ELSEVIER Discrete Applied Mathematics 100 (2000) 17–48

DISCRETE APPLIED MATHEMATICS

Algorithms and codes for dense assignment problems: the state of the art

Mauro Dell'Amico[a,*], Paolo Toth[b]

[a] DISMI, Università di Modena e Reggio Emilia, viale Allegri, 15, 42100 Reggio Emilia, Italy
[b] DEIS, Università di Bologna, v.le Risorgimento, 32, 40136, Bologna, Italy

Received 13 April 1998; received in revised form 31 March 1999; accepted 19 April 1999

Abstract

The paper considers the classic linear assignment problem with a min-sum objective function, and the most efficient and easily available codes for its solution. We first give a survey describing the different approaches in the literature, presenting their implementations, and pointing out similarities and differences. Then we select eight codes and we introduce a wide set of dense instances containing both randomly generated and benchmark problems. Finally we discuss the results of extensive computational experiments obtained by solving the above instances with the eight codes, both on a workstation with Unix operating system and on a personal computer running under Windows 95.

Keywords: Linear assignment problem; Experimental evaluation; Comparison of algorithms; Dense matrices

1. Introduction

Given an $n \times n$ integer cost matrix $[c_{ij}]$, the *Linear Assignment Problem* (AP) is to assign each row to a different column in such a way that the sum of the selected costs is a minimum. Using a binary variable $x_{ij} = 1$ iff row i is assigned to column j, the problem can be formulated as follows:

$$\text{(AP)} \quad z = \min \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{1}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad (i = 1, \ldots, n), \tag{2}$$

* Corresponding author.
E-mail address: dellamico@unimo.it (M. Dell'Amico)

0166-218X/00/$ - see front matter
PII: S0166-218X(99)00172-9

$$\sum_{i=1}^{n} x_{ij} = 1 \quad (j = 1,\ldots,n), \tag{3}$$

$$x_{ij} \in \{0,1\} \quad (i,j = 1,\ldots,n). \tag{4}$$

This is one of the most famous and studied problems in mathematical programming, and it is also a basic topic in combinatorial optimization. Surveys on AP have been presented by Derigs [29], Martello and Toth [44], Bertsekas [15] and Akgül [4], whereas a complete annotated bibliography has recently been proposed by Dell'Amico and Martello [28]. There are more than 100 papers on the problem and several algorithms have been proposed, but, in practice, less than ten efficient codes are available.

The aim of this paper is to present a short survey of the techniques proposed for the solution of AP and to give a complete and extensive computational analysis of the most popular and efficient algorithms. In particular, we consider *sequential* codes available as source listing, either on a diskette accompanying a book, or on the web. Moreover we restrict our study to *dense* instances.

Our first choice (i.e. to consider only sequential codes) is due to the fact that, at present, parallel algorithms are too architecture dependent and two main problems arise when we try to use a code, written for a particular computer, on a different machine. First the translation of the code may be difficult due to the possible strong use of the peculiarities of the architecture. Moreover, even if the translation is carefully made, the two implementations may have very different performances.

Our second choice (i.e. to consider dense instances) is due to the fact that most of the literature presents computational experiments on sparse matrices, although there are important classes of problems which are dense in nature. Consider, e.g., the classic routing problems (Traveling Salesman Problem (TSP), Vehicle Routing Problem (VRP), etc.) which have been traditionally solved using the AP as a subproblem, and which have benchmarks and real-life instances defined by almost full matrices. (When an instance is given by a sparse cost matrix S, we can handle it with a complete matrix $[c_{ij}]$ in which a large positive value (say $+\infty$) is given to each entry (i,j) which does not exist in S.)

Several papers exist which compare algorithms for AP through computational experiments [18,21,45,29,22,37,15,17,7,38,23,51,31,54,53], but this work differs from the previous ones in four main aspects:

- we consider only original codes, implemented by the respective authors, which can be easily obtained;
- we have performed a huge computational analysis on randomly generated and benchmark problems (we consider 730 random instances from six classes, and 141 benchmark instances from the literature);
- we performed our experiments on two of the most common hardware and software platforms: Sun workstation running under Unix System V and a PC Pentium running under Windows 95 operating system;
- we consider only dense instances, which are almost neglected in the literature.

In Section 2 we summarize the main approaches proposed for the solution of AP, then in Section 3 we describe the algorithms we have selected for our analysis and how they implement the general approaches in Section 2. Section 4 describes the modifications to the original codes that have been introduced to perform our experiments, and presents the test instances. The last Section 5 gives the results of our computational experiments and comments on the performances of the competitors.

2. Solution techniques

Before describing the most important techniques proposed for the solution of AP, let us introduce some background material.

It is well known that the constraint matrix defined by (2) and (3) is totally unimodular. Moreover, the right-hand sides of (2) and (3) are integer, so the polyhedron of the feasible solutions of AP has integral vertices, and one can obtain the optimal solution of AP by solving the continuous linear program:

$$\text{C(AP)} \quad z = \min \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$(2),(3)$$

$$\text{s.t.} \quad x_{ij} \geqslant 0 \quad (i,j = 1,\dots,n). \tag{5}$$

By associating dual variables u_i and v_j with constraints (2) and (3), respectively, the corresponding dual problem is

$$\text{D(AP)} \quad w = \max \quad \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \tag{6}$$

$$u_i + v_j \leqslant c_{ij} \quad (i,j = 1,\dots,n). \tag{7}$$

Let $\bar{c}_{ij} = c_{ij} - u_i - v_j$ $(i,j = 1,\dots,n)$ be the *reduced costs* of C(AP). Given a pair of solutions x and (u,v), respectively feasible for the primal and for the dual problems, the *optimality conditions* (or *complementary slackness*) are

$$x_{ij}\bar{c}_{ij} = 0 \quad (i,j = 1,\dots,n). \tag{8}$$

The assignment problem is also known as the *Weighted Bipartite Matching Problem*. Let $\bar{G} = (U \cup V, \bar{E})$ be a bipartite graph with node sets $U = V = \{1,2,\dots,n\}$, edge set $\bar{E} = \{[i,j]\colon i \in U,\ j \in V,\ c_{ij} < \infty\}$ and costs $[c_{ij}]$ associated with the edges. The problem is then to find a perfect matching of minimum cost on $\bar{G}$. It can be shown that each feasible basis of C(AP) induces a *spanning tree* on $\bar{G}$.

Most of the algorithms for AP have a 'dual' nature, that is, they build the optimal solution step-by-step, by iteratively adding assignments to a current partial primal solution (i.e. a solution in which less than n variables is assigned value one, and (2)–(3) are satisfied with the '=' sign substituted by the '$\leqslant$' sign). These techniques usually consist of two phases: in the first phase (*preprocessing*) a primal partial solution and a dual feasible solution are determined which satisfy the complementary slackness conditions (8). In the next phase the primal solution is improved by adding one row–column assignment at a time, until the solution becomes feasible. At each step of this phase the dual solution is updated so that the complementary slackness still holds. At the end of the phase the current primal–dual (solution) pair is optimal. A simple way of implementing the preprocessing phase is to determine a dual feasible solution as follows:

$$\begin{aligned} v_j &= \min\{c_{ij}\colon i=1,\dots,n\} \quad (j=1,\dots,n) \text{ (column reduction)},\\ u_i &= \min\{c_{ij}-v_j\colon j=1,\dots,n\} \quad (i=1,\dots,n) \text{ (row reduction)}. \end{aligned} \tag{9}$$

Then a primal partial solution is determined by selecting (in some order) assignments (i,j) such that row i and column j are currently unassigned and $\bar{c}_{ij}=0$.

The approaches proposed for the solution of AP can be grouped into three classes: primal–dual algorithms (based on the identification of shortest paths), pure primal algorithms, and pure dual algorithms. In the next subsections we briefly describe the three approaches. Note that we do not intend to be completely exhaustive, but we give only some hints on the methods, and we refer the reader to the appropriate literature for a complete and rigorous description. In particular our description emphasizes the algorithmic aspects of the various approaches, but does not give a proof of their correctness.

2.1. Primal–dual (Shortest Path) algorithms

Using the primal–dual approach, Kuhn [39,40] obtained the first polynomial method for the solution of AP, called the *Hungarian* method. The approach can be summarized as follows (see e.g. [48] for a complete description):

(0) Determine a dual feasible solution (u,v) by using row and column reduction.

(i) Given the solution (u,v) solve the restricted primal problem. This is equivalent to finding a maximum cardinality matching on the bipartite subgraph $G'=(U\cup V,E')$, where $E'=\{[i,j]\in E\colon \bar{c}_{ij}=0\}$. Let X be the set of edges in the optimum matching, $\hat{R}\subseteq U$ and $\hat{C}\subseteq V$ be the nodes incident to an edge in X, and define a solution $\bar{x}$ with $\bar{x}_{ij}=1$ if $[i,j]\in X$, $\bar{x}_{ij}=0$ otherwise.

(ii) if $\bar{x}$ is primal feasible (i.e. $|X|=n$), then stop (an optimal primal–dual pair $(\bar{x},(u,v))$ has been found), otherwise obtain a new dual solution by setting $\overline{u}_i=u_i+\delta$ for all $i\in\hat{R}$, $\bar{v}_j=v_j-\delta$ for all $j\in\hat{C}$, where δ is the minimum reduced cost $\bar{c}_{ij}=c_{ij}-u_i-v_j$ among those with $i\in\hat{R}$ and $j\in\hat{C}$. Set $u=\overline{u}$, $v=\bar{v}$ and go to step (i) (note that it is not necessary to recompute the maximum cardinality matching from scratch, but only to reoptimize the existing one).

