Pergamon

S0305-0548(96)00010-X

Computers Ops Res. Vol. 23, No. 10, pp. 917–932, 1996

0305-0548/96 $15.00 + 0.00

LINEAR AND SEMI-ASSIGNMENT PROBLEMS: A CORE ORIENTED APPROACH

A. Volgenant†

Vakgroep Actuariaat en Econometrie, University of Amsterdam, Roetersstraat 11/1018, WB Amsterdam, The Netherlands

(Received October 1994; in revised form February 1996)

Scope and Purpose—The Linear Assignment Problem (LAP) has many practical applications, such as the assignment of people to tasks. It is a subproblem of a lot of problems, e.g. the Travelling Salesman Problem. Given a square matrix of cost coefficients the LAP can be described to select a coefficient of each row and each column so that the sum of the selected coefficients is minimal. In some applications a number of columns (or rows) contain the same cost coefficients; this problem is known as the semi-assignment problem. It is important to be able to solve the LAP fast, especially in the case that it is used as a subproblem, and many instances have to be solved for the solution of some considered problem. The core approach appears to be able to solve fast LAP standard problems as well as non-square problems and semi-assignment problems, while the memory requirement for the algorithm is reduced up to a factor of about 20; thus, a personal computer is sufficient to solve large LAP problem instances within short computer times.

Abstract—A Linear Assignment Problem (LAP) with a dense cost matrix can be solved by first making this matrix sparse, i.e. the problem is solved on the core of the matrix. For a known LAP algorithm we give the related modifications. Computational results show that the algorithm is then suited to solve large problems. A standard personal computer can solve problem instances up to size 2000 within about 75 seconds. Furthermore, we describe versions of the algorithm for non-square problems as well as for semi-assignment problems; computational results for problem instances up to size 2500 show again the success of the core oriented approach for assignment problems. Copyright © 1996 Elsevier Science Ltd

1. INTRODUCTION

For the Linear Assignment Problem (LAP) there are different types of algorithms available. The shortest path based approach known from Tomizawa [1] is exploited in many fast LAP algorithms.

Derigs and Metz [2] and Carpaneto and Toth [3] have suggested exploiting the speed of algorithms for sparse problems for the solution of dense problems by selecting the core of the cost matrix, i.e. suitable small elements of the cost matrix, and then solving the problem on the core; if a simple check on optimality is negative, the approach can be repeated on a larger core. In practice this repetition is rare for average problems.

Jonker and Volgenant [4] developed the algorithm LAPJV based on the shortest path approach. In an adapted form, LAPJV can also be used on LAPs with incomplete cost matrices i.e. sparse problems. The main idea behind the new developed version of the algorithm LAPJV is that very large problems can be solved within short solution times even on personal computers. This is impossible with LAPJV simply because of the (large) memory requirement to store the complete cost matrix.

In the next section we describe the details of the modification of the LAPJV algorithm resulting in an algorithm that is capable of solving problems of size up to 2000, within about 75 seconds on a standard personal computer.

For non-square problems we have developed a special version of the core oriented LAP algorithm; given the asymmetric role of columns and rows in the LAPJV algorithm it appears to be best to solve non-square problems as problems with less rows than columns. For problem

†A. Volgenant is an associate professor in Operations Research; his main research interests are in the field of Combinatorial Optimization on problems as the Steiner Tree Problem in Graphs, the Travelling Salesman Problem and related problems. He has published in journals as *Operations Research, Operations Research Letters, Networks* and *The European Journal of Operational Research.*

instances of size 2000 and random uniform cost coefficients, the solution times grow almost linearly with the number of rows, up to about 80 seconds on a personal computer.

Kennington and Wang [5] have generalized the LAPJV algorithm for the Semi-Assignment Problem. On the basis of this generalization we have developed a core oriented Semi-Assignment algorithm. Problem instances of size 2500 and 100 columns with random uniform cost coefficients can be solved within 8 seconds again on a personal computer. The solution times grow almost linearly with the number of columns, up to about 50 seconds for problem sizes of 1500 and random uniform cost coefficients.

We mention some possibilities and efforts to improve shortest path based assignment algorithms and, especially, LAPJV.

Nicholson [6] suggested solving shortest path problems faster by a so-called 'Two-sided Dijkstra algorithm'; the main idea is to build up the shortest path tree (for the path between source and destination) from two sides at the same time. In general, a shortest path has been determined faster because the two parts meet each other before one of the trees meet the source or the destination. In this case, however, the adjustment of the dual variables in LAPJV is very time consuming, so we have not succeeded in speeding up LAPJV in this way. Brouwer [7] has simulated results for multiprocessors; the gain of parallel processing is mainly in the initialization phase of the algorithm. For two processors he comes to a gain of about 35% with 5% as a result of the parallel computation of the two-sided shortest paths in the augmenting phase of LAPJV.

Kennington and Wang [8] report speed ups for LAPJV from 3.27 to 4.32 for size 1000 on a ten processor Symmetry S81 (80386 CPUs) using Fortran. Kennington and Wang [5] report about 6% shorter computer times for dense problems by some implementation refinements and using the LAPJV algorithm for the Semi-Assignment Problem.

For sparse problems the LAPJV algorithm is defeated by the auction algorithm of Bertsekas, [9] although the differences become smaller with a diminishing sparsity, resulting in a difference of 35% for a sparsity of 5%.

Hao and Kocur [10] have developed a LAP algorithm that exploits bipartite matching in the initial phase. Their algorithm is faster than LAPJV on small cost range problems, while it is slower for large cost ranges and for sparse problem instances. For these instances and size 16 384 with 229 376 cost coefficients they report an average solution time of about 27 seconds on an HP-9000/720 workstation for the LAPJV algorithm (coded in C language).

2. THE CORE ORIENTED APPROACH

It is interesting that problems with a sparse cost matrix can be solved much faster than problems of the same order with a full density cost matrix. In addition, computational experience has shown that the entries of the cost matrix associated with an optimal assignment have, in general, very small values compared with the other entries of the cost matrix. These two considerations suggested constructing a modified algorithm for the complete problem by selecting the core of the matrix, i.e. the relevant entries or, alternatively formulated, the elements that have a good chance to be part of an optimal solution.

