ZOR – Methods and Models of Operations Research (1993) 38:281–307

A Heuristic for Decomposing Traffic Matrices in TDMA Satellite Communication

GÜNTER ROTE AND ANDREAS VOGEL

Technische Universität Graz, Institut für Mathematik, Steyrergasse 30, 8010 Graz, Austria

Abstract: **A heuristic for decomposing traffic matrices in TDMA satellite communication.** With the time-division multiple access (TDMA) technique in satellite communication the problem arises to decompose a given $n \times n$ traffic matrix into a weighted sum of a small number of permutation matrices such that the sum of the weights becomes minimal. There are polynomial algorithms when the number of permutation matrices in a decomposition is allowed to be as large as n^2. When the number of matrices is restricted to n, the problem is NP-hard. In this paper we propose a heuristic based on a scaling technique which for each number of allowed matrices in the range from n to n^2 allows to give a performance guarantee with respect to the total weight of the solution. As a subroutine we use new heuristic methods for decomposing a matrix of small integers into as few matrices as possible without exceeding the lower bound on the total weight. Computational results indicate that the method might also be practical.

Key Words: Matrix decomposition problem, TDMA satellite communication, greedy heuristics, edge coloring, bottleneck assignment problem, voting systems, apportionment.

1 Introduction

1.1 Background and Description of the Problem

In satellite communication, one satellite can serve several radio stations on earth. In order to allow signals to be sent from each radio station to each other radio station, the TDMA (time division multiple access) technique is used. At any instant, the satellite is set to a fixed *switching mode*: All radio stations transmit and receive data simultaneously, and the switching mode determines for each radio station the radio station which receives the data which the former transmits. In mathematical terms, a switching mode is a one-to-one mapping on the set of radio stations, i.e., a permutation. The satellite time-multiplexes regu-

This work was supported by the Fonds zur Förderung der wissenschaftlichen Forschung, Project S32/01.

0340–9422/93/38:3/281–307 $2.50 © 1993 Physica-Verlag, Heidelberg

larly between different switching modes in short intervals, according to a fixed cyclic schedule.

The communication needs between the radio stations are given by a matrix $T = (t_{ij})$, the *traffic matrix*. t_{ij} is the amount of information per time unit that has to be transmitted from the i-th to the j-th radio station. More information on the technical background can for example be found in Burkard (1985). We consider the problem of setting up a schedule for the satellite, i.e., a sequence of switching modes and a duration for each switching mode. Formally, the *matrix decomposition problem* can be stated as follows:

> Given an $n \times n$ matrix $T = (t_{ij})$ with nonnegative entries, find a *decomposition* of T, i.e., a sequence of permutation matrices $P^1, P^2, \ldots, P^q$ and a sequence of nonnegative weights $l_1, l_2, \ldots, l_q$ such that

$$T \leq \sum_{k=1}^{q} l_k P^k \quad \text{(elementwise)} \; . \tag{1}$$

The total duration d of the decomposition is given by

$$d = \sum_{k=1}^{q} l_k \; .$$

The first goal in setting up a switching schedule is of course to keep the total duration as small as possible. On the other hand, every change of the switching mode incurs a certain overhead and loss of time. Therefore, the number of matrices, q, should not be too large. There is a trade-off between the two objectives, d and q.

1.2 Related Results

Inukai (1979) and Burkard (1985) have shown that the optimal total duration is equal to t^*, the maximum row or column sum of the traffic matrix, but in general, a time-optimal decomposition may require up to $n^2 - 2n + 2$ matrices, which is too large for practical purposes. Burkard (1985) has also given an algorithm which takes $O(n^4)$ steps and constructs a decomposition where the number q of matrices is at most $n^2 - 2n + 2$, or, if T is an integer matrix, at most t^*, whichever number is smaller.

It is clear that any decomposition must consist of at least n matrices, unless some entries in the traffic matrix are zero. Gopal and Wong (1985) and Rendl (1985) have shown that the problem of constructing a shortest decomposition

into at most n matrices is NP-complete. A closely related problem, which also tends to decompose the traffic matrix into few matrices, has been attacked by Ribeiro, Minoux, and Penna (1989). With a branch and bound procedure they could solve problems optimally for up to $n = 15$ cities, but their solution uses hours of CPU-time.

Thus, it makes sense to look for heuristics. The currently best heuristic for decomposing into n matrices is due to Balas and Landweer (1983). Decomposing into a number q of matrices which is slightly larger than n has also been considered, for example by Lewandowski, Liu, and Liu (1983), who decompose into $2n$ matrices (which are given in advance), cf. also Burkard (1985), section 4. A more extensive review of results concerning the matrix decomposition problem can be found in Burkard (1991).

1.3 Results and Overview of the Present Paper

In this paper (in section 2) we propose a simple and fast "scaling" heuristic for constructing a short schedule with a given upper bound Q on the number q of switching modes. (Thus we solve "problem 3" in the classification of Balas and Landweer (1983).) We can prove a relative error guarantee for the total duration d of the decomposition. The method is not applicable if $Q = n$ or Q exceeds n only slightly. When Q is somewhat larger than n (of the order $2n$ or $3n$), the error bound is still very crude, but it improves as the ratio of Q and n becomes larger.

As a subproblem, we address the problem of decomposing a matrix under the constraint that the lower bound t^* on the total duration has to be achieved; the number q of matrices remains as the objective function to be minimized. (This is "problem 1" in Balas and Landweer (1983).) The traffic matrices that we have in mind for this problem are matrices with small integer entries. Here we use two heuristics: one based on a bottleneck assignment problem and on matching techniques, and a more powerful one which solves maximum flow problems.

As a side issue, we mention that one other subproblem that we have to solve has some interesting connections with voting systems.

Section 3 describes an implementation of the proposed procedures and presents the results of numerical experiments measuring the actual behavior of our heuristics (as opposed to the worst-case error guarantee). We compare our algorithm to the heuristic of Balas and Landweer (1983) for decomposing into only n matrices.

The concluding section 4 discusses the merits of our heuristics and relates them to other algorithms from the literature.

A preliminary version of this paper was presented at the 1988 Annual GAMM Conference in Vienna, and an extended abstract was published in Rote (1989).

2 The Heuristic

Our algorithm is based on the simple idea of scaling the entries of the given traffic matrix and rounding them to small integers. A matrix with small integers will require a small number q of matrices for decomposition; theoretically, we will utilize the trivial upper bound t^* on the number q of required permutation matrices. (Recall that t^* is the maximum row or column sum of the traffic matrix.)

Globally, the algorithm runs as follows:

Input: A non-negative real $n \times n$ matrix T.

(a) *Choose* some "unit" $F > 0$.
(b) Round the entries of the matrix upwards to the next multiple of F:

$$\bar{t}_{ij} := F \cdot \left\lceil \frac{t_{ij}}{F} \right\rceil .$$

(c) Solve the matrix decomposition problem for the resulting matrix $\bar{T}$ (or equivalently, for the *integer* matrix $(u_{ij}) := (\lceil t_{ij}/F \rceil)$ obtained by dividing through F).
(d) The resulting decomposition can be adjusted downwards to compensate for the rounding up in step (b).

The quality of the solution produced depends first of all on the choice of F in step (a). The idea is to choose F so large that the matrix (u_{ij}) consists of small integers and only few permutation matrices are needed for its duration-optimal decomposition in step (c), and so small that the error incurred in the rounding in step (b) is not too large. By choosing F appropriately, we will be able to give a performance guarantee for the quality of the solution produced by the heuristic.

Step (c) is the heart of the algorithm. The principal goal of this step, a duration-optimal decomposition, is relatively easy to achieve, but we also want *few* matrices, since their number will be the number of matrices that the solution will have. We shall discuss three methods for carrying out this step.

2.1 *Method I – Simple and Fast: Edge Coloring*

Before giving more details, we will formulate a couple of lemmas about the heuristic as stated so far. The first lemma repeats the already mentioned bound

on q for matrices with small entries and gives a possible implementation of step (c).

Lemma 1: An integer $n \times n$ matrix U with maximum row and column sum u^* can be decomposed into $q = u^*$ permutation matrices in time $O(u^* n \log n)$.