It is easy to see that the primal–dual pair determined at step (i) satisfies the complementary slackness conditions (8), so if $\bar{x}$ is primal feasible then the pair is optimal. When a new dual solution is obtained (step (ii)), the complementary slackness still holds for the new pair $(\bar{x},(\overline{u},\bar{v}))$, the dual solution is feasible and at least one pair (i,j) with $i \in R$, $j \in C$ exists such that $c_{ij} - \bar{u}_i - \bar{v}_j = 0$, but $c_{ij} - u_i - v_j > 0$. Therefore, at the next execution of step (i), graph G' has at least one new edge. With the original implementation an $O(n^3)$ time is required to perform steps (ii) and (iii) until a new assignment is identified, hence the overall algorithm runs in $O(n^4)$ time.

The complexity of the Hungarian method was reduced to $O(n^3)$ by Lawler [41]. Lawler's implementation was shown (see [29]) to be equivalent to a successive shortest path algorithm which can be conveniently described by using the weighted bipartite matching model. Given the graph $\bar{G}$ of Section 2, a partial primal solution x and a dual solution (u,v), set $X = \{[i,j] \in \bar{E}: x_{ij} = 1\}$ and define a new bipartite digraph $G = (U \cup V, A)$ with arc set $A = D \cup R$, where $D = \{(i,j): i \in U,\ j \in V,\ [i,j] \in E \setminus X\}$ is the set of the *direct* arcs, and $R = \{(i,j): i \in V,\ j \in U,\ [j,i] \in X\}$ is the set of the *reverse* arcs. Each arc $(i,j) \in A$ is assigned the cost $\bar{c}_{ij}$ if $(i,j) \in D$ and zero if $(i,j) \in R$. (Note that solutions x and (u,v) satisfy the complementary slackness conditions.) Let us call 'unassigned' a node of U or V corresponding, respectively, to an unassigned row or column. One can prove that any dipath of G starting from an unassigned node of U contains, alternatively, an arc in D and an arc in R. Such paths are called *alternating paths*. If the dipath, say P, terminates with an unassigned node of set V, then it is called an *augmenting path*. Indeed, by removing from X the edges in $R \cap P$ and adding to X the edges in $D \cap P$, we obtain a new (partial) primal solution X' with $|X'| = |X| + 1$ assignments. Remembering that the costs of the arcs are the reduced costs of C(AP) one can see that finding a shortest (augmenting) path in G is equivalent to finding a minimum cost solution with $|X| + 1$ assignments. Since the reduced costs are non-negative, then the required path can be determined through Dijkstra's algorithm by selecting as root node an unassigned node of U. The growth of the Dijkstra tree, say $T = (V^T, A^T)$, is halted when it reaches an unassigned node, say j, of V, i.e. when it contains an augmenting path from the root to the leaf j. The dual solution associated with X' is obtained by defining $\delta = \min\{\bar{c}_{ij}: (i,j) \in A^T \cap D\}$, by setting $\hat{R} = U \cup V^T$, $\hat{C} = V \cup V^T$, and by updating the current dual solution as in step (ii) above. AP can thus be solved by identifying $O(n)$ successive shortest augmenting paths. Since the Dijkstra algorithm runs in $O(n^2)$ time, the overall computational complexity of a shortest path algorithm is $O(n^3)$.

This reduction of the time complexity is not surprising. Indeed, one can observe that each shortest augmenting path corresponds to a series of steps (i) and (ii) of the Hungarian method, which lead from a solution with $|X|$ assignments to a new one with $|X| + 1$ assignments. But the original Hungarian method needs $O(n^3)$ times to add an assignment, whereas a shortest path can be computed in $O(n^2)$ time.

At present all the efficient algorithms proposed in the literature and based on shortest paths, have $O(n^3)$ time complexity when applied to dense instances. The various algorithms differ in two points: (a) the preprocessing procedure used to define the first primal–dual pair, and (b) a possible sparsification technique. Sparsification is used by some algorithms to try to reduce the average computing time. In a first phase a core problem CP is defined by selecting a subset of entries from matrix C. Then

CP is solved giving an optimal primal–dual pair and a check is performed to determine if the primal–dual pair is optimal also for the complete instance (i.e. if the reduced costs are non-negative for all the elements of matrix C). If the solution is not optimal the core is enlarged by adding other entries and the procedure is repeated.

The shortest path algorithms we have used for our experiments were described in [37,22,20,15,53]. Both points (a) and (b) above will be discussed in detail in the next section.

2.2. Primal algorithms

The primal algorithms proposed for AP are basically specialized implementations of the network simplex algorithm. We have already recalled that a basis of the continuous relaxation of AP is a spanning tree, say T, of graph $\bar{G}$. A pivot operation performed by the simplex method induces a transformation of the tree T: an edge $e \in E \setminus T$ having-negative reduced cost is added to T, and an appropriately chosen edge is removed from the unique circuit of $T \cup \{e\}$. It is well known that the simplex algorithm is not polynomial in the input size, but primal algorithms exist for AP, which run in polynomial time. The key idea of reducing the time complexity of primal algorithms was independently introduced by Barr et al. [11] and Cunningham [26], and consists of considering as possible candidates only a particular subset of the bases called *alternating path bases* or *strongly feasible trees* (SFT). These bases correspond to trees of $\bar{G}$ with the following characteristics: (a) the root node belongs to U and has degree one (as usual the degree of a node is the number of edges incident to it); (b) all other nodes belonging to U, but the root, have degree two; (c) $x_{ij} = 1$ for each edge $[i,j]$ with $i \in U$, $j \in V$; (d) $x_{ij} = 0$ for each edge $[i,j]$ with $i \in V$, $j \in U$. Using SFT, Hung [34] developed an algorithm which runs in $O(n^3 \log \Delta)$ time, where Δ is the difference between the initial and final solution values. Orlin [46] gave the first strongly polynomial primal algorithm which, for dense matrices, runs in $O(n^4 \log n)$ time. Improved algorithms were presented by Ahuja and Orlin [1] and Akgül [5]. In particular, Akgül's algorithm has an $O(n^3)$ computing time on dense instances, so giving the same time complexity of the primal–dual algorithms. Unfortunately no efficient code has been devised from these results.

2.3. Dual algorithms

Most of the approaches proposed for the solution of AP obtain a primal feasible solution only at the last step, so they could be classified as 'dual' algorithms. In order to simplify the presentation we have already described the shortest path algorithms, which indeed have a dual nature, so in this section we describe four main approaches which can be identified, respectively, as *signature*, *auction*, *pseudoflow* and *interior point* methods.

2.3.1. Signature method

Before describing this method it is necessary to observe that given a tree T of graph $\bar{G}$ we can associate both a primal and a dual solution of C(AP), with it. (A dual solution can be determined by assigning value zero to the dual variable associated with the root node and then iteratively assigning the other dual values so that the reduced cost of each edge of the tree is zero.) If the dual solution is feasible (hence the primal solution is not feasible, unless T corresponds to an optimal solution) we call this tree a *dual feasible tree*.

The method defines a *signature* of a dual feasible tree the vector of the degrees of the nodes of T which belong to U. Using the signatures, Balinski [8] uniquely identifies the extreme points of the dual polyhedron and shows that any two extreme points are joined by a path of at most $(n-1)(n-2)/2$ extreme edges (i.e. he has proved that the Hirsch conjecture holds). Subsequently (see [9]) he obtained a dual polynomial algorithm for AP which runs in $O(n^3)$. This algorithm performs pivot operations to transform a basis into an adjacent one, but it cannot be considered an implementation of a dual simplex method since it may pivot on an edge with zero or positive flow. Genuinely dual simplex algorithms were proposed by Balinski [10] and Akgül [2,3]. Both algorithms use signatures and the idea of restricting the set of the basis to be considered to the so-called *dual strongly feasible trees*.

Up to now no efficient code which implements these techniques is available.

2.3.2. Auction method

The *auction* method was introduced for the first time in [14], where a pseudo-polynomial algorithm for AP was presented. Subsequently, the method was improved through a scaling technique (see [16]) giving an algorithm which runs in $O(n^3 \log(n\Delta))$, for dense instances, where Δ is the maximum $|c_{ij}|$ value. In the following, we briefly describe the original technique and its improvement. Note that this method has usually been presented for the maximization version of AP, but for congruence with the rest of the paper we describe its application to a minimization problem.

Consider the dual problem D(AP) and observe that, given a dual vector v, the associated optimal vector u is

$$u_i = \min\{c_{ij} - v_j : j = 1, \ldots, n\} \quad (i = 1, \ldots, n). \tag{10}$$

Thus D(AP) is equivalent to the unconstrained problem

$$\max q(v) \tag{11}$$

where $q(v) = \sum_{i=1}^{n} \min_j (c_{ij} - v_j) + \sum_{j=1}^{n} v_j$. Given a row index i, let us define as

$$j(i) = \operatorname{argmin}\{c_{ij} - v_j : j = 1, \ldots, n\} \tag{12}$$

the column index associated with the minimum $c_{ij} - v_j$ value of row i. Note that if the dual vector u is defined as in (10) and we consider the assignment $x_{i,j(i)} = 1$, for $i = 1, \ldots, n$ (not necessarily feasible for AP), then the complementary slackness conditions (8) are satisfied. During its execution, an auction algorithm maintains a triple

$(x,(u,v))$ which is dual feasible and satisfies the complementary slackness conditions. At each iteration the dual vector v is updated and an optimal solution to AP is obtained when v can be associated, through (12), to a primal feasible solution (i.e. $j(i) \neq j(k)$ for $i,k=1,\dots,n,\ i \neq k$).