The adaption of LAPJV mainly concerns a selection procedure and a different data structure, while the main structure remains the same. Instead of a cost matrix c_{ij}, stored as a two-dimensional array, the elements are now stored using three arrays. The first array, *cc*, is a cost array in which all the rows of the original cost matrix have been placed next to each other, leaving out the irrelevant (inadmissible or infinite) cost elements. The second array, *kk*, runs parallel to *cc* and contains the corresponding column indices of the selected entries in *cc*. The last array, called first, contains the pointers to the first entry of each row.

The modified algorithm LAPMOD (a Pascal listing can be found in the Appendix) consists of three phases.

1. The transformation phase; transforming the complete matrix into a sparse one.
2. The solution phase, solving this sparse problem by the (sub)procedures:
   (i) initialization (column reduction, reduction transfer, augmenting row reduction);
   (ii) augmentation (by finding a shortest path);
   (iii) determine (the value of the dual variables).
3. The optimality check phase, checking the optimality for the original problem.

A detailed description and explanation of Phase 2 can be found in Jonker and Volgenant [4], while the other phases are considered now, to start with the transformation phase.

With LAPMOD only a sparse matrix has to be stored, which requires much less memory. The size of this sparse matrix is, apart from the size of the cost matrix, determined by the number of selected elements in a row. The best value of this number is the one that results in the fastest average computing time. This value differs for different sizes and ranges of the cost matrix, and also depends on the selection procedure used. We have constructed two straightforward procedures denoted as

```
Procedure SMALL;
comment selects pp small elements from each row;
begin
   t :=0;
   for i:= 1 to n do
   begin
      SELECTION:= { c_i1, ..., c_ipp };
      cr:= (sum_k=1..pp SELECTION(k)) div pp;
      for j:= pp+1 to n do if c_ij < cr then
      begin
         repeat if t<pp then t:=t+1 else t:=1 until SELECTION(t) ≥ cr;
         SELECTION(t):= c_ij; update cr
      end;
      if c_ii ∉ SELECTION then
      begin t:=pp+1; SELECTION(t):= c_ii end
      { add diagonal element to keep the sparse problem feasible }
   end
end;
```

Fig. 1. The selection procedure SMALL.

```
Procedure SMALLEST;
comment selects the pp smallest elements from each row;
begin
   for i:= 1 to n do
   begin
      SELECTION:= { c_i1, ..., c_i,pp };
      cr:= max_k=1..pp [ SELECTION(k) ] → k*;
      for j:= pp+1 to n do if c_ij < cr then
      begin
         SELECTION(k*):= c_ij; update cr (and k*)
      end;
      if c_ii ∉ SELECTION then SELECTION(pp+1):= c_ii
      {add diagonal element to keep the sparse problem feasible}
   end
end;
```

Fig. 2. The selection procedure SMALLEST.

SMALL and SMALLEST. The former selects reasonably small cost elements, the latter selects the smallest elements of each row. Carpaneto and Toth [3] used a threshold method, which is unsuited for memory restricted situations since the threshold is determined after reduction of the cost matrix.

The procedure SMALL (see Fig. 1) starts with (temporarily) selecting the first *pp* cost coefficients of a row of the cost matrix, with *pp* a constant to be chosen *a priori*. Then each following cost element is compared with the (running) average of the elements selected until then. If this element is smaller it is exchanged with a selected cost element that is larger than the average; otherwise the next cost element is considered; see for the details also the appendix where SMALL is given in Pascal as a part of the LAPMOD algorithm.

Procedure SMALL appears to be considerably faster than procedure SMALLEST (see Fig. 2), but uses a less subtle way of selecting and is thus more likely to miss important entries. Hence, procedure SMALL has to select more elements than procedure SMALLEST in order to have a good change of finding an optimal assignment at one go, which not only increases the selection time, but also has an effect on the actual solution time. As a consequence, the two LAP procedures that only differ in the selection procedures, are about equally fast, but only for large problems with small cost ranges. When applied to small problems or problems with large cost ranges procedure SMALL is preferred.

The sparse version of the LAPJV algorithm can solve the sparse problem created in the initialization phase. To guarantee that it has at least one feasible solution, the diagonal cost elements have been added to the sparse matrix in both selection procedures. The optimal solutions (x') and (u', v') are computed, corresponding to the primal and dual problems (P') and (D') associated with the sparse cost matrix (c'). The check whether these solutions are also optimal for the original primal and dual problems (P) and (D), is based on the following proposition.

Proposition: If the dual solution (u', v') of problem (D') is feasible for the original dual problem (D), i.e. if $c_{ij} - u_i' - v_j' \geq 0$ holds for all i and j, then (u', v') is optimal for problem (D) and the primal solution (x') of problem (P') is feasible and optimal for the original problem (P).

The proof is omitted, since it is straightforward and well known.

So for each row $i(= 1, \ldots, n)$ we have to check whether $c_{ij} - u_i - v_j < 0$ holds for $j = 1, \ldots, n$ and if not, then row i has to be de-assigned. If no de-assignment occurs an optimal assignment has been found because of the proposition, otherwise the algorithm repeats the augmentation subprocedure in Phase 2 to assign the de-assigned rows.

It is an option to terminate the algorithm after such an unsuccessful optimality check. The quality of the obtained (heuristic) solution is easy to give. As a corollary of the proposition the gap between the heuristic and the optimal criterion value is at most

$$\sum_{(i,j)\in A} |c_{ij} - u_i - v_j| \text{ with } A = \{(i,j): c_{ij} - u_i - v_j < 0\}.$$

Of course the value of *pp* can be adapted row-wise to the properties of the problems to solve, e.g. a larger value of *pp* is better for a row in the cost matrix where the values of the elements are close to the minimal cost element in the row.

It can be rational to adapt the core creation to non-standard or artificial cost matrices, e.g. in the case that the LAP is solved as a subroutine in the context of another problem, say a quadratic assignment problem, see Section 4 for some computational results concerning this issue.

3. IMPLEMENTATION

We now discuss details on the implementation of the procedures for LAPJV and LAPMOD that have been developed in Pascal.