Proof: We restrict the problem to decomposition into "unit" permutation matrices, i.e., we allow only weights $l_k = 1$ in (1). This problem is essentially an edge coloring problem for a bipartite multigraph with vertices r_i and c_j $(i = 1, \ldots, n)$, with u_{ij} parallel edges between r_i and c_j. An *edge coloring* of this graph is an assignment of colors to all edges such that no two edges of the same color share a common vertex. An edge coloring corresponds to a solution of our matrix decomposition problem: The edges of one color form a matching, and the corresponding $n \times n$ adjacency matrix can be (arbitrarily) filled to a complete permutation matrix to get formulation (1). The number of colors is the number q of permutation matrices.

It is well known that, in a bipartite (multi-)graph, the number of colors required (the chromatic index) equals the maximum degree, which is equal to u^* in our case. Cole and Hopcroft (1982) gave an algorithm to find an edge coloring with this minimum number of colors in time $O(E \log n)$, where E is the number of edges. They described their algorithm only for simple graphs, but it is straightforward to extend it to multigraphs. In this case E has to count the edges by their multiplicities. The upper bound nu^* for the total multiplicity of all edges yields the claimed time bound. ■

The bound $q \leq u^*$ of lemma 1 is tight if and only if $u^* \leq \left(\frac{n+1}{2}\right)^2$, as is proved in the appendix of Rote and Vogel (1990). However, the bound of the following theorem 1, which relies on this lemma, will be surpassed by theorem 4 anyway, and therefore this fact is not so important in the context of this paper.

The next lemma relates F and the quality of the solution.

Lemma 2: If F is chosen as the smallest value such that $u^* \leq M$, for some given value $M \geq n$, then the following relation holds between the maximum row and column sum Fu^* of the rounded-up matrix and the corresponding value t^* of the original matrix:

$$Fu^* \leq \frac{M}{M - n + 1} \cdot t^* \ .$$

Proof: As $u_{ij} = \lceil t_{ij}/F \rceil$, the following relation holds between t_{ij} and u_{ij}:

$$Fu_{ij} \geq t_{ij} \geq F(u_{ij} - 1) \ . \tag{2}$$

If we would decrease F by a small amount, the maximum row and column sum of U would jump above M. For definiteness, let us assume, w.l.o.g., that this new maximum would occur in row 1: Then the first row sum would jump from $r_1 := \sum_j u_{1j}$ to $r_1 + n_1 > M$, where n_1 is the number of elements in the first row with $t_{1j} = Fu_{1j}$. Using the right side of (2) for the remaining $n - n_1$ elements of row 1, we can write:

$$t^* \geq \sum_j t_{1j} \geq F \cdot \left(\sum_j u_{1j} - (n - n_1) \right) = (r_1 + n_1 - n)F \geq (M - n + 1)F \ .$$

So we get the claimed bound for the maximum row and column sum of $F \cdot U$:

$$u^*F \leq MF = \frac{M}{M - n + 1}(M - n + 1)F \leq \frac{M}{M - n + 1}t^* \ . \qquad \blacksquare$$

Theorem 1: An $n \times n$ matrix T can be decomposed into a weighted sum of no more than Q permutation matrices ($Q \geq n$) with a total duration that is within a factor of $Q/(Q - n + 1)$ of the value t^* that is obtainable without restriction on the number of matrices in the decomposition. The decomposition can be found in time $O(Qn \log n)$.

Proof: It is clear that u^* can be made $\leq n$ by choosing F larger than the largest matrix entry; as we make F smaller and smaller, u^* will increase. By choosing F as the smallest value which gives $u^* \leq Q$ (as in lemma 2 with $M = Q$) and running steps (b) and (c) we get the bound for the quality. Lemma 1 yields the bound for the time complexity of step (c). Step (b) is trivial, and the time bound for step (a) will follow from lemma 3 in the next section. (We do not use step (d) to achieve our performance guarantee.) $\blacksquare$

2.2 *How to Determine F: Proportional Voting Systems*

We want to find the smallest value F such that the maximum row or column sum of the matrix $\lceil t_{ij}/F \rceil$ is at most some given value M. We can do this by

looking at each row and column individually, finding the smallest F such that the sum of $\lceil t_{ij}/F \rceil$ in this row or column is at most M, and taking the maximum of all those F's. To be specific, let us look at the first row of T. We look for the smallest F such that

$$\sum_{j=1}^{n} \lceil t_{1j}/F \rceil \leq M \ .$$

This problem occurs in another application, namely in proportional voting systems and the theory of apportionment: Given n parties and a number t_{1j} of votes for each party, M seats of a parliament have to be distributed to the parties; or the M representatives in a parliament are to be allocated to n districts, where t_{1j} is now the number of inhabitants in each district. The above method of allocation is known as Huntington's method of the smallest divisor, cf. e.g. Woodall (1982): F is the quota for one seat, and $u_{1j} := \lceil t_{ij}/F \rceil$ representatives are allocated to the j-th party or district. (A related voting system, which is widely used in European countries is the method of d'Hondt or Jefferson method, which allocates $\lfloor t_{ij}/F \rfloor$ seats.)

Algorithmically, we proceed by setting up the following array:

$$t_{11}, t_{11}/2, t_{11}/3, t_{11}/4, \ldots$$

$$t_{12}, t_{12}/2, t_{12}/3, t_{12}/4, \ldots$$

$$\vdots$$

$$t_{1n}, t_{1n}/2, t_{1n}/3, t_{1n}/4, \ldots$$

If some element t_{1j} is 0, this element does not contribute anything; otherwise the following holds: If F is between the $(l-1)$-st and the l-th entry in row j (or equal to the l-th entry) then $\lceil t_{1j}/F \rceil = l$; in other words, $\lceil t_{1j}/F \rceil$ is one plus the number of elements in the row which are greater than F. Therefore,

$$\sum_{j=1}^{n} \lceil t_{1j}/F \rceil = n' + \text{the number of elements in the array which are} > F \ ,$$

where n' is the number of non-zero elements. Thus we are looking for the smallest F such that at most $M - n'$ elements are larger, i.e., F is the $(M - n' + 1)$-largest element in the array.

We can determine the k-largest element in an array with n sorted rows by a method due to Frederickson and Johnson (1982), which takes $O(n \cdot \max\{1, \log(k/n)\})$ time. If $k \geq n$ this complexity increases with k. But by

computing some simple bounds for F we can ensure that we need only find the k-largest element with $k \leq n$.

An upper bound on F is given as follows:

$$\bar{F} := \frac{r_1}{M - n + 1} ,$$

where $r_1 = \sum_{j=1}^{n} t_{1j}$, because

$$\sum_{j=1}^{n} \lceil t_{1j}/\bar{F} \rceil < \sum_{j=1}^{n} (t_{1j}/\bar{F} + 1) = r_1/\bar{F} + n = (M - n + 1) + n = M + 1 ,$$

and thus $\sum \lceil t_{1j}/\bar{F} \rceil \leq M$.

If we remove from each nonzero row j of the array the first $\lceil t_{1j}/\bar{F} \rceil - 1$ elements, we effectively remove all elements which are larger than $\bar{F}$, and their number is $\underline{M} := \sum_{j=1}^{n} \lceil t_{1j}/\bar{F} \rceil - n'$, which is between $M - 2n' + 1$ and $M - n'$. Thus F, which is the $(M - n' + 1)$-largest element in the original array, is the k-largest element in the reduced array, where $k = (M - n' + 1) - \underline{M}$ is between 1 and n.

The k-largest or the k-smallest element in an array with n sorted rows can be found in $O(n)$ time, for $k \leq n$, by the method of Frederickson and Johnson. Since we have to repeat the whole process $2n$ times (once for reach row and column) we get

Lemma 3: For any given value $M \geq n$, the smallest value F such that the maximum row or column sum of the matrix $\lceil t_{ij}/F \rceil$ is $\leq M$ can be found in $O(n^2)$ time. ■

A theoretically slower but simpler and more practical algorithm would select the k-largest element in $O(k \log n)$ time by putting the current element of each row in a priority queue and retrieving the elements of the array in sorted order. The resulting complexity of $O(n^2 \log n)$ would still by far be dominated by the time for step (c). As will be discussed in section 3, not even this level of sophisticated data structures was needed for our computational experiments.