The algorithm maintains a set S of assigned rows (initially empty) and, at each iteration, it selects a row $\bar{\imath} \notin S$ and performs the following steps. It computes the first and the second minimum of the quantities $c_{\bar{\imath}j} - v_j$, i.e. the value $u_{\bar{\imath}}$ (see (10)), and the value

$$u''_{\bar{\imath}} = \min\{c_{\bar{\imath}j} - v_j,\ j=1,\dots,n,\ j \neq j(\bar{\imath})\}. \tag{13}$$

If $u_{\bar{\imath}} < u''_{\bar{\imath}}$ or $u_{\bar{\imath}} = u''_{\bar{\imath}}$ and $j(\bar{\imath}) \neq j(i)$, $\forall i \in S$ then the current $v_{j(i)}$ value is decreased by the quantity $|u_{\bar{\imath}} - u''_{\bar{\imath}}|$, row $\bar{\imath}$ is added to S, and the row $\hat{\imath} \in S$ such that $j(\hat{\imath}) = j(\bar{\imath})$, if any, is removed from S. If otherwise $u_{\bar{\imath}} = u''_{\bar{\imath}}$ and a row $\hat{\imath} \in S$ exists such that $j(\hat{\imath}) = j(\bar{\imath})$, then the algorithm performs a labeling procedure, like that of the Hungarian method, which either finds an augmenting path with zero reduced cost (so a new row is assigned), or it determines a value δ to be subtracted from the dual values associated with the labeled columns (so a new vector v is defined). In the first case, set S is updated according to the augmenting path found, whereas in the second case we set $S = S \backslash \{\hat{\imath}\} \cup \{i\}$. The algorithm terminates when $|S| = n$. For dense instances the algorithm runs in $O(n^3 + n^2 \Delta)$ time (where Δ is again the maximum $|c_{ij}|$ value).

The improved algorithm uses the following relaxed version of the complementary slackness conditions. Given a primal–dual pair and a value $\varepsilon > 0$, conditions (8) are considered to be satisfied if

$$\bar{c}_{ij} \leqslant \varepsilon \quad \forall i,j : x_{ij} = 1. \tag{14}$$

This is called an *ε-relaxation* of the problem. The algorithm starts with a large ε value and determines an optimal primal–dual pair for the ε-relaxed problem, then it reduces the value of ε and reoptimizes the solution. It is possible to show that a primal–dual pair is optimal for AP when $\varepsilon < 1/n$. The optimal ε-relaxed primal–dual pair is determined by means of a method similar to the above one.

First a list $L = \{1,\dots,n\} \setminus S$ containing all the unassigned rows is defined. Then a *bidding phase* is performed by computing, for each row $\bar{\imath} \in L$, the value $b(\bar{\imath}) = v_{j(\bar{\imath})} + u_{\bar{\imath}} - u''_{\bar{\imath}} - \varepsilon$ (called *bidding* of row $\bar{\imath}$ for column j). In a subsequent *assignment phase* the algorithm sets $v_{j(\bar{\imath})}$ to $b(\bar{\imath})$ (observe that this updating preserves the ε-slackness conditions), removes row $\bar{\imath}$ from L and adds $\bar{\imath}$ to set S. If a row $i \in S$ exists such that $j(i) = j(\bar{\imath})$ then i is removed from S and added to a second list $L2$, initially empty.

When all rows of L have been examined, if $|L2|$ is larger than a given threshold value, then the algorithm sets $L = L2$, $L2 = \emptyset$ and repeats the above procedure. Otherwise ($|L2|$ is small) if $\varepsilon < 1/n$ then the current solution is optimal, else ε is reduced, L is defined again as $\{1,\dots,n\} \setminus S$ and the procedure is repeated.

This implementation of the auction method is known as the "Gauss–Seidel version". In a different implementation, called the "Jacobi version", the bidding $b(i)$ is computed for all unassigned rows, instead of that for a single row, then the dual value v_j of each column j which received a bid is updated. The Jacobi version is more efficient for parallel implementations, whereas the Gauss–Seidel version is superior for sequential implementations.

The auction algorithms we have used for our experiments implement the Gauss–Seidel version and were described in [15].

2.3.3. Pseudoflow method

Given a digraph $\hat{G} = (\hat{V}, \hat{A})$ and a capacity $b(i,j) > 0$ for each arc $(i,j) \in \hat{A}$, a *pseudoflow* is a function $f : \hat{A} \to \Re^+$ satisfying $f(i,j) \leqslant b(i,j)\ \forall (i,j) \in \hat{A}$. For each pseudoflow f and node $k \in \hat{V}$ the *excess flow* into k is defined as $e(k) = \sum_{(i,k) \in \hat{A}} f(i,k) - \sum_{(k,j) \in \hat{A}} f(k,j) + d(k)$, where $d(k)$ is the *supply* of node k ($d(k)$ is positive if k is a source, negative if k is a sink, and null, otherwise). If $e(k) = 0$ for each $k \in \hat{V}$ the pseudoflow f is called a *flow*.

Given an instance of AP defined by a bipartite graph $\bar{G} = (U \cup V, \bar{E})$ (see Section 2) and a pseudoflow f, we can define a bipartite digraph $G = (U \cup V, A)$, with arc set $A = D \cup R$. The definition of sets D and R is similar to that in Section 2.1: D contains direct arcs, i.e. arcs directed from U to V and R contains reverse arcs, i.e. arcs from V to U. More precisely, $D = \{(i,j)\colon i \in U,\ j \in V,\ [i,j] \in \bar{E},\ f(i,j) < 1\}$ and $R = \{(j,i)\colon j \in V,\ i \in U,\ [i,j] \in \bar{E},\ f(i,j) > 0\}$ (note that a single edge $[i,j] \in \bar{E}$ with $0 < f(i,j) < 1$ produces two arcs in A). With each direct arc $(i,j) \in D$ we associate a capacity $b(i,j) = 1$ and a cost c_{ij}, whereas with each reverse arc $(j,i) \in R$ we associate a capacity $b(j,i) = 1 - f(i,j)$ and a cost $-c_{ij}$. Finally, the supply function is given value 1 for each node $k \in U$ and value -1 for each node $k \in V$.

The pseudoflow method uses a cost scaling technique to determine, by successive approximations, an optimal solution to AP. Given a value $\varepsilon > 0$, the algorithm sets $f(i,j) = 0\ \forall (i,j) \in A$ (i.e. it defines a zero pseudoflow) and transforms this pseudoflow into a flow which is optimal for the ε-relaxation of the problem (see (14)). Then the value of ε is reduced and a new ε-optimal flow is determined. The procedure is iterated until $\varepsilon < 1/n$, which guarantees the optimality of the flow for the original problem (see Section 2.3.2).

The method used to convert a pseudoflow into an ε-optimal flow uses two main operations: *push* and *relabel*. The *push* operation is applied to an arc $(i,j) \in A$ to increase the flow on the arc by one unit (note that since the maximum capacity of an arc is one, then *push* can be applied only to arcs with zero flow). After a *push* the capacity is saturated, therefore the arc is removed from A and substituted with its opposite (j,i). The *relabel* operation is applied to a node k to change the value of its dual variable preserving the ε-optimality. If $k \in U$ then u_k is set to $\min_{(k,j) \in A}\{c_{kj} - v_j\}$, if instead $k \in V$ then the value of v_k is set to $\max_{(k,i) \in A}\{c_{ik} - u_i - \varepsilon\}$. Given the dual variables v_j $(j = 1, \ldots, n)$, let us call a *scaling phase* an iteration of the algorithm which defines $u_i = \min_{(i,j) \in A}\{c_{ij} - v_j\}\ \forall i \in U$, sets the initial pseudoflow to zero and applies a series of push and relabel operations, to determine an ε-optimal flow. It is possible to show (see, e.g. [33]) that during a scaling phase:

(a) the values of the dual variables u monotonically increase, whereas the values of the dual variables v monotonically decrease;

(b) for each $i \in U$, the value u_i increases by $O(n\varepsilon)$ and for each $j \in V$, the value v_j decreases by $O(n\varepsilon)$.

If we call again Δ the maximum $|c_{ij}|$ value, and we assume that the value of ε is divided by α at each scaling phase, then the maximum number of scaling phases is $1 + \lfloor \log_\alpha(n\Delta) \rfloor$.

Using the above method, Orlin and Ahuja [47] and Goldberg et al. [32] independently developed algorithms which solve AP on sparse graphs with m edges in $O(\sqrt{n}m \log(n\Delta))$ computing time. In particular, the Orlin and Ahuja algorithm can be seen as a hybrid of the auction algorithm and the shortest path algorithm. We will discuss in the next section some similarities of the algorithms based on pseudoflow, shortest path and auction.

In our experiments, we have used the pseudoflow-based algorithm described in [31], which runs in $O(nm \log(n\Delta))$ computing time.

2.3.4. *Interior point method*

Since any extreme point of the polyhedron of AP is integral, then all methods developed for the solution of a continuous linear problem can be applied to AP. This is also the case of the interior point method. However, to our knowledge, there is only one algorithm that solves AP by means of this technique (see [49]). Computational experiments with that code were presented in [31], where it is shown that the approach is not competitive, therefore we have tested no interior point method.

3. The competitors

We have selected and tested the eight most popular and easily available codes for AP. In Table 1 we give the acronym we use to identify the algorithm, a pointer to the literature, the nature of the method implemented (we indicate by SP the shortest path method, by AU the auction method and by PF the pseudoflow method), and the language used for the original implementation (we indicate with FOR, PAS and C, the languages FORTRAN, Pascal and C, respectively).

Four algorithms are pure shortest path methods, one is a mixture of auction and shortest path technique, two are implementations of a pure auction technique and the last one is a pseudoflow-based algorithm.

The FORTRAN source code of algorithm APC is available in the diskette accompanying the book by Simeone et al. [52]. The codes JV and CTCS are widespread diffused and are available as pseudocode listing in papers [37] and [22], respectively, or directly from the authors (Tom Volgenant, E-mail: `tonv@fee.uva.nl` and Paolo Toth, E-mail: `ptoth@deis.unibo.it`). The code LAPm is available as a pseudocode in [53], or as a Pascal listing from the author. The FORTRAN codes NAUC, AFLP

Table 1
The competitors

acronym	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
reference	[20]	[22]	[37]	[53]	[15]	[15]	[15]	[31]
method	SP	SP	SP	SP	SP+AU	AU	AU	PF
language	FOR	FOR	PAS	PAS	FOR	FOR	FOR	C

and AFR are contained in the diskette accompanying the book by Bertsekas [15] and are available at the URL: `http://www.mit.edu/people/dimitrib/home.html`.