The maximum size of a single variable in the Turbo Pascal (version 5.0) used is 64 kbytes, restricting the size n of the problems that can be solved. In LAPJV the complete cost matrix, containing n^2 integers (or $2n^2$ bytes), has to be stored. Consequently, the maximum value for n is 181. (For other Pascal versions or other languages and other memory situations, e.g. on mainframes or on workstations the situation is, of course, different). This restriction can easily be avoided by making use of the 'heap' memory. The cost matrix c is not stored as a two-dimensional array, but as a one-dimensional array with length n that contains pointers to one-dimensional

arrays with length n of integers:

```
type vec        = array[1..n] of integer;
     heapmatrix = array[1..n] of ^vec;
var c: heapmatrix.
```

This adaption leads to an increased maximum for n—with maximum heap size the algorithm can solve problems of size 500 easily—and to reduced computing times for larger problems; of course this reduction is Pascal compiler dependent. The adapted LAPJV will be denoted as LAPJV$^+$.

The LAPMOD procedure only has to store the sparse cost matrix c', which is much smaller than the original matrix c. However, the memory requirement for the storage of c' will exceed 64 kbytes for increasing values of n. Again this restriction can be overcome:

(a) the long arrays *cc* and *kk* analogously can be stored as sparse 'heap-matrices',
(b) the array 'first' of length $n+1$ is now replaced by an array 'number' of length n, containing the number of elements selected in each row. As an additional advantage the elements with negative reduced cost in the optimality check are easily inserted:

```
type vecpp        = array[1..maxpp] of integer;
     spheapmatrix = array[1..n] of ^vecpp;
var  cc, kk: spheapmatrix.
```

With a heap size of about 500 kbytes, it is possible to select 125 elements from each row when $n = 100$ (both *cc* and *kk* then contain 125 000 integers or 250 000 bytes); for increasing values of n the maximum number of selected elements per row decreases linearly.

4. COMPUTATIONAL RESULTS

Results are given for problem sizes up to 2000 and cost coefficients uniformly drawn on the intervals 1–100, 1–1000 and 1–10 000. The computation times are obtained on a personal computer with 80486SX processor, 25 MHz. They are all averages for ten full density problems generated with the Turbo Pascal 5.0 random generator (randseed = 1, ..., 10). The given times are net times and do not include times for the input or the generation of cost matrices.

We have tested the procedures SMALL and SMALLEST that select elements from each row of the cost matrix, as discussed in Section 2. The best number of elements for both procedures can only be experimentally determined. Different values for the number of selected elements result in different average computing times. Table 1 shows that for LAPMOD and experimentally seen SMALL is the best selection procedure. It is generally faster than SMALLEST, and also less sensitive to the range of cost coefficients. However, when using SMALL for large problems, it is possible that the arrays *cc* and *kk* cannot entirely be contained in memory because too many elements have to be selected. In

Table 1. Computation times (in seconds) of LAPMOD for problem size 1000 (minima starred; averages of 10 problems)

Number of selected elements	Range of cost coefficients			Average	
	1–100	1–1000	1–10 000		
10	22.4	25.5	26.5	24.8	
11	18.2	21.7	25.1	21.7	
12	18.4	21.1	23.7	21.2	procedure
13	17.8	19.8	22.4	20.0	
14	18.0*	20.0	20.2	19.4	SMALLEST
15	18.3	19.5	20.5	19.4	
16	18.5	19.0*	19.2*	18.9*	
19	23.1	20.7	21.0	21.6	
20	22.3	19.2	20.2	20.6	
21	22.4	20.1	19.5	20.7	procedure
22	20.6	19.4	18.9*	19.6	
23	18.1*	19.5	18.9	18.8	SMALL
24	18.2	18.8*	19.0	18.7*	
25	18.3	18.9	19.2	18.8	

that case selection procedure SMALLEST can help since LAPMOD with this subprocedure needs, in general, much fewer elements to find an optimal solution.

We note that in Table 1 (and in tables to come) sometimes the computing times decrease with increasing cost range. We have no explanation for this unexpected phenomenon, which can also be seen in other LAP algorithms, see, for example, Hao and Kocur [10].

For problems up to size 500 it is possible to store the cost matrix c and still have enough memory left to contain the related arrays cc and kk. Table 2 shows the total computation times for LAPJV$^+$ (where c is a heapmatrix type variable, see Section 3) and LAPMOD. For problems with sizes much larger than 500 the original cost matrix cannot be stored in internal memory. The last column in the table gives the number of selected elements that results in the fastest average computing time for a problem size. For the problem sizes 1500 and 2000, however, the mentioned values are not necessarily the best values. We have chosen them as ample in order to avert the risk of having more than one optimality check.

When too few elements have been selected, it is possible that LAPMOD will not find an optimal assignment at one go and more optimality checks will be needed. When the cost matrix c is stored in the external memory then with each optimality check c has to be re-read from this memory, e.g. a hard disk. Then it is wise, because of the relative slowness of the external memory, to choose the number of selected elements from each row to be larger than the values mentioned in Tables 1 and 2. In this way the risk of having more than one optimality check can be decreased. Table 3 shows the average number of optimality checks; in most cases an optimal solution is found at once; this holds for all problem instances with sizes 1000, 1500 and 2000.

For the range 1–10 000 and problem size 1000, we can compare LAPJV with LAPMOD by means of the results of Kennington and Wang [11]; they reported a solution time of 80.64 s using a Symmetry computer with a 80386 processor and a code in Fortran. Where Fortran is faster than Pascal (twice?) and this processor slower than the 80486 processor yielding a time of about 19 s, we think that LAPMOD is faster than LAPJV. Kennington and Wang [8] have reported a computation time of 40.0 s on the Symmetry S81 (a multiprocessor system with 20 Intel 80386 CPUs).