2.3 *Method II – Greedy: Bottleneck Assignment*

The fast solution of step (c) by using edge coloring techniques yields a decomposition into u^* permutation matrices, but there is no way to adapt this algorithm to use fewer matrices. In practice, we would like to have an algorithm which uses

as few matrices as possible, because this would allow us to choose F smaller than indicated by the worst-case bound of theorem 1, yielding a larger value of u^* and thus losing less in the rounding-up of step (b).

We try to reduce q by the following greedy strategy: We select the *first* weighted permutation matrix $l_1 P^1$ in such a way that the maximum row and column sum of the remaining traffic matrix $\max(U - l_1 P^1, 0)$ is reduced by as much as possible. This will reduce the bound u^* on the number (and, hopefully, also the actual number) of further matrices which will be needed in the decomposition. We continue this strategy with the remaining matrix until we are done.

Let us now discuss how to determine P^1 and l_1. We denote the i-th row sum and the j-th column sum by r_i and c_j, respectively. For keeping the total duration of the decomposition within u^*, the maximum row and column sum of U must be reduced by l_1 when the matrix $\min(l_1 P^1, U)$ is subtracted. This can be formulated as follows:

$$\text{If } P^1_{ij} = 1 \text{ then } r_i - \min\{l_1, u_{ij}\} \leq u^* - l_1 \text{ and } c_j - \min\{l_1, u_{ij}\} \leq u^* - l_1 \ .$$

Since $r_i \leq u^*$ and $c_j \leq u^*$, this is equivalent to

$$\text{If } \quad P^1_{ij} = 1 \quad \text{then} \quad r_i - u_{ij} \leq u^* - l_1 \text{ and } c_j - u_{ij} \leq u^* - l_1 \ ,$$

or in other terms:

$$\text{If } \quad P^1_{ij} = 1 \quad \text{then} \quad l_1 \leq u_{ij} + (u^* - \max\{r_i, c_j\}) \ . \tag{3}$$

Let us interpret this formula. For *critical* rows (and columns), i.e., rows with $r_i = u^*$, l_1 must be $\leq u_{ij}$. Non-critical rows and columns have some *slack* $u^* - r_i$ or $u^* - c_j$, respectively, which allows to weaken this inequality for their elements: The smaller of the row slack and the column slack for each element can be added to the bound u_{ij}.

Now, for given l_1, a possible permutation matrix corresponds to a complete matching in a bipartite graph with $n + n$ nodes, whose edges are given by the above conditions. The maximum value of l_1 can be found by solving a bottleneck assignment problem whose cost matrix is given by the right side of (3).

After determining P_1 and l_1, $\min(l_1 P^1, U)$ is subtracted from U. P^2 and l_2 are determined by the same procedure for the remaining matrix, and so on.

Example 1: Consider the matrix

$$U = \begin{array}{r} \\ 14 \\ 15 \\ 16 \end{array} \begin{array}{c} \begin{array}{ccc} 16 & 15 & 14 \end{array} \\ \begin{bmatrix} 3 & 5 & 6 \\ 8 & 6 & 1 \\ 5 & 4 & 7 \end{bmatrix} \end{array}$$

We have $u^* = 16$, and the row and column sums are given beside the matrix. Now, let us try $l_1 = 7$, i.e., we want to reduce u^* to 9. We cannot take the element 5 of the first row, because then c_2 could only be reduced to 10. However, we are allowed to take the element 6 of the first row, although it is smaller than $l_1 = 7$: Both its row and its column have a slack of 2, and thus it would be sufficient to reduce this element by $l_1 - (u^* - \max\{r_1, c_3\}) = 5$, and $u_{13} \geq 5$. So we finally get the following pattern of allowed elements:

$$\begin{bmatrix} \cdot & \cdot & 6 \\ 8 & 6 & \cdot \\ \cdot & \cdot & 7 \end{bmatrix}$$

This matrix contains no permutation matrix. But if we try $l_1 = 6$, we get the following pattern of allowed elements:

$$\begin{bmatrix} \cdot & \underline{5} & 6 \\ \underline{8} & 6 & \cdot \\ \cdot & \cdot & \underline{7} \end{bmatrix},$$

in which the underlined elements form a matching. The matrix given by the right side of (3) is obtained by adding to each entry of U the minimum of the corresponding row slack and column slack:

$$\begin{bmatrix} 3 & 5 & 6 \\ 8 & 6 & 1 \\ 5 & 4 & 7 \end{bmatrix} + \begin{matrix} \\ 2 \\ 1 \\ 0 \end{matrix} \begin{matrix} \begin{matrix} 0 & 1 & 2 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{matrix} = \begin{bmatrix} 3 & 6 & 8 \\ 8 & 7 & 2 \\ 5 & 4 & 7 \end{bmatrix}$$

The row and column slacks are written adjacent to the second matrix, and each entry of that matrix is equal to the minimum of its respective slacks. The feasible entries for a given value of l_1 are just those entries which are $\geq l_1$ in the last matrix.

Now, as the largest possible value of l_1 is 6, and the first permutation matrix is as indicated above, the first step in the decomposition is as follows:

$$\begin{bmatrix} 3 & 5 & 6 \\ 8 & 6 & 1 \\ 5 & 4 & 7 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 \\ 6 & 0 & 0 \\ 0 & 0 & 6 \end{bmatrix} + \begin{matrix} \\ 9 \\ 9 \\ 10 \end{matrix} \begin{matrix} \begin{matrix} 10 & 10 & 8 \end{matrix} \\ \begin{bmatrix} 3 & 0 & 6 \\ 2 & 6 & 1 \\ 5 & 4 & 1 \end{bmatrix} \end{matrix}$$

Indeed, the maximum row and column sum of the last matrix has decreased by $l_1 = 6$. Computing again the cost matrix for the bottleneck assignment problem, we get

$$\begin{bmatrix} 3 & 0 & 6 \\ 2 & 6 & 1 \\ 5 & 4 & 1 \end{bmatrix} + \begin{matrix} \\ 1 \\ 1 \\ 0 \end{matrix} \begin{bmatrix} 0 & 0 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 0 & \underline{7} \\ 2 & \underline{6} & 2 \\ \underline{5} & 4 & 1 \end{bmatrix}$$

The bottleneck assignment (of value $l_2 = 5$) is indicated by the underlined elements. Thus, the second matrix can be subtracted:

$$\begin{bmatrix} 3 & 0 & 6 \\ 2 & 6 & 1 \\ 5 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 5 \\ 0 & 5 & 0 \\ 5 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 0 & 1 \\ 2 & 1 & 1 \\ 0 & 4 & 1 \end{bmatrix}$$

Continuing this process, we get

$$\begin{bmatrix} 3 & 0 & 1 \\ 2 & 1 & 1 \\ 0 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 2 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, we have $q = 5$ matrices with $(l_1, l_2, l_3, l_4, l_5) = (6, 5, 2, 2, 1)$. ●

Example 2: A typical sequence of l_k's, which arose in a randomly generated test example, a 10×10 matrix with $u^* = 51$, was as follows:

7 7 6 6 5 4 3 2 2 2 1 1 1 1 1 1 1

with $q = 17$ matrices. ●

The fastest known algorithm for the bottleneck assignment problem is due to Gabow and Tarjan (1988) and takes $O(n^{5/2}\sqrt{\log n})$ time. This has to be multiplied by the number q of matrices to get a total complexity of $O(Qn^{5/2}\sqrt{\log n})$ for method II.

The above example exhibits two properties of the sequence $l_1, l_2, \ldots, l_q$:

- The sequence is decreasing (in the order in which it is generated);
- There are no gaps between successive values, and near the end of the sequence, the small numbers occur repeatedly.