The C language source code of algorithm CSA by Goldberg and Kennedy [31] can be obtained as a **tar** `uuencoded` file by sending an empty E-mail message, with subject `send csas.tar`, to: `ftp-request@theory.stanford.edu`.

3.1. Algorithm APC

This is a pure shortest path algorithm preceded by a simple initialization procedure. A column reduction is first performed (see (9)) and a partial assignment is determined by scanning one column j at a time and by setting $x_{ij}=1$ if c_{ij} is the minimum value of the column and row i is unassigned. Then a row reduction is performed (see (9)) and an attempt is made to enlarge the partial assignment. For each row $i=1,\dots,n$, if column $j(i)$ (see (12)) is unassigned, then $x_{i,j(i)}$ is set to one, otherwise the row $\hat{\imath}$ such that $x_{\hat{\imath},j(i)}=1$ is scanned. If an unassigned column j such that $c_{\hat{\imath},j}-v_j=c_{\hat{\imath},j(i)}-v_{j(i)}$ is found, then the partial assignment is improved by setting $x_{i,j(i)}=1$, $x_{\hat{\imath},i)}=0$ and $x_{\hat{\imath},j}=1$. The last improvement corresponds to performing a labeling phase like that in the Hungarian method restricted to alternating paths of length two.

3.2. Algorithm CTCS

The initialization phase of algorithm CTCS is the same as that of algorithm APC. If the number of assignments in the partial primal solution is smaller than $0.6n$, then the algorithm completes the solution with a standard shortest path technique, thus it operates exactly as APC. If, instead, the partial solution contains at least $0.6n$ assignments, then a sparse matrix $\hat{C}$ is obtained by heuristically selecting a subset of elements from matrix C. The following shortest path phase operates on matrix $\hat{C}$ by using an implementation of APC which works for sparse matrices. Due to the sparsification it may happen that no feasible solution exists for $\hat{C}$. In this case it is necessary to add more elements of C to $\hat{C}$ and to continue to search for a complete solution, with the updated matrix. If, instead, a primal–dual pair which is optimal for $\hat{C}$ is found, it is necessary to check if the dual solution is feasible for the complete matrix C. If not, for each pair (i,j) such that $\bar{c}_{ij}<0$ the assignments of row i and column j are removed, and entry c_{ij} is added to $\hat{C}$ (note that, for the sparse matrix $\hat{C}$, due to the optimality of the corresponding dual solution, an entry such that $\bar{c}_{ij}<0$ cannot exist). If, for a given number of iterations, no feasible solution has been found on $\hat{C}$, or the solutions found are not optimal for

C, then the algorithm discards the sparse matrix, and completes the solution by means of the standard shortest path method.

To complete the description of the algorithm we have to specify how the elements of matrix $\hat{C}$ are selected. Given a parameter σ (set to ten in the original code) the algorithm considers the $\sigma+1$ columns $1, \lfloor n/\sigma \rfloor, 2\lfloor n/\sigma \rfloor, \ldots, \sigma\lfloor n/\sigma \rfloor$ and determines the average value, say μ, of the elements in these columns. Then it computes the threshold $\theta = \lfloor 0.5 + 2\mu \log_{10} n/\alpha \rfloor$, where α is the number of assignments in the initial partial solution. Matrix $\hat{C}$ is given by the elements of matrix C with $c_{ij} \leqslant \theta$.

3.3. Algorithm JV

This algorithm, originally named LAPJV, is one of the shortest path algorithms which has received most attention in the literature. The peculiarity of the algorithm is the massive use of preprocessing procedures to determine the first primal–dual pair. With this algorithm the preprocessing is the most time-consuming phase, but usually the resulting primal partial solution has a large number of assignments, so a few shortest paths are needed to complete the solution.

Algorithm JV first performs a column reduction and determines the corresponding partial solution, as done by algorithm APC (see above). Then it executes a so called *reduction transfer* procedure, which closely resembles the original auction method (see [14]). For each unassigned row i the values u_i and u_i'' are computed (see (10) and(13)) and $v_{j(i)}$ is reduced to $v_{j(i)}-(u_i''-u_i)$ (the aim of this updating is to make the assignment of row i to a column easier, by imposing that the minimum reduced cost of row i is achieved at two columns).

The second procedure, called *augmenting row reduction* (ARR) performs a series of updating of the dual variables v, which are again close to those made by the auction method. Let us define $j''(i)$ as a column such that $u_i'' = c_{i,j''(i)} - v_{j''(i)}$, and let $r(j)$ be the row currently assigned to column j (with $r(j)$ empty if column j is unassigned). For each unassigned row i, ARR computes the values u_i and u_i'' and updates the dual variables according to the following two cases: $u_i < u_i''$ or $u_i = u_i''$.

In the first case ($u_i < u_i''$), $v_{j(i)}$ is reduced to $v_{j(i)} - (u_i'' - u_i)$ and row i is assigned to column $j(i)$. If $r(j(i))$ is empty, then the procedure starts a new iteration by considering a new unassigned row. Otherwise ($r(j(i)) > 0$) the assignment of row $r(j(i))$ to column $j(i)$ is removed and the procedure starts a new iteration with the (now) unassigned row $r(j(i))$.

In the second case ($u_i = u_i''$), if one of the two columns $j(i)$ and $j''(i)$ is unassigned, then row i is assigned to the unassigned column, and ARR continues with a new unassigned row. Otherwise row i is assigned to column $j''(i)$, the assignment of row $r(j''(i))$ to column $j''(i)$ is removed, and the procedure continues with the unassigned row $r(j''(i))$.

It is worth noting that a series of executions of procedure ARR, starting with different unassigned rows, can be seen as a particular implementation of an auction phase, without the ε-relaxation. Jonker and Volgenant have shown that an algorithm which iteratively applies procedure ARR, finds an optimal solution to AP in $O(n^3\Delta)$ time, where Δ is again the maximum $|c_{ij}|$ value. In the original implementation of algorithm JV, procedure ARR is repeated twice, then the partial solution is completed through shortest paths.

The shortest path phase has been implemented with particular care and several tricks have been adopted to accelerate the search of the shortest paths.

3.4. Algorithm LAPm

Similar to algorithm CTCS above, procedure LAPm (originally named LAPMOD) works with a sparse matrix. Given a parameter τ depending on n, for each row i, LAPm includes the values $c_{i,1}, c_{i,2}, \ldots, c_{i,\tau}$ in the sparse matrix. Then it examines the remaining entries of row i looking for a value c_{ij} smaller than the average values of the τ entries currently selected. If such an entry exists, one of the entries already selected is substituted with the new entry, and the search continues. The algorithm also provides for the entry (i,i) to be included in the sparse matrix $\hat{C}$, possibly with a very large cost (say $+\infty$), thus ensuring that a feasible solution always exists for $\hat{C}$. The resulting instance is solved through an implementation of algorithm JV which works with sparse matrices. When a primal–dual pair optimal for $\hat{C}$ has been obtained, the same method used for CTCS is applied to check if the dual solution is feasible for the full instance. If the solution is unfeasible, the sparse matrix is enlarged (again with the same technique used for CTCS) and the shortest augmenting path phase of algorithm JV is repeated.

3.5. Algorithm NAUC

The original name of this algorithm, presented in [15], was NAUCTION_SP which stands for "naive auction and sequential shortest path" algorithm. The author describes the code as follows.

"This code implements the sequential shortest path method for the assignment problem, preceded by an extensive initialization using the naive auction algorithm. The code is quite similar in structure and performance to a code of the author [14] and to the code of Jonker and Volgenant [36,37]. These codes also combined a naive auction initialization with the sequential shortest path method."

In practice, the algorithm performs a prefixed number of auction cycles, each of which is similar to procedure ARR of algorithm JV. The number of cycles is defined as a function of the sparsity of the matrix and, for dense instances, it is equal to two. After the auction phase, the partial solution is completed by means of shortest paths.

3.6. Algorithms AFLP and AFR

We have used two different implementations of the auction method. All the algorithms we have considered up to now are implemented using integer variables and performing operations with integer arithmetic (which are faster than floating point operations). To implement the scaling version of the auction method one has two possibilities.

The first one is to use real variables so that it is possible to manage directly values of ε smaller than one: this is the method used by code AFLP (the acronym stands for Auction with FLoating Point variables).

The second possibility is to multiply all data by a constant K such that the values assumed by ε, after the scaling, are larger than one. In this case it is possible to use integer variables (note that the number of significant decimal digits of ε is given by the order of magnitude of K, i.e. if $K = O(10^\alpha)$, α decimal digits from ε are significant). Unfortunately this technique reduces the set of instances to which the algorithm can be applied. Indeed, calling M the largest integer value representable with an integer variable, the method solves only instances with $\max_{i,j}(c_{ij}) \leqslant M/K$, instead of instances with $\max_{i,j}(c_{ij}) \leqslant M$, as for the other algorithms.

The code AFR we tested uses integer variables and a *forward/reverse* technique (the acronym AFR stands for Auction with Forward/Reverse). In the previous section we have described the forward version of the auction code. The reverse version consists in applying the method to the transposed matrix, i.e. the problem to be solved is

$$\max q'(u) \tag{15}$$

where $q'(u) = \sum_{j=1}^{n} \min_i (c_{ij} - u_i) + \sum_{i=1}^{n} u_i$. In the forward/reverse implementation, the algorithm alternatively performs forward and reverse cycles. The switching is controlled by a function of the current number of assignments in the partial solution. For a more precise description of this algorithm we again use the words of the author.

"This code implements the forward/reverse auction algorithm with ε-scaling for symmetric n by n assignment problems. It solves a sequence of subproblems and decreases ε by a constant factor between subproblems. This version corresponds to a Gauss–Seidel mode and solves ε subproblems inexactly. The code is an improved version of an earlier (September 1985) auction code with ε-scaling written by Dimitri P. Bertsekas".

