Computing 38, 325–340 (1987)

Computing

A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems

R. Jonker and **A. Volgenant,** Amsterdam

Received May 6, 1986

Abstract — Zusammenfassung

A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. We develop a shortest augmenting path algorithm for the linear assignment problem. It contains new initialization routines and a special implementation of Dijkstra's shortest path method. For both dense and sparse problems computational experiments show this algorithm to be uniformly faster than the best algorithms from the literature. A Pascal implementation is presented.

AMS Subject Classifications: 90C08, 68E10.

Key words: Linear assignment problem, shortest path methods, Pascal implementation.

Ein Algorithmus mit kürzesten alternierenden Wegen für dichte und dünne Zuordnungsprobleme. Wir entwickeln einen Algorithmus mit kürzesten alternierenden Wegen für das lineare Zuordnungsproblem. Er enthält neue Routinen für die Anfangswerte und eine spezielle Implementierung der Kürzesten-Wege-Methode von Dijkstra. Sowohl für dichte als auch für dünne Probleme zeigen Testläufe, daß unser Algorithmus gleichmäßig schneller als die besten Algorithmen aus der Literatur ist. Eine Implementierung in Pascal wird angegeben.

1. Introduction

The linear assignment problem (LAP) is useful as a relaxation for difficult combinatorial optimization problems like quadratic assignment, and traveling salesman. Furthermore, theoretical developments for the LAP can often be extended to other problems, such as minimum cost flow and transportation.

The first well known LAP algorithm, Kuhn's Hungarian method [22], was published in 1955. After 1977 several more or less new algorithms were proposed for example by Barr, Glover and Klingman [2], Hung and Rom [18], Bertsekas [3], and Balinski [1]. None of these authors even mentioned the existence of the shortest augmenting path algorithm of Tomizawa [29] from 1971. Dorhout [10, 11] improved it to what was for years the fastest LAP algorithm available. Still earlier, in 1969, Mack [24] developed an algorithm that is a forerunner of Tomizawa's. It is equivalent to the Hungarian method, but more comprehensible.

After a review of assignment algorithms, we describe in Sections 4, 5 and 6 a new algorithm LAPJV, based on shortest augmenting paths. A Pascal code is presented

in Section 7. Its average computation times are uniformly lower than those of the other algorithms, as shown in the final section.

An LAP with costs $c[i,j]\,(i,j=1 \ldots n)$ can be formulated as a linear program:

$$\begin{aligned} &\min \ \Sigma_{ij}\, c[i,j] \cdot x[i,j] \\ &\text{subject to} \\ &\qquad \Sigma_j\, x[i,j]=1 \qquad (i=1 \ldots n), \\ &\qquad \Sigma_i\, x[i,j]=1 \qquad (j=1 \ldots n), \\ &\qquad x[i,j] \geq 0 \qquad (i,j=1 \ldots n). \end{aligned}$$

The dual problem is:

$$\begin{aligned} &\max \ \Sigma_i\, u[i]+\Sigma_j\, v[j] \\ &\text{subject to} \\ &\qquad c[i,j]-u[i]-v[j] \geq 0 \qquad (i,j=1 \ldots n). \end{aligned}$$

With the dual variables $u[i]$ and $v[j]$ the reduced costs are $c[i,j]-u[i]-v[j]$ $(i,j=1 \ldots n)$. So, the dual problem is to find a reduction of the costs matrix with maximum sum and non-negative reduced costs.

In the following, indices i and j refer to rows and columns respectively; $x[i]$ is the column index assigned to row i and $y[j]$ the row index assigned to column j, with $x[i]=0$ for an unassigned row i and $y[j]=0$ for an unassigned column j; the dual variable $u[i]$ corresponds to row i and $v[j]$ to column j. We denote the reduced costs by: $\text{cred}[i,j]=c[i,j]-u[i]-v[j]$, and sometimes we refer to the dual variables as "prices".

2. A Review of Assignment Algorithms

Methods to solve the LAP can be classified in three categories, that are based on algorithms for

a) maximum flow,
b) shortest paths,
c) linear programming.

Most *algorithms based on maximum flow* are primal-dual methods. For an introduction to these methods we refer to Papadimitriou and Steiglitz [26]. The Hungarian algorithm of Kuhn [22] actually served as the algorithm from which the general primal-dual algorithm was derived. The original method has computational complexity $O(n^4)$, but later $O(n^3)$ versions were developed (Lawler [23]). Jonker and Volgenant [20] give some simple, but effective improvements.

Bertsekas [3] also presents a primal-dual algorithm. The method is Hungarian-type, and the best version even switches to the Hungarian algorithm itself as soon as the original method becomes less effective. We describe part of it in Section 4.