The first property is clear: If we had $l_k < l_{k+1}$ then, by exchanging $l_k P^k$ and $l_{k+1} P^{k+1}$ in the summation $\sum_{k=1}^{q} l_k P^k \geq U$, we see that we could as well have taken P^{k+1} before P^k, and thus l_k is not the true value of the k-th bottleneck assignment problem. The second property is not necessarily true but it is typical. It certainly depends on the size of the matrix elements. But these elements are scaled down to small integers in step (b) before we apply the decomposition; thus, even if gaps appear, they can be expected to be small.

These observations suggests a different approach: Test for successive values of l_k in decreasing order, starting with an upper bound $\bar{L}$ on l_1, e.g., the largest matrix entry. Each test amounts to finding a complete bipartite matching. For this purpose, we can use the procedure of Hopcroft and Karp (1973), which requires $O(n^{5/2})$ steps. After a successful test, we get a new solution matrix P^k, and, after reducing U, we can try the same value l_k again. If we don't find a complete matching, we reduce l_k by 1 and try again. In this case, we can take the maximum matching from the previous iteration as a starting solution.

Thus, there will be q successful tests and at most $\bar{L}$ unsuccessful tests. Thus, we get a total complexity of $O((q + \bar{L})n^{5/2})$. $\bar{L}$ can be bounded by u^*, but it is usually much smaller. We could get $\bar{L}$ by solving one initial bottleneck assignment problem to compute l_1. Thus we can state the complexity of method II:

Theorem 2: Method II can be carried out in $O((q + l_1)n^{5/2})$ *time or in* $O(q\sqrt{\log n} \cdot n^{5/2})$ *time.* ■

In our computational experiments we have additionally reduced l_k in each step, if necessary, in order to ensure that the graph defined by (3) has no isolated vertices. This was sufficient to eliminate most of the unsuccessful tests, see section 3.4. Therefore, it would not have paid off to use Gabow and Tarjan's method for the bottleneck assignment problem, and we have only used the Hopcroft and Karp algorithm in our implementation of method II.

It is interesting to compare our approach to the heuristic of Balas and Landweer (1983) (also proposed in Gopal and Wong (1985)), who also use the bottleneck assignment problem as a subroutine in their algorithm. However, they *minimize* the *largest* entry u_{ij} among the remaining entries, whereas we *maximize* the *smallest* entry of a modified matrix in each step. Thus, in a certain sense, our approach is opposite to that of Balas and Landweer: We find the largest part in the decomposition first, whereas they start with the smaller parts.

2.4 *Method III – Even Greedier: Maximum Network Flow*

The sequence of example 2 shows that many matrices P^k belong to a group of equal l_k values. By solving a maximum flow problem on a suitably defined graph,

we can try to determine all permutation matrices P^k which belong to a group of equal l_k's simultaneously. Instead of looking for *one* permutation P^k with a given value of l_k we try to find *as many as possible.* A sum of g matchings corresponds to a flow in a bipartite network with $n + n$ vertices where each row node has a supply of g units and each column node has a demand of g units.

In order to determine capacities we have to make the following considerations: An entry (i, j) which is used in a permutation P^k can "use up resources" of two kinds:

- It can reduce u_{ij} by l_k (in case $u_{ij} \geq l_k$).
- It can also reduce the slacks $u^* - r_i$ and $u^* - c_j$ of the row and column to which it belongs.

We have to model these two kinds of resources by two kinds of arcs (cf. figure 1). The network has two nodes for each row i: A regular source node r_i and a "slack" node $\bar{r}_i$. Similarly, there is a sink node c_j and a column slack node $\bar{c}_j$ for each column j. For each entry u_{ij} we have now two arcs: There is a "direct" arc from r_i to c_j of capacity $\lfloor u_{ij}/l_k \rfloor$. This capacity counts how often a permutation may use the entry u_{ij} in the first way, i.e., by reducing u_{ij}. In addition, there is a "pseudo-arc" from $\bar{r}_i$ to $\bar{c}_j$ of infinite capacity. Using this arc corresponds to using an entry in the second way, i.e., reducing the slacks $u^* - r_i$ and $u^* - c_j$ by l_k. This usage is restricted by the capacities of the "slack" arcs from r_i to $\bar{r}_i$ of capacity $\lfloor (u^* - r_i)/l_k \rfloor$ and from $\bar{c}_j$ to c_j of capacity $\lfloor (u^* - c_j)/l_k \rfloor$, which precede and follow the pseudo-arc.

It is easy to see that a flow in this network which satisfies a constant supply and demand of value g at each source and sink vertex, respectively, corresponds

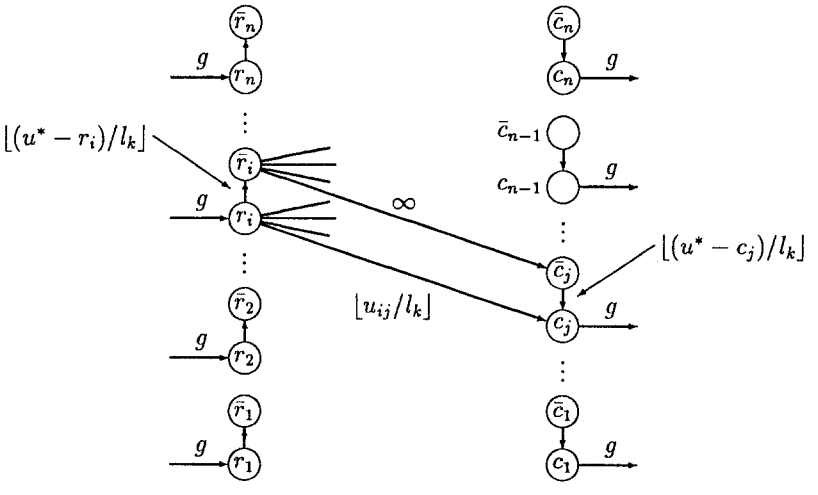

Fig. 1. The network for the maximum flow problem (initial version). From the middle level, only two representative arcs are shown.

to a set of g permutation matrices with weight l_k which can be continued to a duration-optimal solution. (Equation (3) would suggest to use a bipartite graph with $n + n$ nodes and a capacity of $\lfloor (u^* + u_{ij} - \max\{r_i, c_j\})/l_k \rfloor$ for each arc (i, j). An example showing that this would not lead to the correct result is contained in the report Rote and Vogel (1990).)

In addition to the two uses of an entry that have been discussed, there is also a "mixed" use of an entry (i, j) by a permutation P^k: When $l_k > u_{ij}$ but $u_{ij} \neq 0$, this reduces *both* u_{ij} *and* the slacks $u^* - r_i$ and $u^* - c_j$. However, we cannot model this in our network without getting fractional capacities. Thus, as we have defined the network so far, it is possible that there is not even a single matching (i.e., a flow with $g = 1$) in the network although one should exist according to criterion (3) which was used in method II.

Therefore we make the following modification:

> Whenever $u^* - r_i < l_k$, we set the capacity of the slack arc $(r_i, \bar{r}_i)$ to 1 instead of 0, but at the same time we eliminate all arcs out of $\bar{r}_i$ for which $u_{ij} \geq l_k$ or $u_{ij} + u^* - r_i < l_k$.

We do the same for all columns. In this way we have ensured that, if an entry u_{ij} should be usable by criterion (3), then there is a way to send at least one unit of flow from r_i to c_j: If $u_{ij} \geq l_k$ then we can use the direct arc; otherwise we can use the corresponding pseudo-arc. If the above modification is carried out for a node $\bar{r}_i$, only one of the arcs $(\bar{r}_i, \bar{c}_j)$ out of $\bar{r}_i$ can be used in a feasible flow, reducing the slack of row i and possibly the entry u_{ij}. There is no parallel direct arc (r_i, c_j) with positive capacity that would also decrease u_{ij}.

We can even allow some more arcs, as follows:

> When we set the capacity of the slack arc $(r_i, \bar{r}_i)$ to 1 instead of 0, we eliminate only those arcs out of $\bar{r}_i$ for which $(u_{ij} \bmod l_k) + u^* - r_i < l_k$, (and similarly for the columns).

This rule will clearly not eliminate more edges than the former rule. Here the remainder $(u_{ij} \bmod l_k)$, which cannot be "used up" by the flow on the direct edge (r_i, c_j) of capacity $\lfloor u_{ij}/l_k \rfloor$, is put together with the slack $u^* - r_i$ to see whether a total of l_k can be reached.