3.7. Algorithm CSA

In [31] Goldberg and Kennedy presented several implementations of the pseudoflow algorithm. After extensive computational experiments they conclude that their implementation called CSA-Q is best overall, so we have used this code for our tests.

CSA-Q uses the *double-push* method which consists of performing a pair of *push* operations, in sequence. More precisely, given an unassigned node $i \in U$, then *double-push*(i) determines the first and the second arc with smallest reduced costs, among those emanating from i, say (i,j) and (i,k), respectively. Then it sends a unit of flow through arc (i,j) (i.e. assigns i to j) and, if column j was assigned to a row $r(j)$, it performs a second push operation by sending a unit of flow through the reverse arc $(j, r(j))$ (thus removing the assignment $(r(j), j)$). Lastly, the dual value u_i is set to $c_{ik} - v_k$ and the dual value v_j is set to $c_{ij} - c_{ik} + v_k - \varepsilon$. One can observe the near equivalence between the double-push operation and the application of an auction step made by a bidding phase followed by an assignment phase (see Section 2.3.2). Indeed, given a row i, both methods define the same dual value for the column j associated with the minimum reduced cost of row i, and assign/deassign the same elements. The only differences are in the way the calculations are performed and in the computation of the dual values u, which are explicitly made by CSA-Q, whilst the auction method takes care of them implicitly.

A second peculiarity of code CSA-Q is the use of the so-called *fourth-best heuristic* to speed up the search. At the beginning of the algorithm, for each row i, the four smallest partial reduced costs $c_{ij} - v_j$ are determined and the largest of these four costs is stored in $K(i)$. When the algorithm needs to compute the first and the second smallest reduced cost of a row, the search is performed only among the four costs previously identified. Since the values of the dual variables v monotonically decrease, then the partial reduced costs $c_{ij} - v_j$ strictly increase, hence it is necessary to compute again the four smallest partial reduced costs only when all, except possibly one, of the saved elements have partial reduced cost greater than $K(i)$.

4. Codes and test instances

In this section we describe in detail how we have used the original codes introduced in Section 3. Moreover we introduce the classes of instances used to test the codes.

4.1. Adapting the original codes

The original codes we considered are written in three different languages: FORTRAN, C and Pascal. While compilers for the first two languages are available on most hardware platforms, this is not true for the Pascal language (especially under the Unix System). Therefore we have performed a one-to-one translation of the codes JV and LAPm from the original Pascal version to the FORTRAN language. Implementing the FORTRAN version we have adopted some shrewdness to obtain a code that has the same efficiency as the original Pascal code. In particular, we have swapped the row and column indices. Indeed, in Pascal a matrix is stored 'by rows' (i.e. two elements $c_{i,j}$ and $c_{i,j+1}$ are stored in adjacent memory positions), whereas in FORTRAN a matrix is stored 'by columns' (i.e. the two elements $c_{i,j}$ and $c_{i+1,j}$ are stored in adjacent positions). Hence, it is convenient to scan the cost matrix by rows, using Pascal, and by columns, using FORTRAN.

Codes NAUC, AFLP and AFR have been originally implemented in FORTRAN to work with sparse cost matrices. Since our tests consider only dense instances, one has to consider the possibility of substituting the pointer-based data structure, used to store the sparse matrices, with a full matrix. We have modified all the codes accordingly and we have performed a set of preliminary tests. It resulted that the use of a full matrix is advantageous only when the auction method is paired with the shortest path technique. Therefore the code NAUC we tested utilizes a full matrix, instead of the original data structure, whereas algorithms AFLP and AFR have not been changed. Since the original implementation of NAUC stores the sparse cost matrix by rows, we have swapped the row and column indices. Lastly, the auction codes have been designed to solve maximization problems, so we run these algorithms with cost matrix $[-c_{ij}]$ instead of $[c_{ij}]$.

Algorithm CSA is distributed with a package containing a main procedure which reads the input data, prepares the internal data structure and then runs the optimization procedure which solves the problem. In order to use the same main FORTRAN program to run all codes, we have implemented two interfaces for running CSA. The first interface is a FORTRAN subroutine which receives the cost matrix, stores the costs in a single vector and calls the second interface, written in C language, which prepares the data structure and runs the optimization procedure. The CPU time elapsed is calculated only for the optimization phase. Since CSA requires as input a maximization problem and the C language stores the matrices by rows, the interface subroutines give the optimization procedure the opposite of the transposed FORTRAN cost matrix (i.e. we give CSA the costs $d_{ij} = -c_{ji}$, for $i, j = 1, \ldots, n$).

During a first set of preliminary experiments, we encountered some difficulties with the definition of the large positive value we give to an entry that does not exist in an original instance. In particular, we gave these entries the value 2×10^9. The same value is given to the algorithms for internal use. With this choice, algorithm LAPm often entered an infinite loop, especially when we tried to solve geometric instances (see below). We skipped the problem by giving value 10^8 to the largest cost of an entry and giving the value 2×10^9 for internal use.

4.2. The classes of instances

To test the performance of the eight competitors, we have used six classes of randomly generated problems and 141 benchmark instances. The size of the cost matrix varies up to one thousand rows and columns. (It is not easy to determine how much core memory is used by each of the algorithms tested, especially for the C codes which dynamically allocate the memory during the run, but instances with up to 10^6 entries are easily solved with any "of-the-shelf" computer.)

The random problems have been generated by means of the DIMACS completely portable uniform random number generator (see [35]).

4.2.1. Random instances

The six random classes we have considered are as follows.

Uniform random class: The entries of the cost matrix are integers uniformly randomly generated in $[0, K]$, with $K \in \{10, 10^2, 10^3, 10^6\}$. This is the most common class of instances used in the literature to test AP algorithms (see e.g. [29,37,20,31]).

Geometric class: We first generate two sets of points, X and Y, each containing n points with integer coordinates in the square $[1 \times K] \times [1 \times K]$ with $K \in \{10, 10^2, 10^3, 10^6\}$.

Then for each pair (i,j), for $i,j=1,\dots,n$, we assign to c_{ij} the truncated euclidean distance between the ith point of X and the jth point of Y. This class of instances was introduced in [31].

No-wait flow-shop class: It is well known that an instance of the scheduling problem known as *no-wait flow-shop* can be transformed into an instance of the *Asymmetric Travelling Salesman Problem* (ATSP). We have generated ATSP instances as those proposed in [19] (i.e. derived from no-wait flow-shop scheduling problems with ten and twenty machines, and up to one thousand jobs) and we have solved AP with the corresponding cost matrix. This is a new test class for AP.

Two cost class: Each entry of the cost matrix is given cost 1 with probability p and cost 10^6 otherwise, where $p \in \{0.25, 0.5, 0.75\}$. This class is the dense version of the analogous class introduced in [31].

Randomized Machol Wien class: This class, obtained by randomization from the benchmark instances of Machol and Wien [42,43], was first introduced in [22]. In particular, c_{ij} is assigned an integer value uniformly randomly generated in $[0,(i-1)(j-1)]$.

SDVSP class: In the *Single Depot Vehicle Scheduling Problem* (SDVSP) we want to find the minimum cost assignment of buses, located in the same depot, to a set of time-tabled trips. It is known that the problem can be modeled as an AP defined by a particular cost matrix. We have generated SDVSP instances as in [27] and transformed each one into an AP instance. Since the number of rows (and columns) in the assignment must be almost double the number of trips in the SDVSP instance, we limited the number of trips to 600. This is a new test class for AP.

4.2.2. Benchmark instances

The first set of five benchmark instances we used were proposed by Machol and Wien in [42,43].

Machol Wien class: This is a famous class of difficult instances defined by $c_{ij} = (i-1)(j-1)$, for $i,j=1,\dots,n$.

The other benchmarks we used are instances of the *Travelling Salesman Problem* (TSP) and of the *Capacitated Vehicle Routing Problem* (CVRP). It is well known that the assignment problem defines a lower bound for TSP, which has been extensively used in the literature. Hence, it is important to test the performance of the AP algorithms on these instances. We used benchmarks with at most 1000 nodes, taken from the TSPLIB (see [50]). The first 75 instances correspond to *symmetric* TSP (i.e. $c_{ij}=c_{ji}$, for $i,j=1,\dots,n, i \neq j$), whereas the following 19 instances define *asymmetric* TSPs.

The first seven Vehicle Routing Problem instances (available in the TSPLIB [50]) are from Christofides and Eilon [24], another three instances are from Fisher [30] and another four instances are from Christofides et al. [25]. Additional 20 instances are unpublished problems randomly generated by Augerat et al. [6], with a clustering technique. More precisely, a random number of clusters of size 10×10 is generated on a 100×100 square, then n points are randomly generated within the clusters. Each cost c_{ij} is given the truncated euclidean distance between i and j.

Lastly, we tested the algorithms on the eight dense instances proposed by Beasley [12] and available in the OR-Library (see [13]).

4.3. Running the codes

The codes in Section 3 have been tested on two very common systems. The first one is a Sun Sparc Ultra 2 workstation running under a Unix operating system (SunOS version 5.5.1). The second one is a personal computer with a CPU Pentium with clock at 100 MHz, running under Windows '95.

On the workstation we used the SUN C and FORTRAN 77 compilers, version 4.0 with the compiler option `-O0` which disables the optimization. Therefore the final command lines used to compile are: `cc -O0 -c` ⟨*filename*⟩ and `f77 -O0 -c` ⟨*filename*⟩ (the option `-c` disables the linking phase). The object codes were linked with the `f77` utility (command line: `f77` ⟨*object files*⟩. The Unix function `times()` was used to determine the CPU time used by each code.