The *methods based on shortest paths* are dual algorithms: dual feasibility exists and primal feasibility has to be reached. This is achieved by considering the LAP as a minimum cost flow problem, solved by steps that involve finding shortest paths on an auxiliary graph.

In this group two algorithms, both of complexity $O(n^3)$, stand out: Hung and Rom's [18] and Tomizawa's [29]. The former is the more ingenious, but the latter the fastest, as shown in Section 8. Tomizawa augments partial assignments into a complete solution by primal steps in each of which one shortest augmenting path is determined. Hung and Rom's initial solution is complete, but may be infeasible. They determine in each step a shortest path tree, which takes more effort, but may contain more disjoint augmenting paths. We consider shortest augmenting path LAP algorithms separately in Section 3.

The so-called Bradford method of Mack [24] is also of interest, especially for its intuitively appealing presentation. As originally presented, it has computational complexity $O(n^4)$. The method is equivalent to the Hungarian algorithm; adapting it to obtain complexity $O(n^3)$ results in an algorithm close to Tomizawa's (Jonker [19]).

Good results on sparse LAPs are obtained by Carraresi and Sodini [7] with an algorithm based on the shortest path method of Glover et al. [15, 16].

The *linear programming based algorithms* in the third category are specialized versions of the simplex method. The best published results are from Barr, Glover and Klingman [2]. A major difficulty with these methods is the phenomenon of zero pivot steps. A drawback is their relatively complex implementation as compared to the other approaches. Computational experiments (Hung and Rom [18], McGinnis [25]) show that these algorithms are outperformed by the best primal-dual and dual methods.

The $O(n^3)$ signature algorithm presented by Balinski [1] and analyzed by Goldfarb [17] also belongs to this category. It considers feasible dual solutions corresponding to trees in the bipartite graph of row and column nodes. The algorithm can be considered a variant of the Hungarian method. Nothing definite is known yet about its computational performance.

3. Shortest Augmenting Paths Based Algorithms

Linear assignment is a special case of minimum cost flow, for which an algorithm exists called "Buildup" in Papadimitriou and Steiglitz [26]. Ford and Fulkerson [14] attribute this method to Jewell (1959) and to Busacker and Gowen (1961). It uses flow augmentation along paths in an auxiliary network, that, depending on the current flow, can be constructed from the original one.

Tomizawa [29] noted that shifting from the original costs of the assignment problem to the (non-negative) reduced costs allows the algorithm of Dijkstra [12] to solve the shortest path problems. With $O(n)$ flow augmentations, this leads to an $O(n^3)$ computational complexity of the algorithm. The theoretical improvements for minimum cost flow algorithms of Edmonds and Karp [13] can also be applied to the assignment problem. The resulting method is equivalent to that of Tomizawa.

In the original version of Tomizawa an augmentation step consists of finding a shortest augmenting path with both initial row and final column specified.

Following Tabourier [28], Dorhout [10, 11] shows computational advantages in leaving a choice for the final column.

The following sections are devoted to the steps of shortest augmenting path LAP algorithms:

Step 1: *Initialization*

Step 2: *Termination*, if all rows are assigned.

Step 3: *Augmentation*
Construct the auxiliary network and determine from an unassigned row i to an unassigned column j an alternating path of minimal total reduced cost, and use it to augment the solution.

Step 4: *Adjust the dual solution* to restore complementary slackness.
Go to step 2.

We discuss initialization strategies in the next section. How to modify shortest path algorithms for assignment augmentation is the subject of Section 5. Section 6 describes a simple way to adjust row and column prices after shortest path augmentation.

4. The Initialization Phase

A standard method of initialization in LAP algorithms is column and row reduction. For each column j a row index i^* is determined with minimum $c[i,j]$ $(i=1 \ldots n)$, $v[j]$ is set to $c[i^*,j]$ and, if i^* is unassigned, j is assigned to i^*. Next, one finds for every unassigned row i the column j^* with $c[i,j]-v[j]$ $(j=1 \ldots n)$ minimum and assigns j^*, if unassigned, to row i.

In our assignment algorithm the initialization is primarily aimed at reaching a high initial reduction of the costs matrix. It consists of three procedures, discussed below:

- reduction of columns,
- reduction transfer from unassigned to assigned rows,
- augmenting reduction of unassigned rows.

The first procedure is standard *column reduction*. An implementation detail is considering the columns in reverse order. So, the low indexed columns are most likely to remain unassigned. As a consequence, if minimum reduced costs in a row occur at an unassigned column, this column is automatically selected as the first in which the minimum occurs.

The second procedure is *reduction transfer*. Its objective is to further reduce assigned rows, but it has no direct net effect on the reduction sum. Afterwards a higher reduction sum may be obtained when unassigned rows are reduced.