Let us now summarize method III to find permutation matrices of weight l_k into which we can decompose the matrix: We set up the network as described above and look for the largest value of g such that a flow satisfying all supplies and demands of value g exists. This value g is the number of permutation matrices that we get. To find these matrices, we have to decompose the flow that we have found into "permutation flows". When we add up the flow on each arc and its corresponding pseudo-arc and interpret this as the multiplicity of an edge (r_i, c_j) we get a bipartite multi-graph which is regular of degree g. This graph can be decomposed into g complete matchings, for example by the coloring techniques of Cole and Hopcroft (1982), which we used in method I.

If $g > 0$, we have to try the same value of l_k again, since we may have missed some "mixed usage" of a matrix entry. If we got $g = 0$, the next weight l_k must be smaller.

Example 2 (continued): For the same data as above, this improved algorithm yields only 15 matrices instead of 17, with the following weight sequence:

(7 7) (6) (6) (5) (4) (3 3) (2 2) (2) (1 1 1 1)

Here, each parenthesis groups together all l_k's which were obtained in one successful run of the maximum flow algorithm. There was one unsuccessful run, with $l_k = 4$. ●

One possible strategy to search for the maximum value of g is exponential search: Try the values $g = 1, 2, 4, 8, 16, \ldots$ until g is found to be too large, and continue with binary search. In section 3, we shall describe the method that we have implemented.

In theory, we may still be unlucky and find $g = 1$ or $g = 0$ at every step; thus, method III may yield no savings with respect to method II. For dense graphs (with $O(n^2)$ edges), computing a maximum flow takes $O(n^3)$ time in the worst case. In this time we can also afford to solve the bottleneck assignment problem to ensure that we never get $g = 0$; the number of flow problems is at most the number q of matrices, and we get the following upper time bound:

Theorem 3: Method III can be carried out in $O(qn^3)$ time. ■

The fastest implementation of method III would probably be a hybrid method. Initially, use the maximum matching algorithm of Hopcroft and Karp (1973). Before repeated values of l_k's can be expected to appear, switch to the maximum flow techniques with some kind of exponential search for the correct value of g.

2.5 *Theoretical Implications of Method III*

As a consequence of the maximum flow heuristic, we can improve the bound of theorem 1. The following lemma is a variation of lemma 1. It has the additional assumption that no matrix element is 0, but this assumption is only technical and is not needed in the final theorem.

Lemma 4: A positive integer $n \times n$ matrix U with maximum row and column sum u^* can be decomposed into at most $\lceil (u^* + n)/2 \rceil$ weighted permutation matrices.

Proof: First we "fill up" U by enlarging some entries so that all row and column sums become equal. This is always possible as the problem of finding the correct integer numbers to add to the entries of U is just a transportation problem.

Then we set up the network for the flow problem as described above, for the weight $l_1 = 2$. Since all slacks are zero, slack nodes and pseudo-arcs can be ignored. By the max-flow-min-cut theorem, a flow satisfying all supplies and demands exists if and only if the following cut condition is satisfied for all subsets X of nodes:

$$\text{total supply in } X - \text{total demand in } X \leq \text{capacity of the cut } (X, \overline{X}) \ . \tag{4}$$

Here, the cut $(X, \overline{X})$ consists of all arcs going from a node in X to a node not in X, and its capacity is the sum of the capacities of these arcs.

Now let $X = R \cup C$ be an arbitrary node set, where $R \subseteq \{r_1, r_2, \ldots, r_n\}$ and $C \subseteq \{c_1, c_2, \ldots, c_n\}$ are the row nodes and the column nodes of X. We denote their complements by $\overline{R} = \{r_1, r_2, \ldots, r_n\} \backslash R$ and $\overline{C} = \{c_1, c_2, \ldots, c_n\} \backslash C$. As the supply of each row node and the demand of each column node is g, (4) can be written as follows:

$$g|R| - g|C| \leq c(X, \overline{X}) := \sum_{r_i \in R} \sum_{c_j \in \overline{C}} \lfloor u_{ij}/2 \rfloor \ . \tag{4'}$$

The right side can be bounded as follows:

$$c(X, \overline{X}) = \sum_{R} \sum_{\overline{C}} \lfloor u_{ij}/2 \rfloor \geq \sum_{R} \sum_{\overline{C}} \frac{u_{ij} - 1}{2} = \frac{1}{2} \cdot \left\{ \sum_{R} \sum_{\overline{C}} u_{ij} - |R| \cdot (n - |C|) \right\} \ .$$

Now we use the positivity assumption, adding

$$(n - |R|) \cdot |C| - \sum_{\overline{R}} \sum_{C} u_{ij} \leq 0$$

to the last expression. We get

$$\begin{aligned} c(X, \overline{X}) &\geq \frac{1}{2} \cdot \left\{ \sum_{R} \sum_{\overline{C}} u_{ij} - \sum_{\overline{R}} \sum_{C} u_{ij} - |R| \cdot n + |R| \cdot |C| + n \cdot |C| - |R| \cdot |C| \right\} \\ &= \frac{1}{2} \cdot \left\{ \sum_{R} \sum_{\overline{C} \cup C} u_{ij} - \sum_{R \cup \overline{R}} \sum_{C} u_{ij} - n \cdot (|R| - |C|) \right\} \\ &= \frac{1}{2} \cdot \{ |R| \cdot u^* - |C| \cdot u^* - n \cdot (|R| - |C|) \} = (|R| - |C|) \cdot \frac{u^* - n}{2} \end{aligned}$$

Thus, if $g \leq (u^* - n)/2$ the cut condition is certainly fulfilled for all cuts; therefore we can find at least $g \geq \lfloor (u^* - n)/2 \rfloor$ permutation matrices with weight 2, reducing u^* to $u^* - 2g$. The total number of matrices is now

$$g + (u^* - 2g) = u^* - g \leq u^* + \left\lceil \frac{n - u^*}{2} \right\rceil = \left\lceil u^* + \frac{n - u^*}{2} \right\rceil = \left\lceil \frac{n + u^*}{2} \right\rceil . \quad \blacksquare$$

Lemma 4 has the additional assumption that no matrix entries are zero. To apply it, we have to run the heuristic with the modification that, in step (b), zeros are rounded upwards to 1. Lemma 2 is still true because relation (2) also holds for $t_{ij} = 0$ and $u_{ij} = 1$.

Theorem 4: An $n \times n$ matrix T can be decomposed into a weighted sum of no more than Q permutation matrices ($Q \geq n$) with a total duration that is within a factor of $(Q - n/2)/(Q - n + 1/2)$ of the value t^* that is obtainable without restriction on the number of matrices in the decomposition. The decomposition can be found in time $O(n^3 + Qn \log n)$.

Proof: We choose F as the smallest value such that $u^* \leq 2Q - n$, where U is now defined by $u_{ij} := \max\{\lceil t_{ij}/F \rceil, 1\}$. Lemma 2 with $M = 2Q - n$ and lemma 4 give the result. As for the running time, we have one maximum flow problem and two applications of the edge-coloring procedure of lemma 1. $\blacksquare$

Note that in the theorem we start with $l_1 = 2$ and not with the largest possible l_1 as in methods II and III. The following matrix shows that this is indeed necessary for achieving the claimed bound.

$$\begin{bmatrix} 3 & 1 & 1 & 2 & 2 & 2 & 2 \\ 1 & 3 & 1 & 2 & 2 & 2 & 2 \\ 1 & 1 & 3 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 4 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 4 & 1 & 1 \\ 2 & 2 & 2 & 1 & 1 & 4 & 1 \\ 2 & 2 & 2 & 1 & 1 & 1 & 4 \end{bmatrix}$$

The largest permutation matrix which we can remove is the identity matrix with multiplicity $l_1 = 3$. Continuing the decomposition gives a total of $q = 11$ matrices, whereas lemma 4 guarantees an upper bound of 10 matrices. The example is worked out in detail in the report Rote and Vogel (1990).

2.6 *Comparison of the Three Methods*

Method I is by far the simplest and fastest, but its practical performance is not far from the theoretical bound, and thus will usually be too weak for practical purposes. It is used as a subroutine in method III.