On the personal computer we used the Watcom FORTRAN 77/32 compiler version 10.6 and the Watcom C32 compiler version 10.6. We used no specific compiler instruction, but only the option `-5`, which tells the compilers that the microprocessor is a Pentium and the option `-od` which disables the compiler optimization. The final command lines used to compile are: `wfc386 -5 -od` ⟨*filename*⟩ for the FORTRAN language and `wcc386 -5 -od` ⟨*filename*⟩ for the C language. To link the codes we used the Watcom linker utility and the PharLap TNT DOS extender. With this system the resulting executable code can be run both under Windows and under MS-DOS. The command lines used are `wlink FILE` ⟨*object files*⟩ `NAME main.exp`, and `rebind main.exp` (the rebind command is used to obtain the final executable code `main.exe` from the intermediate code `main.exp`). We run the code in an MS-DOS window while no other program was running. The Watcom routine `gettim()` was used to calculate the CPU time used by each code.

5. Computational experiments

In this section we discuss the behavior of the selected algorithms, when solving the test instances described in Section 4.2.

For each class, for each value of the possible parameter defining the class, and for each value of n (with $n \in \{200, 400, 600, 800, 1000\}$), we have generated and solved ten instances. A time limit of 500 s was given to each algorithm for solving a single instance (but the limit was extended to 1500 s for each Machol Wien instance). In the tables, for each code and for each value of n, we report the average CPU time with respect to the number of instances solved within the time limit. The symbol 'tl' is used to indicate that the time limit was reached for all the ten instances. The symbol 'c' is used when code AFR cannot solve the instances, due to the restrictions

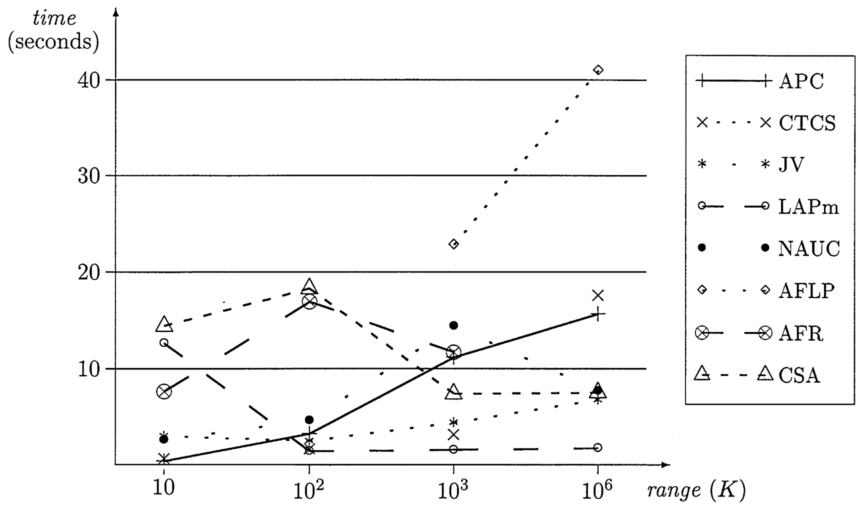

Fig. 1. Uniform random class, $n = 1000$, Sun Sparc Ultra 2.

on the magnitude of the costs. The single digit after the CPU time gives the number of successful runs, when the number of solved instances is between one and nine.

We report in detail the computational experiments performed with the Sun Sparc Ultra 2 workstation, running under Unix, whereas for the results obtained with the Personal Computer, running under Windows 95, we give only some qualitative description and a figure. Indeed, we have observed that the speeds of the two computers are comparable (see Figs. 1 and 2) and the performance on the PC is close to that on the workstation (with a single exception that we will point out in the following). However, the running times on the PC are sensitive to the environment (total quantity of memory allocated, number of algorithms linked in the same executable file, etc.), so we prefer to present numerical results only with respect to the workstation which does not present such anomalies.

Fig. 1 depicts the average running times for the uniform random class, and for instances with $n = 1000$ (the numerical values are given in Table 2). The maximum running time for each of the algorithms APC, CTCS, JV, LAPm, NAUC and CSA, is at most 20% larger than the average time, thus showing a good robustness of these algorithms when solving uniform random instances. Instead, the maximum running time of the two pure auction methods, AFLP and AFR, is up to three times the average, thus showing a strong dependence on the instance. Table 3 gives the results obtained with the uniform random class, for all values of n and for all values of the range parameter K (with $K = 10, 10^2, 10^3, 10^6$).

In Fig. 2 we report the average running times on the personal computer, for the same instances of Fig. 1 (note that the CPU time of AFLP for $K = 10^6$ is outside the figure since it is about 60 s). Comparing Figs. 1 and 2 one can see that the performances of the algorithms on the workstation and on the PC are quite similar, the only exceptions

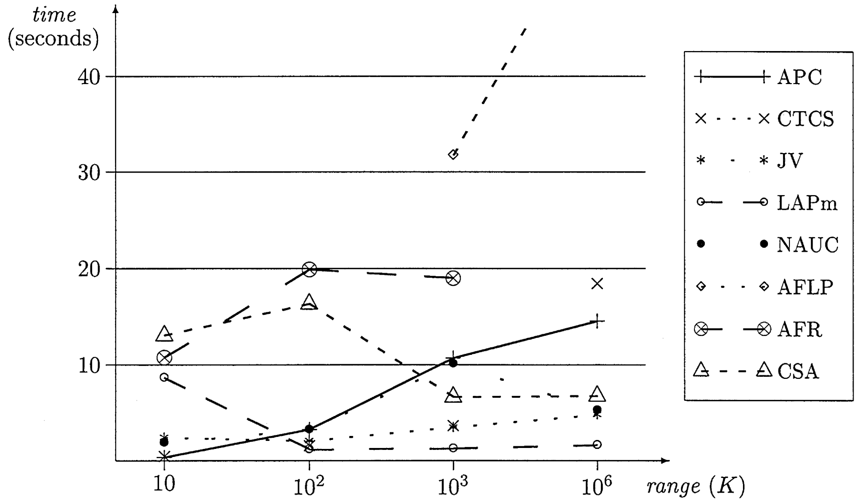

Fig. 2. Uniform random class, $n = 1000$, PC Pentium 100.

Table 2
Uniform random class, $n = 1000$, Sun Sparc Ultra 2 (time in seconds)

Range	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
10	0.42	0.65	3.00	12.15	2.61	tl	7.65	14.45
10^2	3.24	1.66	2.50	1.43	4.60	tl	16.92	18.29
10^3	11.18	3.13	4.39	1.55	14.41	22.82	11.69	7.40
10^6	15.67	17.58	6.76	1.72	7.68	40.99	c	7.52

being algorithms AFLP and AFR that we have already observed have a behavior less stable than that of the other methods.

For small costs instances ($K = 10$) the fastest algorithms are APC and CTCS, but their running time increase with the range. Algorithm LAPm is slower than APC and CTCS when the costs are small, but it is very fast for the other three ranges. The average performances of algorithm AFR are not too bad, but the running time for a single instance may be very high (about 40 s for an instance with $K = 10^3$) and the algorithm cannot be used for $K = 10^6$. Finally, the running time of AFLP is always very large, especially for small costs. Indeed, no instance with $K \leqslant 100$ was solved within the time limit of 500 s.

The results with the geometric class, for instances with $n = 1000$, are summarized in Fig. 3 (see also Tables 4 and 5). For all algorithms, except for LAPm and AFLP, the maximum running time never exceeds the average time by more than 30%. LAPm and AFLP, instead have maximum times up to five times larger than the average times. More specifically, LAPm presents a large variance of the running times for $n \leqslant 800$, whereas it is substantially stable for $n = 1000$. Nevertheless, for $K > 10$, LAPm is the fastest algorithm, followed by CSA. For $K = 10$ the fastest code is JV, which is also

Table 3
Uniform random class, Sun Sparc Ultra 2 (time in seconds)

n	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
$c_{ij} \in [1, 10]$								
200	0.02	0.05	0.09	0.34	0.09	8.10	0.22	0.28
400	0.07	0.12	0.43	1.90	0.39	73.38	0.85	1.69
600	0.16	0.23	1.03	4.59	0.92	257.17	2.23	4.27
800	0.28	0.42	1.88	7.98	1.65	tl	3.27	8.23
1000	0.42	0.65	3.00	12.15	2.61	tl	7.65	14.45
$c_{ij} \in [1, 10^2]$								
200	0.22	0.13	0.11	0.09	0.20	0.47	0.17	0.17
400	1.25	0.41	0.52	0.29	1.28	23.39	0.92	1.38
600	2.41	0.64	0.94	0.59	2.65	134.54	2.45	5.20
800	2.80	1.09	1.60	0.99	3.55	422.72 7	7.91	10.93
1000	3.24	1.66	2.50	1.43	4.60	tl	16.92	18.29
$c_{ij} \in [1, 10^3]$								
200	0.24	0.14	0.12	0.09	0.16	0.25	0.15	0.17
400	1.08	0.51	0.56	0.30	0.99	1.14	0.64	0.89
600	3.06	1.13	1.44	0.60	3.08	3.03	1.60	2.21
800	6.10	2.02	2.77	1.02	7.46	11.26	4.81	4.44
1000	11.18	3.13	4.39	1.55	14.41	22.82	11.69	7.40
$c_{ij} \in [1, 10^6]$								
200	0.28	0.15	0.32	0.12	0.32	0.53	c	0.19
400	1.35	0.58	1.06	0.34	0.97	2.67	c	0.97
600	4.04	4.58	2.54	0.70	2.74	8.18	c	2.52
800	8.55	9.70	4.47	1.17	4.97	23.92	c	4.41
1000	15.67	17.58	6.76	1.72	7.68	40.99	c	7.52

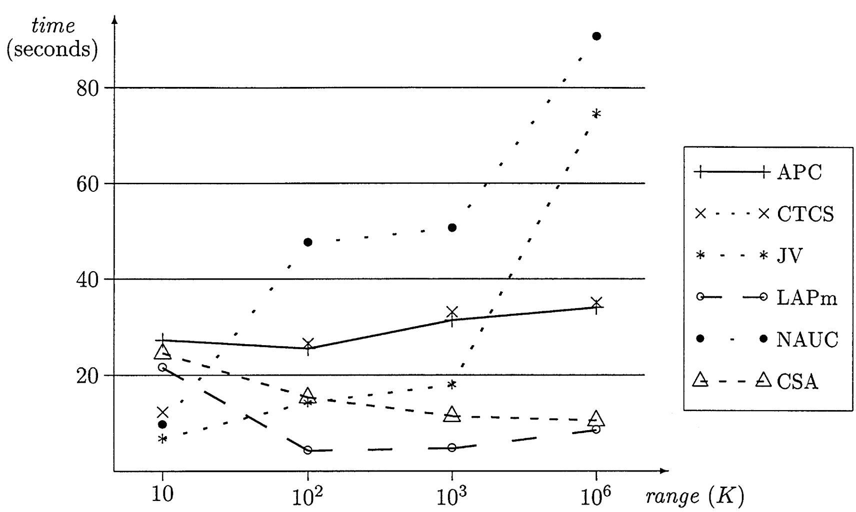

Fig. 3. Geometric class, $n = 1000$, Sun Sparc Ultra 2.