Consider a row i assigned to a column, say k. By sufficiently decreasing the price of column k, row i can be reduced by the current second minimum of the reduced costs. This additional reduction of assigned rows leads to an increase of some reduced costs in unassigned rows, so that these may later be reduced further. The effect of the procedure is twofold, as assigned columns are made more expensive relative to the

unassigned ones. Bertsekas [3] calls this "outpricing" of assigned columns. In general the shortest paths in the augmentation phase will now earlier reach some unassigned column.

The straightforward procedure for reduction transfer is given in Fig. 1. For clarity we have not taken into account that at the start of the procedure all $u[i]$ have value zero. Furthermore, one can keep track during the column reduction for which rows reduction transfer will certainly be useless.

```
procedure REDUCTION TRANSFER;
begin
    for each assigned row i do
    begin
        j1:=x[i]; μ:=min {c[i,j]−v[j] | j=1 ... n, j<>j1};
        v[j1]:=v[j1]−(μ−u[i]); u[i]:=μ
    end
end.
```

Fig. 1. Procedure for reduction transfer

Clearly, reduction transfer may also be applied in the course of the augmentation phase. Computational experiments showed that this does not improve the performance of the algorithm LAPJV (Section 7).

Augmenting row reduction is the third initialization procedure. An attempt is made to find augmenting paths starting in unassigned rows, to which at the same time reduction is transferred. In the process assigned columns remain so, but rows may become assigned, unassigned or reassigned.

```
procedure AUGMENTING ROW REDUCTION;
begin
    LIST:={all unassigned rows};
    for all i∈LIST do
    repeat
        u1:=min {c[i,j]−v[j] | j=1 ... n};
        select j1 with c[i,j1]−v[j1]=u1;
        u2:=min {c[i,j]−v[j] | j=1 ... n, j<>j1};
        select j2 with c[i,j2]−v[j2]=u2 and j2<>j1;
        u[i]:=u2;
        if u1<u2 then v[j1]:=v[j1]−(u2−u1)
                 else if j1 is assigned then j1:=j2;
        k:=y[j1]; if k>0 then x[k]:=0; x[i]:=j1; y[j1]:=i; i:=k
    until u1=u2 (* no reduction transfer *) or k=0 (* augmentation *)
end.
```

Fig. 2. Procedure for augmenting row reduction

The procedure is given in Fig. 2. In each step of the for-loop an alternating path is started up from an unassigned row, say i. Consider the column j where the minimum reduced costs in row i occur. If j is unassigned, the alternating path leads to augmentation of the solution. If not, the path is extended by reassigning column j to

row i, thus unassigning the row k previously assigned to column j. Clearly, row i can now be reduced by its minimum reduced costs, but if the second minimum is higher, reduction transfer is possible by increasing the elements in column j. If so, we next consider row k, as for this row the minimum reduced costs may now occur in a column different from j. If this is the case, the alternating path is extended as before. If not, and furthermore the minimum is still unique, the path may even reverse direction, and be extended by rows and columns visited before. The process is continued until either an additional assignment is found, or no reduction transfer takes place.

The two previous reduction procedures both have computational complexity $O(n^2)$. It can be shown that for augmenting reduction at most $O(n^2 \cdot R)$ steps are taken, with R the range of the cost coefficients. With each step involving $O(n)$ operations, the complexity is at most $O(n^3 \cdot R)$. A different argument is as follows. We determine $O(n)$ alternating paths. Each extension of a path takes $O(n)$ operations. So, an $O(n^3)$ computational complexity is obtained by simply allowing no more than n path extensions. In practice the procedure never even approaches this number of extensions.

Computational experiments show best results when the procedure of augmenting row reduction is performed twice.

The advantages of this initialization strategy over standard column and row reduction amply compensate the additional computation time. On full density problems with $n = 100$ average total time has been decreased by 10% (on cost range $1 - 100$) to 18% (on $1 - 1000$). This initialization phase takes 60% to 70% of total run time, whereas simple column and row reduction would take about 20%. We expect the effect of these initialization routines to be larger on augmentation approaches less efficient than ours. Table 1 illustrates the advantages. It gives the average reduction sum (the value of the current dual solution) and the average number of assignments in the partial primal solution. These figures indicate how much effort must still be put in the augmentation phase of the algorithm. The increased average numbers of zero reduced costs coefficients suggest that this method is also useful for assignment algorithms based on maximal flow.

Table 1. *Column and row reduction compared to the initialization in LAPJV (averages for 25 problems of each type with $n = 100$)*

	cost range	column and row reduction	initialization in LAPJV
reduction sum	1 – 100	87.2	96.6
in % of optimum	1 – 1000	87.2	98.0
number of	1 – 100	75	90
assignments	1 – 1000	75	95
number of zero reduced	1 – 100	188	205
cost coefficients	1 – 1000	142	162

5. The Augmentation Phase

Augmentation starts by finding an alternating path. This is a sequence of, alternately, row and column indices, with the first an unassigned row, the last a column, and the intermediate columns and rows assigned in successive pairs. If the final column is unassigned, augmentation of a partial solution can take place along such a path by assigning all rows in the path to their succeeding column, which results in one more assignment.