Methods II and III are similar in spirit. Method III *explicitly* formulates the problem of finding as many permutation matrices P^k with a given value of l_k as possible *as a single problem.* This can certainly be no worse than selecting these matrices one by one as in method II, but it is usually better: If there are many matrices with equal l_k there are many possibilities for the first complete matching. Method II chooses an arbitrary one, which remains fixed afterwards, without regarding the effect on subsequent solutions. This is avoided in method III. In this sense, method III is better at being greedy than method II.

3 Implementation of the Heuristics and Computational Results

We have programmed methods II and III and applied them to randomly generated test problems of various sizes in the range from $n = 10$ to $n = 100$. We have tried to get approximately $q \approx 2n$ matrices. We shall first describe our implementation and then report the computational results.

When discussing the quality of the solution of various methods and the effect of different implementation decisions, we shall measure the quality of a solution by the *relative excess* $(d - t^*)/t^*$ of the total duration d over the lower bound t^*. This is an upper bound on the error of the heuristic relative to the optimal solution; since we don't know the optimal solution, we take the lower bound t^*. Thus, when we say that the relative excess has decreased by 40%, for example, this would mean that an instance with $t^* = 100$ which previously had a solution with total duration $d = 110$ has now a solution with $d = 106$.

3.1 *Step (a): Finding the "Unit" F*

The theoretical derivations in section 2 used the bounds $q \leq u^*$ or $q \leq \lceil (u^* + n)/2 \rceil$ on the number q of generated matrices. In practice these bounds are too pessimistic, and thus it makes no sense to try to achieve a specified value of u^* exactly. So we tried to achieve $u^* \approx M$, where M is an input parameter of the program which was determined experimentally. (M had to be taken between about $20n$ and $70n$ to get $q \approx 2n$ matrices, depending on n and on the method.)

We then computed F by the formula

$$F = \frac{1}{2}\left(\frac{t^*}{M-(n-1)} + \frac{t^*}{M}\right), \tag{5}$$

which is the average between the lower and the upper bound for the correct value of F (cf. section 2.2), and we set $u_{ij} := \lceil t_{ij}/F \rceil$. We always accepted the resulting value of u^* and continued the computation with this value u^*. In our experiments, u^* never deviated more than 1% from the target value M except in two instances, and we found this acceptable. In practice, when one is not satisfied with the final resulting number q of matrices, one can go back to the beginning and adjust F.

On the other hand, we did not simply accept the value F that we had: We tried to reduce it as much as possible while still keeping the same u^*. This F-correction step was carried out by a method which is inspired by the methods described in section 2, but simpler and more practical:

We select the maximum of all entries t_{ij}/u_{ij}. (This entry will be the first element u_{ij} which will increase to $u_{ij} + 1$ as F decreases gradually.) If this element belongs to a critical row or column (i.e., $r_i = u^*$ or $c_j = u^*$), u_{ij} is not allowed to increase and thus F must not decrease below t_{ij}/u_{ij}: We stop with this value as the value of F. Otherwise we set $u_{ij} := u_{ij} + 1$, update r_i and c_j accordingly, and repeat the whole procedure.

We can speed up this repeated search for the maximum by computing an initial lower bound F_0 for F: F_0 is the maximum of t_{ij}/u_{ij} over all elements in critical rows and in critical columns. Elements t_{ij}/u_{ij} which are $\leq F_0$ need not be considered in the search for the maximum. Whenever we make a new row or column critical we can try to raise F_0 in order to further reduce the search effort.

We have implemented the search for a maximum among the set of entries t_{ij}/u_{ij} as a sequential search in a linear list. This simple procedure was sufficient, as the average effort for the F-correction step was always less than 5% of the total time for the heuristic. In many cases, the list which had to be searched for a maximum was already empty initially, and thus the matrix U did not have to be changed.

3.2 Step (c): the Decomposition

In step (c), we never used an explicit bottleneck assignment procedure. We rather used the simple heuristic of assuring that no vertex is isolated, as discussed after theorem 2 in section 2.3. This inexpensive starting procedure is used in published codes for the bottleneck assignment problem, e.g. in Burkard and Derigs (1980), chapter 2. As the results showed, we could thus reduce the number of graphs that

were found to contain no matching to a small fraction. For finding matchings we used the method of Hopcroft and Karp (1973).

In method III, step (c) contains maximum flow problems. For finding the largest parameter value g for which the network of section 3.4 has a feasible flow, we used the algorithm of Dinic, which is not the theoretically fastest, but which has been regarded as the fastest in practice (of the traditional algorithms, at least). With this algorithm we successively tested the values $g = 1, 2, 3, 4, \ldots$ by incrementally sending an additional unit of flow into each source vertex.

When we find that no feasible flow for some supply and demand value g exists we have to undo the flow augmentations since the last "valid" flow (with supplies and demands of value $g - 1$), because we need this flow pattern for decomposing it into permutations (unless $g = 1$, of course, in which case we can throw away the whole graph and decrease l_k by one.) We have solved this problem by storing all flow changes in a list at the same time as we carry them out. The maximum required length of this list, for 100×100 matrices, was 297 in all our test runs. It is possible that other maximum flow algorithms would have improved our running times, but as our primary goal was to investigate the quality of the solutions, we did not try alternate implementations.

The decomposition of the flow pattern into matchings was not done by the very fast edge coloring techniques as suggested in section 2.4, but by applying the algorithm of Hopcroft and Karp g times in succession. The reason was that we had already an implementation of this algorithm from method II. The average computation time for this step in our implementation was always less than 10 percent of the total computation time, which is tolerable for an experimental implementation.

3.3 Step (d): Adjusting the Solution

Step (d) has not been discussed at all so far. Note that the performance of steps (a)–(c) is mainly determined by M, since the total duration d after step (c) is Fu^*; and since $u^* \approx M$, and F is given by (5), we have

$$(d - t^*)/t^* \approx \frac{(n-1)/2}{M - (n-1)} .$$

A minor influence comes from the F-correction in step (a), which improves F and makes the quality a little better than the above estimation. The decomposition method in step (c) has the strongest effect on the quality, but only indirectly, by allowing a larger value of M.

Step (d) could be solved optimally as a linear program with n^2 inequalities and q variables $l_1, l_2, \ldots, l_q$, by just using formulation (1), where the matrices

$P^1, P^2, \ldots, P^q$, which are given as the output of step (c), determine the coefficient matrix. The values $l_1, l_2, \ldots, l_q$ which are given by step (c) play no role in this process except possibly as a starting solution.

With this implementation, step (d) is actually not necessarily a "downward adjustment" of the values l_k, as described in section 2. In fact, it is nothing but the original problem with a restricted set of permutation matrices. (Lewandowski, Liu, and Liu (1983) and Burkard (1985) have considered the case of $2n$ given permutation matrices which correspond to a pair of orthogonal Latin squares. In this case the linear program has a special structure which can be used.) We have tried the linear programming approach on a couple of examples, using an experimental code for a numerically stable simplex algorithm developed at our institute by Zhong-Liang Yu. However, the running times were prohibitively high, and thus we did not pursue this direction further.

In our experiments we just used a simple approximation procedure: We successively lower $l_1, l_2, \ldots, l_q$ as much as possible while still maintaining the relation $\sum_{k=1}^{q} l_k P^k \geq T$. This procedure takes $O(nq)$ steps. We also tried to run through the l_k's in reverse order or in random order, but on the average the natural order gave the best results. This simple implementation of step (d) reduced the relative excess $(d - t^*)/t^*$ by about fifty percent for $n = 10$ and by about ten percent for $n = 100$. For method III, the reduction was only about five percent for $n = 100$.

Let us finally discuss the mutual influence between step (d) and the F-correction in step (b). One might wonder whether computing a smallest F which gives a certain value of u^* in step (a) is really necessary in view of step (d). Of course, the total duration $F \cdot u^*$ after step (c) decreases, but one might think that step (d) might diminish the effect of computing the tightest F. A parallel series of test runs which took F directly from (5) without subsequent correction showed that the effect of the F-correction is reduced by step (d). In most cases the F-correction step improved the duration a little; the improvement was smaller for larger n: On the average, it reduced the relative excess of the duration over the lower bound t^* by one fifth for $n = 10$ and by one twentieth for $n = 50$. Only in some rare cases did the solution with F-correction have a slightly longer total duration than without it; thus, we report only the results of the runs *with* F-correction.