Table 4
Geometric class, $n = 1000$, Sun Sparc Ultra 2 (time in seconds)

Range	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
10	27.26	12.35	6.80	21.57	9.67	tl	tl	24.56
10^2	25.50	26.54	14.27	4.26	47.52	tl	tl	15.32
10^3	31.42	33.10	17.96	4.74	50.58	tl	tl	11.42
10^6	34.08	35.07	74.57	8.52	90.55	tl	c	10.54

Table 5
Geometric class, Sun Sparc Ultra 2 (time in seconds)

n	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
$c_{ij} \in [1, 10^2]$								
200	0.33	0.31	0.18	0.40	0.31	7.53	4.85	0.45
400	2.25	1.50	0.86	2.03	1.55	84.83	96.97	2.74
600	6.74	4.06	2.09	5.42	3.32	319.93	161.63 7	7.45
800	15.28	7.81	4.26	11.39	5.94	tl	262.68 1	15.25
1000	27.26	12.35	6.80	21.57	9.67	tl	tl	24.56
$c_{ij} \in [1, 10^2]$								
200	0.43	0.49	0.30	0.32	0.58	17.39	16.67	0.33
400	2.36	2.62	1.61	0.88	3.78	219.25	169.73 4	1.54
600	6.81	7.55	4.10	2.16	11.29	324.27 4	tl	4.35
800	15.27	15.92	8.41	4.44	25.21	tl	tl	10.83
1000	25.50	26.54	14.27	4.26	47.52	tl	tl	15.32
$c_{ij} \in [1, 10^3]$								
200	0.53	0.56	0.38	0.35	0.64	43.66	22.94	0.27
400	2.97	3.21	1.96	1.03	3.94	267.32 5	199.88 3	1.26
600	8.07	8.68	4.97	2.76	11.95	494.25 1	tl	3.38
800	18.60	19.68	10.40	5.47	27.21	tl	tl	6.59
1000	31.42	33.10	17.96	4.74	50.58	tl	tl	11.42
$c_{ij} \in [1, 10^6]$								
200	0.56	0.58	4.08	1.13	4.19	32.81	c	0.31
400	3.17	3.31	12.86	2.23	13.55	192.36 9	c	1.41
600	8.74	9.06	29.24	5.12	31.44	334.22 6	c	3.38
800	20.17	20.51	48.83	8.73	57.70	tl	c	6.45
1000	34.08	35.07	74.57	8.52	90.55	tl	c	10.54

competitive for $K = 10^2$ and 10^3, but it is dramatically slow for the largest cost range. By solving geometric instances with $K = 10^6$ we have observed that algorithm LAPm has a different behavior when running on the workstation or on the personal computer. Indeed, the CPU times on the PC are about one order of magnitude larger than those on the workstation. However, this happens only with this algorithm (LAPm) and only with this class and range. All other algorithms and instances have similar running times on the two systems. We have not been able to find any convincing justification for this behavior.

The barchart in Fig. 4 reports the CPU times used by each algorithm to solve the no-wait flow-shop instances with 1000 jobs, and 10 or 20 machines, respectively. Algorithm CSA outperforms all other methods. Algorithms JV, LAPm and AFR are the second best methods, but their running times are one order of magnitude larger than

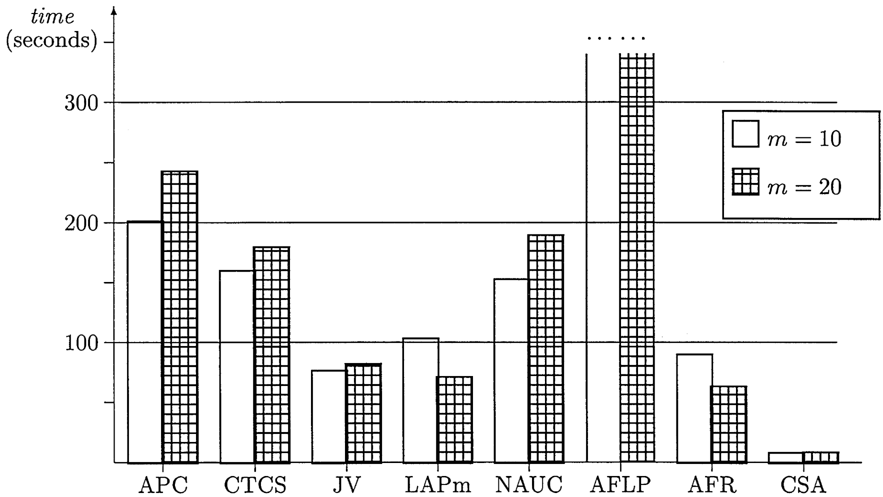

Fig. 4. No-wait flow-shop class, $n = 1000$, Sun Sparc Ultra 2.

Table 6
No-wait flow-shop, Sun Sparc Ultra 2 (time in seconds)

m	jobs	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
	200	2.07	1.64	0.88	1.56	1.56	5.44	0.86	0.24
	400	15.01	11.45	6.03	9.63	11.17	70.82	8.47	1.10
10	600	47.89	36.99	18.56	27.57	35.65	184.80	17.96	2.37
	800	108.90	83.94	40.91	55.98	80.27	338.29 6	47.63	4.21
	1000	205.86	159.30	75.89	102.69	152.34	459.27 1	89.66	7.18
	200	2.36	1.63	0.88	0.80	1.63	4.36	0.51	0.22
	400	17.22	12.74	6.08	5.50	12.70	30.88	4.07	1.00
20	600	54.35	42.91	19.16	17.30	41.30	120.39	19.29	2.32
	800	125.98	91.78	43.09	36.70	98.11	302.76	26.75	4.29
	1000	242.04	178.82	81.25	70.19	188.82	388.29 5	62.40	7.69

that of CSA. AFLP is about twice as slow as the worst among the other algorithms and is not able to solve all the instances within the time limit (see Table 6). It is worth noting that when we increase the number of machines of an instance, while keeping the same number of jobs, some algorithms (e.g. APC and NAUC) require longer running times, whereas some other algorithms (e.g. LAPm and AFR) require shorter computing times. Finally we note that the difference between the maximum and the average computing time never exceeds 10% of the average time, for all algorithms.

Concerning the two cost instances, we observe that the maximum running time of each algorithm is almost identical to its average running time. Moreover there is no significant difference, when the percentage of high cost entries increases from 25% to 75%. Therefore we decided to give the results only for $p=0.50$, see Fig. 5 and Table 7. Algorithms APC and CTCS are very fast, and beat the other methods by at least one order of magnitude. Algorithm AFLP is not competitive at all, whereas AFR cannot be

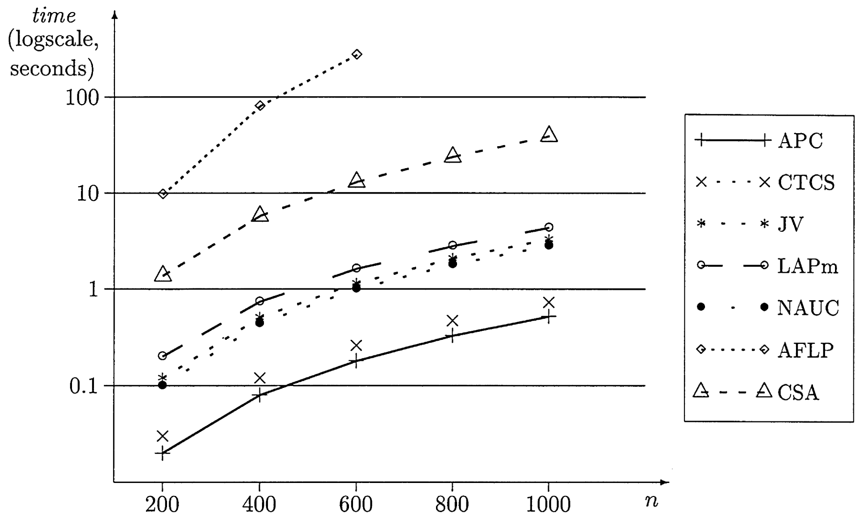

Fig. 5. Two cost class, $p = 0.50$, Sun Sparc Ultra 2.

Table 7
Two cost class, $p = 0.50$, Sun Sparc Ultra 2 (time in seconds)

n	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
200	0.02	0.03	0.12	0.20	0.10	9.67	c	1.38
400	0.08	0.12	0.51	0.74	0.44	80.20	c	5.77
600	0.18	0.26	1.15	1.62	1.00	272.40	c	13.09
800	0.33	0.47	2.08	2.80	1.80	tl	c	23.91
1000	0.52	0.73	3.29	4.36	2.83	tl	c	39.31

used, due to the large cost values. Algorithm CSA is two orders of magnitude slower than APC.

The randomized Machol Wien instances (see Fig. 6 and Table 8) are solved with no great effort by all the algorithms, with the exception of AFR that cannot solve instances with $n \geqslant 600$, since the costs are too large. The maximum computing times are at most 1.2 larger than the average times, thus confirming that these instances are substantially not too difficult. The fastest algorithm is CSA followed by four almost equivalent codes, namely JV, LAPm, AFLP and NAUC. Algorithm AFR is fast, but can solve only small instances. It is worth noting that the running time of LAPm increases linearly with n.