Augmentation in shortest path based algorithms (step 3, in Section 3) can best be described without direct reference to the underlying minimum cost flow problem. It requires only a simple modification of the shortest path method of Dijkstra [12]. In Fig. 3 we give two procedures. Dijkstra's algorithm SHPATH1 determines a shortest path tree rooted in node kk and traceable in the pred-array. The set $A[i]$ contains all nodes j for which arc $\langle i,j\rangle$ exists. Procedure SHPATH-AUGMENT is its modification for the assignment problem, which determines the shortest augmenting path for one additional assignment in row kk.

```
procedure SHPATH1(kk);
begin
    TOSCAN:={1 ... n}-{kk}; for j:=1 ... n do d[j]:=∞;
    i:=kk; d[kk]:=0; μ:=0;
    repeat
        for all j∈A[i]∩TOSCAN do
            if μ+c[i,j]<d[j] then begin d[j]:=μ+c[i,j]; pred[j]:=i end;
        μ:=∞;
        for all j∈TOSCAN do if d[j]<μ then begin μ:=d[j]; i:=j end;
        TOSCAN:=TOSCAN-{i}
    until TOSCAN={}
end.

procedure SHPATH-AUGMENT(kk);
begin
    TOSCAN:={1 ... n}; for j:=1 ... n do d[j]:=∞;
    i:=kk; d[kk]:=0; μ:=0;
    repeat
        for all j∈A[i]∩TOSCAN do
            if μ+cred[i,j]<d[j] then begin d[j]:=μ+cred[i,j]; pred[j]:=i end;
        μ:=∞;
        for all j∈TOSCAN do if d[j]<μ then begin μ:=d[j]; μj:=j end;
        i:=y[μj]; TOSCAN:=TOSCAN-{μj}
    until y[μj]=0;
    <augment along the path from column μj to row kk>
end.
```

Fig. 3. A shortest path algorithm according to Dijkstra and a modified version for augmentation in assignment algorithms

This procedure only describes the method. For the actual implementation an adapted version of Dijkstra's shortest path algorithm as in Fig. 4 is to be preferred. The shortest path algorithms in Figs. 3 and 4 differ in the use of a set SCAN, containing all rows that can be scanned for the current minimum d-value (the

variable μ). The set may be updated while scanning, and (relatively expensive) searching for a new value of the minimum does not take place until SCAN is empty. Especially for sparse networks this leads to substantially lower computation times.

```
procedure SHPATH2(kk);
begin
  TOSCAN:={1 ... n}-{kk}; for j:=1 ... n do d[j]:=∞;
  d[kk]:=0; SCAN:={kk}; μ:=0;
  repeat
    select any i∈SCAN; SCAN:=SCAN-{i};
    for all j∈A[i]∩TOSCAN do if μ+c[i,j]<d[j] then
    begin
      d[j]:=μ+c[i,j]; pred[j]:=i;
      if d[j]=μ then begin SCAN:=SCAN+{j}; TOSCAN:=TOSCAN-{j} end
    end;
    if SCAN={} then
    begin
      μ:=min {d[j] | j∈TOSCAN; d[j]>μ};
      for j∈TOSCAN do if d[j]=μ then SCAN:=SCAN+{j};
      TOSCAN:=TOSCAN-SCAN
    end
  until SCAN={}
end.
```

Fig. 4. Modified version of Dijkstra's shortest path method, basis for improved augmentation

Augmentation in our LAP algorithm is given by the procedure AUGMENT in Fig. 5, which is based on SHPATH2. Some minor improvements have been made. Instead of a list SCAN containing rows, a list of columns facilitates updating of column prices. Furthermore, we do not keep and update row prices. These can be determined easily when needed due to complementary slackness. Updating of column prices takes place as will be discussed in Section 6.

The column sets READY, SCAN and TODO are mutually disjoint, and READY $\cup$ SCAN $\cup$ TODO $=\{1 \ldots n\}$. So in the code one array COL of length n is sufficient, with the elements of READY kept in front, just before the elements of SCAN. This set is scanned first-in-first-out. So, its elements transfer automatically to READY. The remaining places in the array are used for TODO.

Just like the initialization, the augmentation phase has computational complexity $O(n^3)$. So this also holds for the entire algorithm. As for the memory requirements: the full density version uses a costs matrix and eight arrays of n elements.