3.4 *Computational Results*

For method II, we created matrices of various sizes n, whose elements were uniform random numbers selected from the range $\{1, 2, \ldots, 100\}$. The results, which are shown in table 1, are the averages of 100 instances for each size n. M is the desired value of u^* which was given as input to the program. As mentioned in section 3.1, the actual value of u^* fluctuates very little about M.

Table 1. Computational results of method II. Average results of 100 matrices each, with entries in the range 1–100

n	M	q	failures	time	$(d-t^*)/t^*$	$(d-t^*)/t^*$ (B&L)
10	220	20.10 (18–23)	4.8	0.20	1.007%	7.84%
15	340	30.39 (28–33)	4.6	0.55	1.328%	7.86%
20	450	40.61 (39–43)	3.7	1.16	1.480%	5.13%
30	700	60.27 (58–63)	2.2	3.40	1.648%	5.47%
40	1050	80.68 (77–85)	2.0	7.57	1.596%	4.09%
50	1400	100.81 (98–104)	1.2	14.07	1.504%	2.68%
60	1750	120.47 (115–124)	0.9	23.51	1.460%	2.36%
80	2600	160.34 (156–165)	0.6	53.09	1.375%	2.62%
100	3900	201.77 (196–208)	0.7	100.57	1.121%	1.78%

The values which characterize the quality of the solution are q, number of matrices in the decomposition, and the total duration d. As mentioned above, we chose M in such a way that q would be about $2n$. The table, which also includes the range of values q that appeared in the 100 instances, shows that q is quite stable. The variation of q is still much larger than the variation of u^*, and it is mostly due to step (c).

The sixth column shows the second factor of the quality, the total duration, normalized in terms of the relative excess, in order to make the results comparable for different types of matrices. For contrast, the last column contains the results of the heuristic of Balas and Landweer (1983) for decomposing into only $q = n$ matrices. Of course, this is not a fair comparison because this heuristic solves a more restricted problem. Still, one can see how much may be gained in total duration by allowing more than n matrices. Balas and Landweer (1983) also report that a variation of their algorithm for $q = 2n$ matrices achieved an average relative excess of 2.19% on random 20×20 matrices of the same kind as we used. In this case, with 1.48%, our algorithm is clearly superior. Note the peak of the relative excess at $n = 30$. We have no explanation for this phenomenon.

The fifth column shows the CPU-time in seconds. The programs were written in PASCAL and run on a DEC VAX 11/785 computer. The average times grow slower than $O(n^3)$, which is much better than the behavior of $O(n^{3.5+\varepsilon})$ predicted by theorem 2.

The column entitled "failures" shows the number of unsuccessful trials, i.e., the number of graphs in which no matching was found. Thus, the total number of matching problems solved is q plus the number of failures. This number was often zero, and the maximum of all instances (of method II) decreased from 15 for $n = 10$ to 4 for $n = 100$. (For method III, the situation was similar.) The small numbers in this column, especially for larger n, show that the simple rule of decreasing l_k until the graph has no isolated vertices is a sufficient substitute for the bottleneck assignment algorithm, at least for the random examples which we

Table 2. Computational results of method III (the maximum flow method). Average results of 100 matrices each, with entries in the range 1–1000

n	M	q	flows	failures	time	$(d - t^*)/t^*$
10	350	20.15 (17–22)	20.6	5.1	0.87	0.598%
15	520	29.74 (27–32)	24.5	3.4	2.18	0.870%
20	720	39.48 (36–42)	29.2	2.5	4.49	0.993%
30	1200	59.59 (56–62)	38.0	1.5	12.68	1.014%
40	1800	80.04 (75–86)	46.8	1.0	27.49	0.962%
50	2500	100.39 (96–105)	55.3	0.6	51.31	0.884%
60	3200	119.86 (114–126)	63.3	0.6	84.62	0.847%
80	5000	160.28 (154–166)	78.8	0.3	193.25	0.743%
100	7000	200.25 (192–208)	93.5	0.3	361.22	0.683%

tried. This is in accordance with the theory of random graphs, where it is known that the probability that the procedure which we used produces a graph which has a perfect matching converges to 1 as n goes to infinity, see Bollobás and Thomason (1985) or the book of Bollobás (1985), theorem VII.1, p. 158. This result holds under the assumption that the matrix elements are drawn independently at random from the same distribution. It is therefore not directly applicable to our algorithm because we repeatedly modify the matrix and destroy independence even if it was satisfied for the initial matrix. When we compare our failure rates with the empirical results of Bollobás and Thomason (1985), table 7, p. 60, we see that we have in fact much fewer failures than might be expected from carrying over these results in a straightforward way. One reason which accounts for this discrepancy is that our matrix entries as defined by (3) are small integers by construction. Thus, whenever we decrease the threshold l_k by 1, a whole bunch of new edges comes into the graph. This means that we usually end up with more edges than if we would add edges strictly *one by one* in random order, as prescribed by the model of Bollobás and Thomason.

Table 2 shows the results of method III, the maximum flow method. Since method III is more successful in achieving a small number of matrices, we could allow as larger value of u^* (see the second column). Since this u^* would be of the same order of magnitude as t^* for matrices with entries in the range 1–100, we generated the entries between 1 and 1000. As M is chosen larger, we may expect an improvement over method II (cf. the remarks at the beginning of section 3.3). The last column confirms that this is indeed true. Note again the mysterious peak of the relative excess about $n = 30$. The column entitled "flows" gives the number of flow problems which had to be solved, including the unsuccessful ones. One determination of the maximum value of g for which the network has a feasible flow is counted as one flow problem. For large n, the number of flow problems is clearly smaller than the number q of matrices.

The algorithm takes longer than method II by a factor of about 3.5. This is caused by the greater overhead for building the more complicated network and for the maximum flow algorithm.

Table 3. The effect of the range of the matrix entries t_{ij} on the relative excess with method III. The results are averages of hundred 50 × 50-matrices with $M = 2500$

range of values for t_{ij}	q	flows	failures	time	$(d - t^*)/t^*$
1–100	99.84	54.87	0.69	52.22	0.827%
1–1000	100.39	55.30	0.59	52.49	0.884%
1–10000	100.30	55.32	0.71	52.30	0.890%

Table 4. Method III applied directly to the matrix T. Average results of 100 matrices each

n	entries in the range 1–1000			entries in the range 1–10000		
	q	failures	time	q	failures	time
10	27.90 (24–33)	112.36	3.65	32.76 (25–39)	1102.67	26.41
15	42.64 (38–47)	97.71	7.45	51.33 (45–59)	956.10	47.30
20	56.84 (53–62)	55.31	10.52	69.21 (62–76)	570.96	51.31
30	85.36 (81–93)	23.32	22.21	105.51 (100–111)	243.63	63.35
40	113.33 (108–120)	14.37	43.88	140.95 (137–147)	159.66	96.15
50	140.42 (135–146)	9.23	78.95	175.41 (169–181)	95.46	138.65
60	166.65 (162–172)	6.73	126.62	209.55 (199–218)	69.91	202.68
80	217.98 (210–225)	4.13	282.10	275.20 (267–283)	36.51	403.20
100	267.19 (261–276)	3.23	516.13	338.01 (328–346)	27.37	730.12

We suspected that the good performance of the methods might perhaps be due to the small range of integer values from which the elements of the traffic matrix were taken. Thus we ran a series of test runs with 50 × 50 matrices with entries from various ranges. The results, which are presented in table 3, show that there is at most a small influence on the relative excess. The other parameters are also quite unaffected by the range.

Since method III was so successful in getting few matrices we also tried it directly on the generated matrices T, without intermediate scaling. This always leads to a decomposition with optimal duration. The results are presented in table 4. We can see that the number of matrices depends on the size of the matrix elements, although perhaps to a lesser extent than might be expected. This shows clearly that the scaling approach is definitely a good idea to keep the number of matrices low.

