-negative matrices.
Allen and Unwin, London (1973).

Shen, K. K., Peterson, J. L. (1974):
A weighted buddy method for dynamic storage allocation.
CACM 17,10 (October 1974), 558-562; Corrigendum, CACM 18,4 (April 1975), 202.

Shore, J. E. (1975):
On the external storage fragmentation produced by first fit and best fit allocation stragegies.
CACM 18,8 (August 1975), 433-440.

Shore, J. E. (1977):
Anomalous behaviour of the fifty-percent rule in dynamic memory allocation.
CACM 20,11 (November 1977), 812-820.

Smith, W. L. (1955):
Regenerative stochastic processes.
Proc. Royal Society A 232 (1955), 6-31.

Smithies, F. (1958):
Integral equations. (Cambridge tracts in mathematics and mathematical physics, no. 49).
Cambridge University Press (1958).

Solomon, H. (1965):
Random packing density.
Stanford University Department of Statistics technical report 105 (July 1965).

Stevens, W. L. (1939):
Solution to a geometrical problem in probability.
Annals of Eugenics, London 9 (1939), 315-320.

Stewart, W. J. (1977):
A new approach to the numerical analysis of Markovian models.
In: Computer performance (editors: K. M. Chandy, M. Reiser), North Holland (1977), 279-295.

Stewart, W. J. (1978):
A comparison of numerical techniques in Markov modelling.
CACM 21,2 (February 1978), 144-152.

Ting, D. W. (1975):
Some results of the space requirements of dynamic memory allocation algorithms.
Cornell University Department of Computer Science technical report 75-229 (February 1975).

Totschek, R. A. (1965):
An empirical investigation into the behaviour of the SDC timesharing system.
Report SP 2191, System Development Corporation, Santa Monica, California (1965), AD 622 003.

Votaw, D. F. jr. (1946):
The probability distribution of the measure of a random linear set.
Annals of Mathematical Statistics 17 (1946), 240-244.

Weinstock, C. B. (1976):
Dynamic storage allocation techniques.
Ph. D. Thesis, Carnegie-Mellon University (1976).

Wynn, P. (1961):
The epsilon algorithm and operational formulas of numerical analysis.
Math. Comp. 15 (1961), 151-158.