Derigs and Metz [8] investigated implementations of Dijkstra's shortest path algorithm in assignment methods. The fastest implementation turned out to be very similar to ours. They discovered that in their algorithms it is advantageous to determine complete shortest path trees, so that more than one augmenting path per iteration may be found. Our algorithm is faster when in each iteration only one augmenting path is determined, which is probably due to the extensive initialization procedures.

```
procedure AUGMENT;
begin
    for all unassigned i* do
    begin
        for j:=1 ... n do begin d[j]:=c[i*,j]-v[j]; pred[j]:=i* end;
        READY:={}; SCAN:={}; TODO:={1 ... n};
        repeat
            if SCAN={} then
            begin
                μ=min {d[j] | j∈TODO}; SCAN:={j | d[j]=μ}; TODO:=TODO-SCAN;
                for all j∈SCAN do if y[j]=0 then go to augment
            end;
            select any j*∈SCAN; i:=y[j*]; SCAN:=SCAN-{j*}; READY:=READY+{j*};
            for all j∈TODO do if μ+cred[i,j]<d[j] then
            begin
                d[j]:=μ+cred[i,j]; pred[j]:=i;
                if d[j]=μ then
                    if y[j]=0 then go to augment else
                    begin SCAN:=SCAN+{j}; TODO:=TODO-{j} end
            end
        until false; (* repeat always ends with go to augment *)
augment:
        (* price updating *)
        for all k∈READY do v[k]:=v[k]+d[k]-μ;
        (* augmentation *)
        repeat
            i:=pred[j]; y[j]:=i; k:=j; j:=x[i]; x[i]:=k
        until i=i*
    end
end.
```

Fig. 5. The procedure AUGMENT for augmentation in the algorithm LAPJV

We experimented with augmentation based on the new shortest path methods from Glover et al. [15, 16]. However, we did not find a faster procedure than one based on SHPATH2. Carraresi and Sodini [7] report very good results with an algorithm based on these shortest path methods. It must be noted that this LAP algorithm performs well only on (very) sparse problems. Karp [21] also improved shortest path based algorithms, but his modifications are more theoretically interesting than practically. By using priority queues, he reduced expected running time for the LAP to $O(n^2 \cdot \log n)$.

6. Adjustment of the Dual Solution

After augmentation of a partial assignment the values of the dual variables must be updated to restore complementary slackness, that is,

$$c[i,k]-u[i]-v[k]=0, \quad \text{if } x[i]=k \quad (i=1 \dots n), \tag{1}$$

$$c[i,j]-u[i]-v[j] \geq 0 \qquad (i,j=1 \dots n). \tag{2}$$

Substituting the values $u[i]$ from (1) in (2) leads to:

$$c[i,k]-v[k] \leq c[i,j]-v[j] \qquad (j=1 \dots n).$$

So, all assignments in the (partial) solution must correspond to row minima in the reduced costs matrix. This simple observation is useful in the update step of shortest path based algorithms. The price $v[k]$ of every assigned column k (with $y[k]=i$) must be adjusted so that

$$c[i,k]-v[k]=\min\{c[i,j]-v[j] \mid j=1 \ldots n\}.$$

In the procedure AUGMENT (Fig. 5) this is achieved just before augmentation actually takes place. Where necessary, column entries in the reduced costs matrix are increased by decreasing the current column prices $v[j]$ $(j=1 \ldots n)$ by $\mu-d[j]$ if $d[j]<\mu$, so that μ is the minimum value in row kk. The corresponding values of $u[i]$ with $i=y[j]$ must be increased by the same amount. The d-values and μ are obtained from the modified shortest path procedure, clarifying the role of the $d[j]$ $(j=1 \ldots n)$ in the procedures SHPATH-AUGMENT and AUGMENT.

7. A Pascal Implementation

In Fig. 6 we present a Pascal code for the algorithm from the previous sections. We suppose integer valued costs $c[i,j]$ $(i,j=1 \ldots n)$, but the code is easily adapted for a real valued costs matrix. The type "mat" is an integer $n \times n$ matrix, and the type "vec" an integer array of length n.