Table 2. Computation times (in seconds) for large full density assignment problems (—no result because of memory shortage)

Problem size	Cost range	LAPJV$^+$	LAPMOD	Number of selected elements
300	1–100	0.88	2.19	16
	1–1000	0.91	1.95	
	1–10 000	0.95	2.03	
500	1–100	2.26	5.38	18
	1–1000	2.80	5.09	
	1–10 000	3.52	5.09	
1000	1–100	—	18.22	24
	1–1000	—	18.76	
	1–10 000	—	19.02	
1500	1–100	—	40.31	30
	1–1000	—	41.72	
	1–10 000	—	42.04	
2000	1–100	—	71.78	39
	1–1000	—	75.37	
	1–10 000	—	74.74	

Table 3. Average number of optimality checks

Problem size	Number of selected elements	Range of cost coefficients		
		1–100	1–1000	1–10 000
100	11	1.30	1.20	1.00
150	15	1.10	1.00	1.00
200	15	1.30	1.00	1.10
300	16	1.50	1.00	1.10
400	17	1.00	1.30	1.10
500	18	1.40	1.10	1.10

The LAP instances solved in this section have uniformly distributed cost matrices, in contrast to problems which have some rows and columns being more expensive than others. In such a case LAPMOD will select elements mainly from the cheaper columns and the found first assignment will probably not be optimal. A difficult cost matrix can be created as follows:

(1) generate a uniformly distributed cost matrix with cost range 1–100;
(2) for each row: add a random number (1–100) to all the elements in this row;
(3) for each column: add a random number (1–100) to all the elements in this column.

Ten problems of this type were solved by LAPJV$^+$ and LAPMOD for the sizes 100 and 150. On 'normal' problems, algorithm LAPJV$^+$ was slightly faster than LAPMOD, while on difficult instances LAPJV$^+$ appears to approximately three times faster than LAPMOD. With 'normal' problems the average number of optimality checks is 1.3 and 1.1 for the sizes 100 and 150; with difficult instances these numbers increase to 3.5 and 4.0 for the same problem sizes. In these type of instances it is wise to increase the number *pp* of coefficients that are selected.

Finally, we will try to compare the LAPMOD algorithm with the algorithm CTCS, the best algorithm of Carpaneto and Toth [3]. This comparison is not straightforward, since CTCS has been developed in Fortran for CDC computers (Cyber 730) and on supercomputers (Cray XMP/12), while LAPMOD is tailored for Pascal on personal computers. A second difference is the motivation for the development of the algorithms; the former aims first for short solution times and the latter for a restricted memory, with short solution times as a second place intention.

Nevertheless we can give a rough comparison by means of an 'intermediary' algorithm, LSAP, as published by Burkard and Derigs [12]. Table 4 combines the results of Carpaneto and Toth [3], column 'CTCS', and the results of Jonker and Volgenant [4], column 'LAPJV'. This table indicates that CTCS and LAPMOD are roughly comparable in speed.

We note that the developed LAPMOD algorithm is also suited to solve sparse problems, although we will not give computational results for this type of problem.

5. LINEAR ASSIGNMENT ON RECTANGULAR MATRICES

There are two options to modify LAPJV for rectangular matrices: the cost matrix is constructed with less rows than columns, or with more rows than columns. In the first case all the rows but not all the columns will be assigned; we denote this version by LAP(MOD)row and the second one by LAP(MOD)column. We discuss the modifications of the steps of LAPJV for the first case of cost matrices, i.e. with fewer rows than columns.

The first step of the LAPJV-algorithm is *column reduction*. For each column j a row index i^* is found with minimum $c[i, j]$ $(i = 1, \ldots, n)$. If row i^* is unassigned, then column j is assigned to i^*. This assignment is thus determined by the minimum cost in column j. The minimum $c[i^*, j]$ in column j however, is not necessarily the minimum for row i^*. The assignment of row i to column j must be based on the minimum in row i, so column reduction is not allowed.

The purpose of the second step, *reduction transfer*, is to reduce further the reduced costs $c[i, j] - u[i] - v[j]$ of assigned rows. Because column reduction is not permitted, all the rows are still unassigned; so this phase is useless.

Augmenting row reduction is the third step of the algorithm; in this step an attempt is made to find augmenting paths starting in an unassigned row; for this row i a column j^* is found where the

Table 4. A comparison of CTCS and LAPMOD by means of LSAP (—unavailable result); the smaller the factor, the faster the considered algorithm

Problem size	Cost range	LAP algorithm		
		CTCS	LAPJV	LAPMOD
150	1–100	0.312	0.335	0.273
	1–10 000	—	0.496	0.226
	1–100 000	0.442	—	—
200	1–100	0.226	0.237	0.226
	1–10 000	—	0.496	0.364
	1–100 000	0.442	—	—

minimum reduced costs occur. If j^* is unassigned, the alternating path leads to augmentation of the solution. After an assignment is made row i can be reduced by its minimum reduced costs; if the second minimum is higher, reduction transfer can take place by increasing the elements in column j^*. It is possible to apply augmenting row reduction to matrices with fewer rows than columns because the procedure is based on the minimum reduced costs in a row. The application of this step gives a partial solution of assigned rows and columns.

The final step of the algorithm is *augmentation*. For each unassigned row an alternating path of rows and columns is built until an unassigned column is found. By assigning all rows in the path to their succeeding column one more assignment is added to the partial solution. If each row is assigned by this procedure an optimal solution is found.

The first step in the algorithm LAP(MOD)column for cost matrices with n rows and m columns ($n > m$) is to add $n - m$ columns. All the elements in these columns must be equal but the value can be chosen arbitrarily (here the value zero is chosen). The cost matrix is now square so the complete algorithm LAPJV can be applied.

The procedure *column reduction* is applied only on the first m columns. The reduction of the added $n - m$ columns does not improve the partial solution. The second minimum reduced costs $c[i,j] - v[j]$ for each row will be found in one of the added columns. This value is zero, thus no reduction transfer will take place. During the augmenting row reduction, and the augmentation phase, an optimal solution for the square cost matrix is determined.

This problem, with more rows than columns, can also be seen as a semi-assignment problem. An extra column is added with each element having the same value (here zero is chosen). To each column j ($j = 1, \ldots, m$) one row will be assigned; the remaining $n - m$ rows will be assigned to the added column. We denote this algorithm as semiLAPcolumn. We describe the structure of the semi-assignment problem in the next section.

Computational experiments showed that for rectangular cost matrices the algorithm LAProw is the fastest compared to LAPcolumn and semiLAPcolumn; for problems of size 500 LAProw is at least five times faster than LAPcolumn and semiLAPcolumn for the cost range 1–100, and at least ten times faster for the range 1–10 000.