The numbers in table 4 are in accordance with experiments of Inukai (1979) who reports that his heuristic produces an average of about $q = 30$ matrices for hundred randomly generated 10 × 10 problems with integer entries in the range 0–1000. Balas and Landweer (1983) mention that they could achieve an average relative excess of 1.44% for $q = 3n$ matrices and 0.89% for $q = 5n$ matrices in the decomposition of random 20 × 20 matrices with entries in the range 1–100. With our algorithm we could get to the lower bound t^* (i.e., achieve 0%) with at most $3n$ matrices in 98 out of 100 runs and in all cases with less than $5n$ matrices, even with entries in the range 1–1000.

As the number of unsuccessful tests shows, the simple ideas that worked so well in the runs of tables 1 and 2 to keep the number of failures low were no longer sufficient, particularly for smaller n. In the examples of table 4, there were large gaps between successive weights l_k, which were not always bridged by the simple rule of ensuring that the graph has no isolated vertices; using faster bottleneck assignment algorithms would have speeded up these runs.

4 Conclusion

The heuristics of sections 2.3 and 2.4 for duration-optimal decomposition (methods II and III) were introduced as subroutines inside a scaling heuristic. However, the computational results reported in section 3 encourage us to believe that they are interesting in their own right.

We could have simplified our algorithms a lot by filling up the traffic matrix to constant row and column sum. For example, in method II, the cost matrix of the bottleneck assignment problem would just be U itself; in method III, we would not have to double the number of nodes and arcs of the network. Whereas this approach would be satisfactory from a theoretical point of view (cf. section 2.5 of Burkard (1985)) it amounts to giving away some freedom that is inherent in the problem. Our algorithms take this freedom into account in the form of column and row slacks and take advantage of it to obtain better solutions.

Our approach is greedy since it tries to minimize the total number of matrices by removing *large* matrices out of the traffic matrix. This is similar in spirit to earlier heuristic algorithms for duration-optimal decomposition by Inukai (1979), algorithm TSA-2, and by Ito, Urano, Muratani, and Yamaguchi (1977). (For the latter algorithm, see also Inukai (1978).) On the other hand, we solve the individual steps of the greedy algorithm, i.e., maximizing the weight of the next matrix in the decomposition, optimally. This is in contrast with the other mentioned algorithms, where there was no clear formulation of an objective function for the corresponding steps, which were solved in an ad-hoc greedy-type manner.

Suppose that we have chosen to implement step (d) by linear programming. Then, after the matrices P^k and the durations l_k have been determined in steps (a)–(c), we can actually forget the values l_k. Thus one can look at the whole procedure from a different viewpoint: Steps (a)–(c) appear as a preprocessing phase in which the restricted set of matrices $P^1, \ldots, P^q$ is selected from the set of all permutation matrices for the "actual" optimization phase in step (d). With this in mind, it makes no sense to insist that steps (a)–(c) yield a feasible solution at all; on might for example try different rounding schemes in step (b), like rounding to the nearest integer, or rounding downwards, or other "voting

schemes", cf. Woodall (1986), Petit and Térouanne (1990), or Balinski and Young (1982). The only restriction is that no positive entry is rounded to zero.

We remark that, having implemented the linear programming formulation of step (d), we were tempted to apply column generation techniques to add further matrices to the decomposition, quite similar to Minoux (1986) (cf. also Ribeiro, Minoux, and Penna (1989); these two papers solve a version of the problem which is different from ours). However, since these ideas do not fit into the context of this paper and since the computational experiments have been very limited so far, we only mention them here.

The heuristics and exact algorithms proposed in the literature have solved three types of variations of the matrix decomposition problem:

- decomposition into only n (or even fewer) weighted permutation matrices, so that each entry in the traffic matrix is covered by only one permutation, see Balas and Landweer (1983) or Gopal and Wong (1985). The problem treated by Ribeiro, Minoux, and Penna (1989) is also similar;
- duration-optimal decomposition, e.g., Inukai (1979), Burkard (1985);
- decomposition with a small set (usually about $2n$) of permutation matrices which are given in advance, see Lewandowski, Liu, and Liu (1983).

Our bottleneck assignment heuristic and the maximum flow heuristic fall into the second category. The scaling approach, however, allows to control the number of matrices used, and thus it closes the gap between algorithms of the first kind, which produce few matrices, and algorithms of the second kind, which produce too many matrices, without having the restrictions of the third type of algorithms.

References

Balas E, Landweer PR (1983) Traffic assignment in communication satellites. *Operations Research Letters* 2:141–147

Balinski ML, Young HP (1982) *Fair Representation – Meeting the Ideal of One Man, One Vote.* Yale University Press, New Haven and London

Bollobás B (1985) *Random Graphs.* Academic Press

Bollobás B, Thomason A (1985) Random graphs of small order. *Random Graphs '83*: First Poznań Seminar on Random Graphs, August 1983, (M. Karoński and A. Ruciński, eds.), *Ann. Discr. Math.*, vol. 28, North-Holland, pp. 47–97

Burkard RE (1985) Time-slot assignment for TDMA-systems. *Computing* 35:99–112

Burkard RE (1991) Time division multiple access systems and matrix decomposition. *Proceedings of the Fourth European Conference on Mathematics in Industry (ECMI 4)*, (H. Wacker and W. Zulehner, eds.), B. G. Teubner, Stuttgart, and Kluwer Academic Publishers, Dordrecht, pp. 35–46

Burkard RE, Derigs U (1980) *Assignment and Matching Problems. Solution Methods with FORTRAN-Programs.* Lecture Notes in Economics and Mathematical Systems, vol. 184, Springer-Verlag

Cole R, Hopcroft J (1982) On edge coloring bipartite graphs. *SIAM J. Computing* 11:540–546

Frederickson GN, Johnson DB (1982) The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *J. Computer and System Sciences* 24:197–208

Gabow HN, Tarjan RE (1988) Algorithms for two bottleneck optimization problems. *J. Algorithms* 9:411–417

Gopal IS, Wong CK (1985) Minimizing the number of switchings in an SS/TDMA system. *IEEE Trans. Comm.* COM-33:497–501

Hopcroft JE, Karp RM (1973) An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Computing* 2:225–231

Inukai T (1978) Comments on 'Analysis of a switch matrix for an SS/TDMA system'. *Proc. IEEE* 66:1669–1670

Inukai T (1979) An efficient SS/TDMA time slot assignment algorithm. *IEEE Trans. Comm.* COM-27:1449–1455

Ito Y, Urano Y, Muratani T, Yamaguchi M (1977) Analysis of a switch matrix for an SS/TDMA system. *Proc. IEEE* 65:411–419

Lewandowski JL, Liu JWS, Liu CL (1983) SS/TDMA time slot assignment with restricted switching modes. *IEEE Trans. Comm.* COM-31:149–154

Minoux M (1986) Optimal traffic assignment in a SS/TDMA frame: a new approach by set covering and column generation. *RAIRO Recherche Opérationelle/Operations Research* 20:273–286

Petit JL, Térouanne E (1990) A theory of proportional representation. *SIAM J. Discrete Math.* 3:116–139

Rendl F (1985) On the complexity of decomposing matrices arising in satellite communication. *Oper. Res. Lett.* 4:5–8

Ribeiro CC, Minoux M, Penna MC (1989) An optimal column-generation-with-ranking algorithm for very large scale set partitioning problems in traffic assignment. *European J. Operational Research* 41:232–239

Rote G (1989) Eine Heuristik für ein Matrizenzerlegungsproblem, das in der Telekommunikation via Satelliten auftritt (Kurzfassung). *ZAMM · Z. angew. Math. Mech.* 69:T29–T31

Rote G, Vogel A (1990) A heuristic for decomposing traffic matrices in TDMA satellite communication. Report 1990-73, Technische Universität Graz, Institut für Mathematik, February 1990

Tarjan RE (1983) *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia 1983

Woodall DR (1986) How proportional is proportional representation? *Math. Intelligencer* 8(4):36–46

Received: April 1990
Revised version received: November 1992