Listings of the Fortran code for dense LAPs and of the Pascal and Fortran code for sparse LAPs can be obtained from the authors on request.

```
function LAPJV (n: integer; c: mat; var x,y,u,v: vec): integer;

{n: problem size;
 c: costs matrix;
 x: columns assigned to rows;
 y: rows assigned to columns;
 u: dual row variables;
 v: dual column variables}

label augment;
const inf=1000000; {inf is a suitably large number}
var  f,h,i,j,k,f0,i1,j1,j2,u1,u2,min,last,low,up: integer;
     col,d,free,pred: vec;

{col: array of columns, scanned                (k=1 ... low-1),
                        labeled and unscanned  (k=low ... up-1),
                        unlabeled              (k=up ... n);

 d:    shortest path lengths;
 free: unassigned rows (number f0, index f);
 pred: predecessor-array for shortest path tree;
 i,i1: row indices; j, j1, j2: column indices;
 last: last column in col-array with d[j]<min.}

begin
   for i:=1 to n do x[i]:=0;
```

```
for j:=n downto 1 do                                  {# # # # COLUMN REDUCTION}
begin
    col[j]:=j; h:=c[1,j]; i1:=1;
    for i:=2 to n do if c[i,j]<h then begin h:=c[i,j]; i1:=i end;
    v[j]:=h;
    if x[i1]=0 then begin x[i1]:=j; y[j]:=i1 end
        else begin x[i]:=-abs(x[i]); y[j]:=0 end
end;

f:=0;                                                 {# # # # REDUCTION TRANSFER}
for i:=1 to n do
if x[i]=0 then                                        {# # unassigned row in free-array}
    begin f:=f+1; free[f]:=i end else
if x[i]<0 then                                        {# # no reduction transfer possible}
    x[i]:=-x[i] else                                  {# # reduction transfer from assigned row}
begin
    j1:=x[i]; min:=inf;
    for j:=1 to n do if j<>j1 then
        if c[i,j]-v[j]<min then min:=c[i,j]-v[j];
    v[j1]:=v[j1]-min
end;

cnt:=0;                                               {# # # # AUGMENTING ROW REDUCTION}
repeat
    k:=1; f0:=f; f:=0;
    while k<=f0 do
    begin
        i:=free[k]; k:=k+1; u1:=c[i,1]-v[1]; j1:=1; u2:=inf;
        for j:=2 to n do
        begin
            h:=c[i,j]-v[j];
            if h<u2 then
                if h>=u1 then begin u2:=h; j2:=j end
                    else begin u2:=u1; u1:=h; j2:=j1; j1:=j end
        end;
        i1:=y[j1];
        if u1<u2 then v[j1]:=v[j1]-u2+u1
            else if i1>0 then begin j1:=j2; i1:=y[j1] end;
        if i1>0 then
            if u1<u2 then begin k:=k-1; free[k]:=i1 end
                else begin f:=f+1; free[f]:=i1 end;
        x[i]:=j1; y[j1]:=i
    end;
    cnt:=cnt+1
until cnt=2;                                          {# # routine applied twice}

f0:=f;                                                {# # # # AUGMENTATION}
for f:=1 to f0 do
begin
    i1:=free[f]; low:=1; up:=1;                       {# # initialize d- and pred-array}
    for j:=1 to n do begin d[j]:=c[i1,j]-v[j]; pred[j]:=i1 end;
    repeat
        if up=low then                                {# # find columns with new value for minimum d}
        begin
            last:=low-1; min:=d[col[up]]; up:=up+1;
            for k:=up to n do
            begin
                j:=col[k]; h:=d[j];
                if h<=min then
```

```
                begin
                    if h<min then begin up:=low; min:=h end;
                    col[k]:=col[up]; col[up]:=j; up:=up+1
                end
            end;
            for h:=low to up-1 do
            begin j:=col[h]; if y[j]=0 then goto augment end
        end; {up=low}
        j1:=col[low]; low:=low+1; i:=y[j1];                    {## scan a row}
        u1:=c[i,j1]-v[j1]-min;
        for k:=up to n do
        begin
            j:=col[k]; h:=c[i,j]-v[j]-u1;
            if h<d[j] then
            begin
                d[j]:=h; pred[j]:=i;
                if h=min then
                    if y[j]=0 then goto augment
                        else begin col[k]:=col[up]; col[up]:=j; up:=up+1 end
            end
        end
    until false; {repeat ends with goto augment}
augment:
    for k:=1 to last do                                {## updating of column prices}
        begin j1:=col[k]; v[j1]:=v[j1]+d[j1]-min end;
    repeat                                                        {## augmentation}
        i:=pred[j]; y[j]:=i; k:=j; j:=x[i]; x[i]:=k
    until i=i1
  end; {of augmentation}
  h:=0;                          {#### DETERMINE ROW PRICES AND OPTIMAL VALUE}
  for i:=1 to n do begin j:=x[i]; u[i]:=c[i,j]-v[j]; h:=h+u[i]+v[j] end;
  lapjv:=h
end.
```

Fig. 6. Pascal function for the linear assignment algorithm LAPJV

8. Computational Results

We compare our algorithm LAPJV with several other methods, both on dense and on sparse problems. The computational results are for Fortran codes run on a CDC Cyber 750 (with OPT = 2), but the algorithms were developed in Pascal on personal computers (Apricot PC and F1, Olivetti M24). Running times on these computers are typically between 2 and 4 seconds for full dense problems of size 100, using Borland's Turbo Pascal compiler (version 3.0).

The classical Hungarian code of Silver [27] is an intermediary for a comparison with the shortest path based algorithm of Hung and Rom [18]. They give the ratio of their computing times to those of the Silver code translated into Fortran. Dividing the same ratios for LAPJV by those published by Hung and Rom shows that on problems of full density our algorithm is about twice as fast (Table 2).