An explanation of this result lies in the number of augmenting paths. In LAProw n (= the number of rows) paths must be found, which is the minimum of n and m. In LAPcolumn also n paths are found but this is the maximum of n and m. Only m paths are necessary for an optimal solution. Even with a partial solution obtained from column reduction the augmenting row reduction and augmentation steps take more time in LAPcolumn and semiLAPcolumn than in LAProw. Because all the added columns have the same values, it takes more time to find an unassigned column, thus the alternating paths are longer. So each augmentation step takes more time than in a square matrix with randomly generated cost elements. In LAProw it is easier to find an unassigned column, as the total number of columns is larger than the number of rows.

Due to the structure of the semi-assignment algorithm (see the next section) the computation times for semiLAPcolumn are much longer. If in the row reduction augmentation step a row is de-assigned from column j this row must contain the lowest reduced costs of all the rows assigned to this column. Each time a row is de-assigned all the rows assigned to j must be checked. This phase costs too much time to make the semiLAPcolumn algorithm competitive.

LAPMOD for rectangular matrices

Given the computational results the modification of the algorithm for problems up to size 2000 is based on LAProw. This modification, called LAPMODrow, consists of three steps:

(1) selection of pp small cost elements (SELL_PP),
(2) augmenting row reduction (ARR),
(3) augmentation.

In the first step, SELL_PP, pp elements are selected according to the procedure SMALL (see Section 2) for each row in the cost matrix. The free array contains all the rows, because no assignments are made during this procedure. The ARR and augmentation steps are not modified compared with LAPMOD.

Table 5 gives a comparison for the results with LAPMOD and LAPMODrow. The values chosen for pp (number of selected elements) are based on the number of columns (pp is 24 for 1000 columns,

Table 5. Computation times (in seconds) for rectangular matrices with LAPMODrow algorithm

		Problem size (rows × columns)						
LAP algorithm	Cost range	500 1000	750 1000	1000 1000	500 1500	1000 1500	1250 1500	1500 1500
LAPMOD row	1–100	9.0	13.6	18.4	13.2	26.8	33.6	40.8
	1–1000	9.0	13.6	19.3	13.4	26.8	33.6	43.2
	1–10 000	9.0	13.7	20.0	13.4	26.8	33.6	44.1
LAPMOD	1–100			18.2				40.3
	1–1000			18.8				41.7
	1–10 000			19.0				42.0
LAP algorithm	Cost range	500 2000	1000 2000	1500 2000	1750 2000	1900 2000	1950 2000	2000 2000
LAPMOD row	1–100	17.6	35.6	53.8	62.9	68.5	70.4	72.6
	1–1000	17.8	35.5	53.5	62.7	69.1	72.3	78.6
	1–10 000	17.8	35.6	53.5	62.8	68.9	71.8	78.3
LAPMOD	1–100							71.8
	1–1000							75.4
	1–10 000							74.7

30 for 1500 columns and 39 for 2000 columns). For problem instances with the number of rows much smaller than the number of columns, the value of *pp* can be chosen smaller than these values, resulting in even shorter solution times.

We can conclude that, as expected, the computation times to find optimal solutions for rectangular matrices are shorter than the times for the corresponding square matrices as appears from the results in the previous sections. So the development of a specialized algorithm for rectangular matrices is justified. We note that the solution times of LAPMODrow for size 2000 are about proportional to the number of rows.

6. SEMI-ASSIGNMENT

In the assignment problems discussed so far, each row was assigned to a distinct column. In some applications a number of columns contain the same cost coefficients. Instead of assigning rows to each of these columns, it is possible to assign all these rows to the same column. This problem, known as the semi-assignment problem, can be formulated as:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{m} c[i,j] \cdot x[i,j]$$

$$\text{subject to } \sum_{j=1}^{m} x[i,j] = 1 \qquad (i = 1, \ldots, n)$$

$$\sum_{i=1}^{n} x[i,j] = b_j \qquad (j = 1, \ldots, m)$$

$$x[i,j] \geq 0 \qquad (i = 1, \ldots, n;\ j = 1, \ldots, m)$$

where b_j denotes the number of assignments for column j, with $\Sigma_{j=1}^{m} b_j = n$. Kennington and Wang [5] designed a shortest path algorithm (SAP) based on LAPJV for this problem.

The shortest augmenting path algorithm may be viewed as a dual method, because dual feasibility and complementary slackness conditions are maintained at each step, while the solution moves to primal feasibility.

We discuss the modification of LAPJV and LAPMOD for the semi-assignment problem.

Modification of LAPJV

The algorithm consists of four phases; three of them heuristics: *column reduction*, *reduction transfer* and *row reduction augmentation*. If these heuristics fail to obtain an optimal solution, a shortest augmenting path phase takes care of the final assignments.

The *column reduction* for the semi-assignment problem (the procedure is given in Fig. 3 top) is essentially the same as in LAPJV. For each column j a set TIE is created, containing all rows i with

```
Procedure COLUMN REDUCTION;
  begin
    for each column j do
    begin
      z[j]:= b[j];
      h:= min {c[i, j] | i = 1, ..., n};
      TIE:= {all rows i | c[i, j] = h};
      repeat
        select i1 ∈ TIE;
        TIE:= TIE − {i1};
        if x[i1] = 0 then begin assign i1 to j; z[j]:= z[j] − 1 end
      until TIE = ∅ or z[j] = 0
    end {for j}
  end;

procedure REDUCTION TRANSFER;
  begin
    SATURATED:= {all columns j | z[j] = 0};
    for all j∈ SATURATED do
      begin
        I_j:= {all rows i assigned to column j};
        Δ_j:= min {c[i,k] − u[i] − v[k] | i∈I_j, k = 1, ..., n, k≠j};
        v[j]:= v[j] + Δ_j;
        for all i∈I_j do u[i]:= u[i] + Δ_j
      end {for j}
  end;

procedure ROW REDUCTION AUGMENTATION;
  begin
    LIST:= {all unassigned rows};
    for all i∈ LIST do
      begin
        u1:= min {c[i, j] − u[i] − v[j] | j = 1, ..., n};
        SCAN:= {j | c[i, j] − u[i] − v[j] = u1, j = 1, ..., n};
        for all j∈ SCAN do
          if z[j] > 0 then begin assign i to j; goto augment end;
        select a j∈ SCAN;
        I_j:= {all rows assigned to j};
        u2:= min {c[i,k] − u[i] − v[k] | i∈I_j, k = 1, ..., n, k≠j};
        select a p with c[p, j] − u[p] − v[j] = u2;
        x[p]:= 0; x[i]:= j;                    { de-assign p and assign i to j }
augment: u[i]:= u[i] + u1;
        if z[j] = 0 then REDUCTION TRANSFER on column j
      end {for i}
  end;
```

Fig. 3. The procedures for semi-assignment.

minimum $\{c[i,j]\}$. Therefore, more assignments may be achieved at each step. The number of unassigned places in column j is given by array z, while the array y contains row indices with, on the first $b[1]$ places, the rows assigned to column 1, etc.