Table 2. *Computation times of LAPJV divided by those of Hung and Rom*

problem size	range of cost coefficients		
	1 – 100	1 – 1000	1 – 10000
50	.34	.44	.72
100	.31	.41	.52

We left primal simplex methods out of consideration, as the literature shows that these are outperformed by several other methods. The algorithm of Hung and Rom is "about twice as fast" as that of Barr, Glover and Klingman [2], which is one of the best primal simplex methods. Glover confirmed this in a private communication. Carpaneto and Toth [6] compare the same primal simplex method with their Hungarian code for sparse problems SPASS and with a Tomizawa based method, adapted from a code in Burkard and Derigs [4]. Both algorithms are faster by some margin. Finally, the Hungarian method as implemented by McGinnis [25] is "roughly comparable in solution speed", but in an addendum he improves his method to a much faster one.

The computation times of LAPJV in Table 3 are averages for ten full density problems, compared to those of the algorithms:

- ASSCT: $O(n^4)$ Hungarian method coded by Carpaneto and Toth [5], making extensive use of pointer techniques to locate zero valued reduced costs,
- LSAP: Dorhout's improved version of Tomizawa's algorithm [10, 11] translated into Fortran and published by Burkard and Derigs [4],
- ASSIGN: the algorithm of Bertsekas [3], as made available by the author and adapted for full density costs matrices.

Table 3. *Computation times for assignment problems of full density (in ms)*

cost range	problem size	LAP algorithm			
		ASSCT	LSAP	ASSIGN	LAPJV
1 – 100	50	51	32	22	15
	100	149	168	114	64
	150	283	535	256	179
	200	420	1363	520	323
1 – 1000	50	121	30	24	17
	100	637	165	123	77
	150	1447	456	364	225
	200	2217	850	665	406
1 – 10000	50	145	31	28	25
	100	1085	168	134	103
	150	3562	453	410	259
	200	6989	919	779	456

As usual with pure Hungarian methods, ASSCT turns out to be very sensitive to cost range. Only for small cost ranges the use of pointer techniques makes the algorithm competitive. LSAP performs strangely on the cost range $1-100$ with relatively large computation times, also observed by Derigs and Metz [9]. LAPJV is clearly faster than ASSIGN, and less sensitive to the range of the cost coefficients.

For a comparison on sparse problems we adapted the data structure of LAPJV, yielding LAPJVsp. Average computation times (for ten problems) are compared in Table 4 with those of two algorithms for sparse LAPs:

- SPASS: Lawler's $O(n^3)$ Hungarian method [23] coded by Carpaneto and Toth [6],
- ASSIGN: again Bertsekas' algorithm [3] as provided to us (ASSIGN requires too much memory for LAPs with $n=400$ and 20% density of the costs matrix).

An indirect comparison may be made with the code SPTM from Carraresi and Sodini [7] for sparse LAPs. The SPTM computation times in Table 4 have been obtained by multiplying the SPASS times with the ratios calculated from [7]. ASSIGN and LAPJVsp clearly outperform the Hungarian method. The margin in speed of LAPJVsp over ASSIGN is about the same as on full density problems.

Table 4. *Computation times for sparse assignment problems (in ms) ("." indicates not run or not known)*

density and cost range	problem size	LAP algorithm SPASS	ASSIGN	SPTM	LAPJVsp
5%/1−100	100	65	38	.	19
	200	211	110	67	62
	400	713	451	335	253
5%/1−1000	100	82	50	.	25
	200	361	174	113	81
	400	1553	688	356	333
20%/1−100	100	71	54	.	26
	200	304	290	234	154
	400	1046	.	1029	657
20%/1−1000	100	119	69	.	33
	200	576	384	253	188
	400	2123	.	1039	788

We may also compare results with the shortest augmenting paths algorithms for sparse LAPs of Derigs and Metz [8]. Their best code SAPM3 is faster than SPASS: 40% to 50% on problems with $n=200$, about 5% density, and on cost ranges $1-100$ and $1-1000$. This is substantially slower than LAPJVsp. Consequently, the code LAPJVsp will provide a better basis for an in-core-out-of-core algorithm for full density LAPs as proposed by Derigs and Metz [9]. Such a code solves a sparse version of full density problems that cannot be entirely contained in memory. It contains a procedure to check whether the not considered assignments price out correctly, and a reoptimization procedure for use if they do not. An in-core-out-of-

core code can also be used as an all in-core code. The advantages of solving only sparse problems has to be weighed against the effort to construct the sparse problems. Unfortunately this task is computationally non-trivial, leading to about the same total computation times as for LAPJV.

9. Conclusions

The computational results show that the average computation times of the algorithm LAPJV are uniformly lower than the best of other algorithms. The code is of moderate size, and the memory requirements are small. The algorithm is suited for both dense and sparse assignment problems, and its sensitivity to cost range is relatively low.

References

[1] Balinski, M. L.: Signature methods for the assignment problem. Operations Research *33*, 527–536 (1985).

[2] Barr, R., Glover, F., Klingman, D.: The alternating path basis algorithm for assignment problems. Mathematical Programming *13*, 1–13 (1977).

[3] Bertsekas, D. P.: A new algorithm for the linear assignment problem. Mathematical Programming *21*, 152–171 (1981).

[4] Burkard, R. E., Derigs, U.: Assignment and Matching Problems: Solution Methods with Fortran Programs, pp. 1–11. Berlin-Heidelberg-New York: Springer 1980.

[5] Carpaneto, G., Toth, P.: Algorithm 548 (solution of the assignment problem). ACM Transactions on Mathematical Software *6*, 104–111 (1980).

[6] Carpaneto, G., Toth, P.: Algorithm 50: algorithm for the solution of the assignment problem for sparse matrices. Computing *31*, 83–94 (1983).

[7] Carraresi, P., Sodini, C.: An efficient algorithm for the bipartite matching problem. European Journal of Operational Research *23*, 86–93 (1986).

[8] Derigs, U., Metz, A.: An efficient labeling technique for solving sparse assignment problems. Computing *36*, 301–311 (1986).

[9] Derigs, U., Metz, A.: An in-core/out-of-core method for solving large scale assignment problems. Zeitschrift für Operations Research *30*, 181–195 (1986).

[10] Dorhout, B.: Het Lineaire Toewijzingsprobleem: Vergelijking van Algorithmen. Rapport BN 21/73, Mathematisch Centrum, Amsterdam (1973).

[11] Dorhout, B.: Experiments with some algorithms for the linear assignment problem. Report BW 39, Mathematisch Centrum, Amsterdam (1975).

[12] Dijkstra, E. W.: A note on two problems in connexion with graphs. Numerische Mathematik *1*, 269–271 (1959).

[13] Edmonds, J., Karp, R. M.: Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM *19*, 248–264 (1972).

[14] Ford jr., L. R., Fulkerson, D. R.: Flows in Networks. Princeton: Princeton University Press 1962.

[15] Glover, F., Klingman, D., Phillips, N.: A new polynomially bounded shortest path algorithm. Operations Research *33*, 65–73 (1985).

[16] Glover, F., Klingman, D., Phillips, N., Schneider, R.: New polynomial shortest path algorithms and their computational attributes. Management Science *31*, 1106–1128 (1985).

[17] Goldfarb, D.: Efficient dual simplex algorithms for the assignment problem. Mathematical Programming *33*, 187–203 (1985).

[18] Hung, M. S., Rom, W. O.: Solving the assignment problem by relaxation. Operations Research *28*, 969–982 (1980).

[19] Jonker, R.: Traveling salesman and assignment algorithms: design and implementation. Faculty of Actuarial Science and Econometrics, University of Amsterdam (1986).

[20] Jonker, R., Volgenant, A.: Improving the Hungarian assignment algorithm. Operations Research Letters *5*, 171–175 (1986).
[21] Karp, R. M.: An algorithm to solve the $m \times n$ assignment problem in expected time $O(mn \log n)$. Networks *10*, 143–152 (1980).
[22] Kuhn, H. W.: The Hungarian method for the assignment problem. Naval Research Logistics Quarterly *2*, 83–97 (1955).
[23] Lawler, E. L.: Combinatorial Optimization: Networks and Matroids. New York: Holt, Rinehart & Winston 1976.
[24] Mack, C.: The Bradford method for the assignment problem. New Journal of Statistics and Operational Research *1*, 17–29 (1969).
[25] McGinnis, L. F.: Implementation and testing of a primal-dual algorithm for the assignment problem. Operations Research *31*, 277–291 (1983).
[26] Papadimitriou, C. H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Englewood Cliffs, N.J.: Prentice-Hall 1982.
[27] Silver, R.: An algorithm for the assignment problem. Communications of the ACM *3*, 605–606 (1960).
[28] Tabourier, Y.: Un Algorithme pour le Problème d'Affectation. R.A.I.R.O. Recherche Opérationnelle /Operations Research *6*, 3–15 (1972).
[29] Tomizawa, N.: On some techniques useful for the solution of transportation problems. Networks *1*, 173–194 (1971).

Roy Jonker* and
Ton Volgenant
Faculty of Actuarial Sciences
and Econometrics
University of Amsterdam
Jodenbreestraat 23
1011 NH Amsterdam
The Netherlands

* Current address:
Koninklyke/Shell-Laboratorium
Amsterdam
The Netherlands