The second heuristic, *reduction transfer*, reduces the values of the dual variables of the saturated columns. After this procedure (see Fig. 3 middle) a saturated column is made more expensive relative to the unsaturated columns, so in general an alternating path in the augmentation phase will earlier reach an unsaturated column. Let I_j be the set of rows assigned to column j. The amount Δ_j by which the $v[j]$ can be reduced is selected in such a way that

$$\Delta_j = \min_{i \in I_j} \{c[i,k] - u[i] - v[k] \mid k \neq j\}.$$

For each row i assigned to column j, the value of $u[i]$ will be increased by an amount of Δ_j.

During the *row reduction augmentation* (RRA) procedure (Fig. 3 bottom) the number of unassigned rows is further reduced while maintaining dual feasibility and the complementary slackness conditions. For each unassigned row i^* the procedure determines a set SCAN that contains the columns j in which the minimum reduced costs $u1$ occur for row i^*. If a column j in SCAN is unsaturated, then row i^* is assigned to this column; if no such column exists, an arbitrary column j in SCAN is selected. The set I_j holds all the rows i assigned to column j. The row i in I_j that contains the minimum reduced cost $c[i,j] - u[i] - v[j]$ is de-assigned and row i^* is assigned to column j. The dual variable $u[i^*]$ is increased by $u1$ and if $z[j] = 0$ reduction transfer is applied to column j.

There are two alterations for semiLAP with respect to LAPJV. The first concerns the selection of the de-assigned row. In LAPJV only one row is assigned to a column, but in the semi-assignment problem more rows can be assigned, so one row must be chosen. This is the row where the minimum reduced costs occur.

If reduction transfer takes place in LAPJV, a de-assigned row is immediately considered again, because the minimum reduced costs may now occur in a different column. In semiLAP this row must wait until the row reduction augmentation is applied again.

The effect of row reduction augmentation in reducing the number of unassigned rows is extraordinary, particularly for problems with a large cost range. Based on their experience Kennington and Wang [5] recommend applying the row reduction augmentation T times, where $T = 2 \log_{10} R$ with $R =$ cost range.

If the previous heuristics fail to obtain an optimal solution, a shortest alternating path augmentation procedure completes the last (few) assignments. For the semi-assignment problem this procedure is essentially the same as in LAPJV (see Section 2). An array is added which contains the place where the row-index y of row i is stored. This array is needed in the augmenting path to assign row i to its succeeding column in the path.

Semi-LAPMOD

To solve problems of large size the complete cost matrix is transformed into a sparse one. For the semi-assignment problem we denote this algorithm as semiLAPMOD.

In the first step of this algorithm pp small elements are selected according to the procedure SMALL for each row. In LAPMOD the next step is column reduction. For each column j a row index i^* is determined with minimum $c[i,j]$ $(i = 1, \ldots, n)$. In the semi-assignment algorithm a list of ties with minimum $c[i,j]$ is set up to achieve more assignments in each column. So the column reduction step takes place after the selection procedure. Each time the minimum cost value in a column j must be found all rows i are checked; this appears to be time expensive. The computational results in Table 6 show that the algorithm solves problems up to 50% faster without the column reduction and reduction transfer steps.

The row reduction augmentation step in semiLAPMOD is identical to this step in semiLAP. The shortest augmenting path procedure is derived from LAPMOD. Again an array is added to keep track of the row-indices.

Table 7 gives the computation times for semiLAP, SAP_KW (see Kennington and Wang [5]), LAPJV, semiLAPMOD and LAPMOD; the number of columns is fixed on 100, just as in the results of Kennington and Wang [5]. We have chosen the numbers of selected elements (denoted by pp) such that the average solution times of the three ranges are minimal.

Table 8 shows the results for problems of size 1500 and several numbers of columns. Clearly SemiLAPMOD is more sensitive to the range of cost coefficients than LAPMOD.

The results from the previous section indicate that it is justified to develop a specialized algorithm for semi-assignment problems. SemiLAPMOD is the fastest algorithm for problems of size larger than 500; this is due to the selection of a very small number of elements. The computational results show that the computation times for semiLAPMOD grow about linearly with the number of columns.

Table 6. Computational times (in seconds) for semiLAPMOD with and without column reduction (pp is the number of selected cost elements per row)

		Problem size (rows × columns)				
Semi LAPMOD	Cost range	500 × 100 $pp = 11$	500 × 250 $pp = 13$	1000 × 100 $pp = 7$	1000 × 200 $pp = 9$	1000 × 500 $pp = 15$
without column reduction	1–100	1.38	2.80	2.55	4.43	9.86
	1–1000	1.40	2.87	2.57	4.46	10.25
	1–10 000	1.46	3.06	2.67	4.70	10.75
with column reduction	1–100	2.37	5.71	3.90	7.80	22.81
	1–1000	2.39	5.72	3.90	7.88	23.36
	1–10 000	2.40	5.83	4.04	8.06	23.88

Table 7. Computation times (in seconds) for semi- and linear assignment problems with 100 columns (—no results available; pp is the number of selected cost elements per row)

		LAP algorithm						
					semiLAPMOD		LAPMOD	
Number of rows	Cost range	semiLAP	SAP_KW	LAPJV$^+$	Time	pp	Time	pp
200	1–100	0.29	0.61	0.31	0.51		0.91	
	1–1000	0.38	0.75	0.36	0.54	11	0.95	15
	1–10 000	0.41	0.79	0.43	0.58		0.93	
400	1–100	0.77	1.41	1.61	1.05		3.45	
	1–1000	1.01	2.02	1.71	1.11	11	3.53	17
	1–10 000	1.08	2.24	1.93	1.19		3.52	
600	1–100	1.29	2.71		1.59		6.86	
	1–1000	1.74	3.98	—	1.70	11	7.03	20
	1–10 000	1.95	4.66		1.74		7.14	
800	1–100	2.05	3.86		2.03		11.8	
	1–1000	2.47	5.12	—	2.04	7	12.2	22
	1–10 000	3.02	6.16		2.19		12.4	
1000	1–100	2.70			2.77		18.2	
	1–1000	3.20	—	—	2.90	7	18.8	24
	1–10 000	4.02			3.07		19.0	
1500	1–100	3.94			3.78		40.3	
	1–1000	5.68	—	—	3.96	7	41.7	30
	1–10 000	7.28			4.28		42.0	
2000	1–100	5.37			4.85		71.8	
	1–1000	7.98	—	—	5.33	7	75.4	39
	1–10 000	10.19			5.62		74.7	
2500	1–100	7.04			6.06			
	1–1000	10.18	—	—	6.72	7	—	—
	1–10 000	14.30			7.28			

Table 8. Computation times (in seconds) for LAPMOD and semiLAPMOD for problems with 1500 rows

		Number of columns				
LAP algorithm	Cost range	100 $pp = 7$	300 $pp = 9$	500 $pp = 12$	750 $pp = 16$	1500 $pp = 30$
semi LAPMOD	1–100	3.8	9.3	13.8	20.3	40.5
	1–1000	4.0	9.3	15.0	22.7	44.0
	1–10 000	4.2	9.8	15.5	22.8	49.4
LAPMOD	1–100					40.3
	1–1000					41.7
	1–10 000					42.0

7. SUMMARY AND CONCLUSIONS

We have described a core approach for the Linear Assignment Problem and for variants, i.e. semi-assignment and problems with different numbers of rows and columns. We can conclude that the computational results show that the new developed LAPMOD algorithm is suited to solving large linear assignment problems on a personal computer in small computation times, illustrating the success of the core approach for these problems as well as for the semi-assignment and the non-square problems. For the considered problem instances the results give guidance about how many cost elements have to be chosen in each row. In practice, one can select more elements in rows where the values of the elements are closer to the minimum cost element in the row.

The selection procedures SMALL and SMALLEST, although straightforward, give good results. Since the selection forms a substantial part of the LAPMOD algorithm (approximately one half of the total computing time, where solving the sparse problem and the optimality check each take a quarter), it may very well be possible to further improve LAPMOD by a more clever selection procedure.

Given the computational results it is justified to have specialized algorithms for assignment problems with more (or less) rows than columns and semi-assignment problems.

Becuase of the small memory requirements the algorithms are very well suited for use on personal computers. For computers with more memory available we think the core approach will be able to solve problems with much larger sizes.

Acknowledgements—P. Warmerdam and Ms E. Reijers assisted with the preparation of the computational results. An anonymous referee gave constructive comments that have improved the presentation of this article.

REFERENCES

1. N. Tomizawa, On some techniques useful for the solution of transportation problems. *Networks* **1,** 173–194 (1971).
2. U. Derigs and A. Metz, An in-core/out-of-core method for solving large scale assignment problems. *Zeitschrift für Operations Research* **30,** 181–195 (1986).
3. G. Carpaneto and P. Toth, Primal-dual algorithms for the assignment problem. *Discrete Appl. Math.* **18,** 137–153 (1987).
4. R. Jonker and A. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing* **38,** 325–340 (1987).
5. J. Kennington and Z. Wang, A shortest augmenting path algorithm for the semi-assignment problem. *Opns Res.* **40,** 178–187 (1992).
6. T. Nicholson, Finding the shortest route between two points in a network. *The Computer Journal* **9,** 275–280 (1966).
7. F. Brouwer, The linear assignment problem: simulation of a multiprocessor algorithm (in dutch). Master Thesis, University of Amsterdam, Institute of Actuarial Sciences and Econometrics (1988).
8. J. Kennington and Z. Wang, An empirical analysis of the dense assignment problem: sequential and parallel implementations. *ORSA J. Computing* **3,** 299–306 (1991).
9. D. P. Bertsekas, The auction algorithm for assignment and other network flow problems: a tutorial. *Interfaces* **20,** 133–149 (1990).
10. J. Hao and G. Kocur, An implementation of a shortest augmenting path algorithm for the assignment problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **12,** 453–468 (1993).
11. J. Kennington and Z. Wang, Solving dense assignment problems on a shared memory multiprocessor. Department of Computer Science & Engineering, School of Engineering and Applied Science, Southern Methodist University, Dallas, Texas (1988).
12. R. E. Burkard and U. Derigs, *Assignment and Matching Problems: Solution Methods with Fortran Programs.* Springer, Berlin, Heidelberg, New York, 1–11 (1980).

Readers may address inquiries to Dr A. Volgenant at: tonv@fee.uva.nl
The Pascal source code of the algorithms is available on request.

APPENDIX: THE ALGORITHM LAPMOD

```
program LAPMOD(input,output);
uses dos,crt;
const maxn  = 2000; inf = 30000; { inf is a suitably large number}
      pp    = 39;  { number of selected elements in the rows, depends on problem size}
      maxpp = 61;  { check memory available}
type  vec   = array[1..maxn] of integer;
      vcpp  = array[1..maxpp] of integer;
var   cc,kk: array[1..maxn] of ^vcpp;
      c,d,free,lab,number,todo,u,v,x,y: vec;
      dj,f,f0,h,h2,i,i0,i1,j,j0,j1,j2,k,l,l0,last,min,n,t,td1,td2,u1,u2,v0,vj: integer;
      zlap: longint;
      ok: array[1..maxn] of boolean;
```

```
function optcheck: boolean;                                          { # # # optimality check }
var newfree: boolean;
begin
    l:=0;
    for i:=1 to n do
    begin
        for j:=1 to n do read(input, c[j]);
        newfree:=false;
        for j:=1 to n do if c[j] < u[i] + v[j] then
        begin
            inc(number[i]); newfree:=true;
            cc[i]^[ number[i] ]:=c[j];
            kk[i]^[ number[i] ]:=j
        end;
        if newfree then begin y[x[i]]:=0; x[i]:=0; inc(l); free[l]:=i end
    end; { for i }
    optcheck:= (l=0)
end; {optcheck}

procedure selpp_cr;                    { # # # selects pp elements using SMALL and performs column reduction}
var     diag,cr : integer;
        s       : longint;
begin
    for j:=1 to n do v[j]:=inf;
    t:=0;
    for i:=1 to n do
    begin
        for j:=1 to n do read(input,c[j]);
        s:=0;
        for j:=1 to pp do
        begin cc[i]^[j]:=c[j]; kk[i]^[j]:=j; s:=s+c[j] end;
        cr:=s div pp;
        for j:=pp+1 to n do if c[j]<cr then
        begin
            repeat if t<pp then inc(t) else t:=1; h:=cc[i]^[t] until h>=cr;
            cc[i]^[t]:=c[j]; kk[i]^[t]:=j;
            s:=s-h+c[j]; cr:=s div pp
        end; { for j }
        x[i]:=0; diag:=c[i];
        for t:=1 to pp do
        begin
            j:=kk[i]^[t];
            if cc[i]^[t]<v[j] then begin v[j]:=cc[i]^[t]; y[j]:=i end;
            if j=i then diag:=inf
        end; { for t }
        if diag<inf then                { add diagonal element to guarantee the existence of a feasible solution }
        begin
            t:=pp+1; cc[i]^[t]:=diag; kk[i]^[t]:=i;
            if diag<v[i] then begin v[i]:=diag; y[i]:=i end
        end
        else t:=pp;
        number[i]:=t
    end; { for i }
    for j:=n downto 1 do
    begin
        i:=y[j];
        if x[i]=0 then x[i]:=j else begin x[i]:=-abs(x[i]); y[j]:=0 end
    end { for j }
end; {selpp_cr}

procedure transfer;                                          { # # # performs reduction transfer }
begin
    l:=0;
    for i:=1 to n do
    if x[i]<0 then x[i]:=-x[i]
    else if x[i]=0 then begin inc(l); free[l]:=i end else
    begin
        min:=inf; j1:=x[i];
        for t:=1 to number[i] do
        begin
            j:=kk[i]^[t];
            if j<>j1 then if cc[i]^[t]-v[j]<min then min:=cc[i]^[t]-v[j]
        end; { for t}
        t:=1; while kk[i]^[t] <> j1 do inc(t);
```

```
        v[j1]:=cc[i]^[t] - min
    end
end; {transfer}

procedure ARR;                                    { # # # performs augmenting row reduction }
var cnt: integer;
begin
    cnt:=0
    repeat
        h:=1; l0:=l; l:=0;
        while h<=l0 do
        begin
            i:=free[h]; inc(h); v0:=inf; vj:=inf;
            for t:=1 to number[i] do
            begin
                j:=kk[i]^[t]; dj:=cc[i]^[t] - v[j];
                if dj<vj then if dj>=v0 then begin vj:=dj; j1:=j end
                else begin vj:=v0; v0:=dj; j1:=j0; j0:=j end
            end; { for t }
            i0:=y[j0];
            if v0<vj  then v[j0]:=v[j0] - vj+v0
                    else if i0>0 then begin j0:=j1; i0:=y[j0] end;
            x[i]:=j0; y[j0]:=i;
            if i0>0 then
                if v0<vj  then begin dec(h); free[h]:=i0 end
                        else begin inc(l); free[l]:=i0 end
        end; { while }
        inc(cnt)
    until cnt=2
end; { ARR }

procedure augmentation;
label 1,2;
begin
    repeat
        l0:=l;
        for l:=1 to l0 do
        begin
            for j:=1 to n do begin d[j]:=inf; ok[j]:=false end;
            min:=inf; i0:=free[l];
            for t:=1 to number[i0] do
            begin
                j:=kk[i0]^[t]; dj:=cc[i0]^[t] - v[j]; d[j]:=dj; lab[j]:=i0;
                if dj<=min then
                begin
                    if dj<min then begin td1:=0; min:=dj end;
                    inc(td1); todo[td1]:=j
                end
            end; { for t }
            for h:=1 to td1 do
            begin j:=todo[h]; if y[j]=0 then goto 2; ok[j]:=true end;
            td2:=n; last:=n+1;
            repeat
                j0:=todo[td1]; dec(td1); i:=y[j0]; tod[td2]:=j0; td2:=td2 - 1;
                t:=1; while kk[i]^[t]<>j0 do inc(t); h:=cc[i]^[t] - v[j0] - min;
                for t:=1 to number[i] do
                begin
                    j:=kk[i]^[t];
                    if not ok[j] then
                    begin
                        vj:=cc[i]^[t] - v[j] - h;
                        if vj<d[j] then
                        begin
                            d[j]:=vj; lab[j]:=i;
                            if vj=min then
                            begin
                                if y[j]=0 then goto 1;
                                inc(td1); todo[td1]:=j; ok[j]:=true
                            end
                        end
                    end
                end; { for t }
                if td1=0 then
                begin
```

```
                min:=inf-1; last:=td2+1;
                for j:=1 to n do if d[j]<=min then if not ok[j] then
                begin
                    if d[j]<min then begin td1:=0; min:=d[j] end;
                    inc(td1); todo[td1]:=j
                end; { for j }
                for h:=1 to td1 do
                begin j:=todo[h]; if y[j]=0 then goto 1; ok[j]:=true end
            end
        until false;
    1: for k:=last to n do begin j0:=todo[k]; v[j0]:=v[j0]+d[j0]-min end;
    2: repeat i:=lab[j]; y[j]:=i; k:=j; j:=x[i]; x[i]:=k until i=i0
    end; {for l }
    for i:=1 to n do
    begin
        j:=x[i]; t:=1; while kk[i]^[t]<>j do inc(t);
        u[i]:=cc[i]^[t]-v[j];
    end; { for i }
  until optcheck;
end; {augmentation}

begin                                                    { # # # Main program }
  n:=maxn;
  for i:=1 to n do begin new(cc[i]); new(kk[i]) end;
  assign(input,'file'); reset(input);
  selpp_cr; transfer; arr; augmentation;
  zlap:=0; for i:=1 to n do zlap:=zlap+u[i]+v[i];
  close(input)
end.
```