The instances from the single depot vehicle scheduling class (see Table 9) are very difficult for the pure auction methods, which are outperformed by all other methods. Code JV is able to solve these instances in few seconds, and CSA is only slightly slower. APC, CTCS and NAUC are also not too bad, whereas LAPm is five to ten times slower than JV.

In Tables 10–14 we report the results obtained with the benchmark instances.

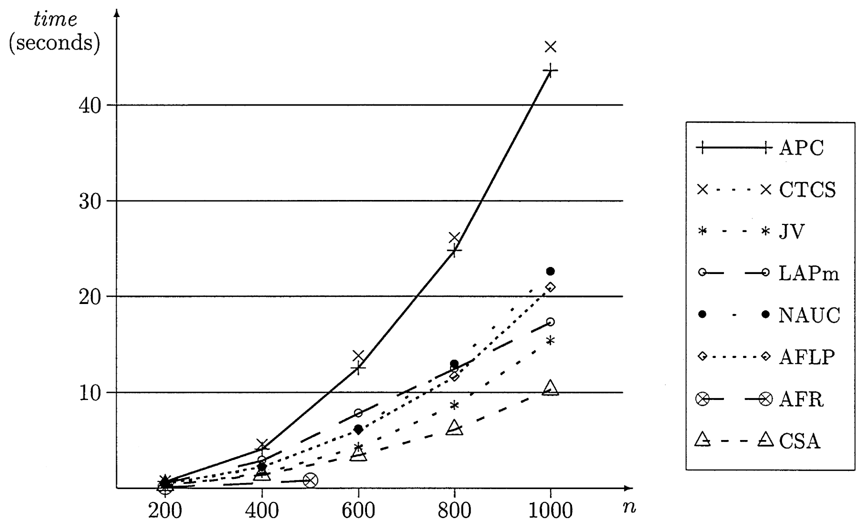

Fig. 6. Randomized Machol Wien class, Sun Sparc Ultra 2.

Table 8
Randomized Machol Wien, Sun Sparc Ultra 2 (time in seconds)

n	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
200	0.73	0.75	0.30	0.56	0.39	0.47	0.19	0.30
400	4.12	4.63	1.57	2.92	2.17	2.28	0.81	1.40
600	12.54	13.88	4.33	7.81	6.16	6.06	c	3.45
800	24.80	26.15	8.67	12.48	12.89	11.66	c	6.18
1000	43.63	46.09	15.41	17.34	22.56	20.90	c	10.30

Table 9
Single depot vehicle scheduling class, Sun Sparc Ultra 2 (time in seconds)

Trips	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
200	1.02	1.29	0.72	6.34	1.52	421.70 1	64.02 2	1.67
400	7.71	10.16	4.92	45.80	11.35	tl	tl	7.37
600	21.87	27.17	14.71	143.06	34.92	tl	c	16.68

Table 10
Machol Wien class, Sun Sparc Ultra 2 (time in seconds)

n	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
200	6.52	4.62	3.73	9.00	10.75	45.35	114.47	1.70
400	52.95	39.07	28.83	76.15	87.10	748.93	tl	7.28
600	180.10	135.03	97.02	263.32	295.37	tl	c	19.90
800	430.93	314.03	228.82	664.317	705.43	tl	c	55.82
1000	846.80	613.87	446.47	1323.433	1381.95	tl	c	143.97

Table 11
TSPLIB: Symmetric instances, Sun Sparc Ultra 2 (time in seconds)

Name	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
A280.TSP	3.82	4.32	2.22	7.82	8.15	478.85	105.25	5.62
ALI535.TSP	0.52	0.78	0.43	0.77	0.55	tl	47.45	2.02
ATT48.TSP	0.02	0.03	0.02	0.05	0.05	0.05	0.03	0.02
ATT532.TSP	18.63	18.23	12.78	70.88	78.35	tl	tl	6.42
BAYG29.TSP	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.02
BAYS29.TSP	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01
BERLIN52.TSP	0.03	0.03	0.03	0.08	0.03	3.52	0.22	0.03
BIER127.TSP	0.65	0.67	0.68	0.92	1.45	tl	104.22	0.37
BRAZIL58.TSP	0.02	0.02	0.01	0.01	0.01	0.18	0.02	0.02
BRG180.TSP	0.01	0.02	0.05	0.03	0.05	0.07	0.05	0.23
BURMA14.TSP	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02
CH130.TSP	0.37	0.40	0.23	0.92	0.48	14.08	7.58	0.15
CH150.TSP	0.58	0.63	0.37	1.38	0.95	11.13	7.72	0.22
D198.TSP	2.13	2.43	1.60	3.10	5.08	80.15	66.40	1.13
D493.TSP	38.93	43.37	26.20	54.35	84.88	tl	tl	8.58
D657.TSP	64.05	72.53	42.07	124.75	166.77	tl	tl	14.68
DANTZIG.TSP	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.02
DSJ1000.TSP	312.32	324.83	263.87	499.33	tl	tl	0.01	18.43
FL417.TSP	9.62	7.48	4.07	21.38	11.68	tl	tl	10.48
FRI26.TSP	0.01	0.02	0.01	0.01	0.01	0.02	0.01	0.02
GIL262.TSP	2.37	2.32	1.25	6.05	4.15	59.17	7.10	0.72
GR17.TSP	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.01
GR21.TSP	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.01
GR24.TSP	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.01
GR48.TSP	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01
GR96.TSP	0.02	0.03	0.02	0.03	0.02	0.18	0.03	0.03
GR120.TSP	0.03	0.03	0.03	0.03	0.03	3.73	0.23	0.12
GR137.TSP	0.03	0.05	0.03	0.05	0.03	16.07	0.03	0.10
GR202.TSP	0.08	0.12	0.07	0.08	0.08	365.83	22.97	0.38
GR229.TSP	0.12	0.17	0.10	0.13	0.10	47.23	0.18	0.30
GR431.TSP	0.43	0.63	0.33	0.33	0.50	tl	13.73	1.47
GR666.TSP	1.18	1.68	0.78	1.12	1.35	tl	235.77	3.22
HK48.TSP	0.01	0.02	0.01	0.01	0.02	0.02	0.02	0.02
KROA100.TSP	0.20	0.18	0.15	0.47	0.28	16.15	3.63	0.10
KROA150.TSP	0.72	0.77	0.55	1.50	1.15	38.33	9.55	0.27
KROA200.TSP	1.68	1.85	1.07	3.72	2.65	80.60	23.97	0.52
KROB100.TSP	0.20	0.22	0.20	0.50	0.37	10.01	4.72	0.20
KROB150.TSP	0.73	0.80	0.53	1.72	1.12	49.75	21.72	0.33
KROB200.TSP	1.88	2.03	1.23	3.73	2.62	115.00	36.88	0.43
KROC100.TSP	0.20	0.22	0.15	0.47	0.33	5.77	2.33	0.12
KROD100.TSP	0.23	0.23	0.17	0.50	0.33	9.78	2.98	0.12
KROE100.TSP	0.22	0.23	0.22	0.50	0.30	12.42	1.10	0.12
LIN105.TSP	0.28	0.35	0.27	0.57	0.68	36.98	55.77	0.27
LIN318.TSP	7.62	8.05	5.17	13.90	18.03	tl	tl	2.45
LINHP318.TSP	7.58	8.05	5.17	13.90	18.02	tl	tl	2.47
P654.TSP	53.12	50.57	23.38	100.12	96.98	tl	tl	20.60
PA561.TSP	3.78	4.00	1.90	1.22	3.72	4.40	1.35	3.63
PCB442.TSP	19.55	20.90	11.90	35.70	34.68	tl	tl	9.07
PR76.TSP	0.10	0.10	0.10	0.22	0.27	96.10	58.13	0.18

Table 11 (Contd.)

Name	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
PR107.TSP	0.15	0.20	0.13	0.53	0.45	151.88	94.43	0.35
PR124.TSP	0.48	0.48	0.37	1.03	1.08	48.87	156.28	0.28
PR136.TSP	0.45	0.55	0.35	1.17	1.15	295.43	492.90	0.20
PR144.TSP	0.72	0.77	0.55	1.45	1.67	tl	304.85	0.50
PR152.TSP	0.80	0.87	0.60	1.77	1.80	tl	tl	0.72
PR226.TSP	2.60	2.82	1.83	4.70	5.12	tl	tl	1.48
PR264.TSP	3.67	4.00	2.38	7.23	9.33	tl	tl	1.57
PR299.TSP	6.92	7.77	5.57	13.62	17.67	tl	tl	3.78
PR439.TSP	14.23	15.15	9.98	36.83	47.53	tl	tl	4.82
RAT99.TSP	0.12	0.13	0.08	0.42	0.17	0.48	2.70	0.10
RAT195.TSP	1.07	1.28	0.62	2.73	1.60	5.92	12.65	0.52
RAT575.TSP	26.72	35.12	15.47	57.92	40.90	452.62	tl	6.65
RAT783.TSP	61.32	98.07	39.00	194.80	105.35	tl	tl	10.82
RD100.TSP	0.20	0.20	0.15	0.52	0.33	6.78	0.62	0.08
RD400.TSP	10.57	11.40	6.37	28.13	21.60	tl	254.28	2.25
SI175.TSP	0.08	0.12	0.08	0.08	0.13	2.95	0.13	0.17
SI535.TSP	0.52	0.80	0.45	0.60	1.15	71.43	16.78	1.00
ST70.TSP	0.05	0.05	0.05	0.12	0.08	0.07	0.12	0.03
SWISS42.TSP	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01
TS225.TSP	1.27	1.15	0.78	4.35	3.03	tl	tl	1.52
TSP225.TSP	2.53	2.96	1.73	4.13	4.93	103.35	96.37	1.97
U159.TSP	0.88	0.92	0.62	1.80	2.13	tl	339.15	1.27
U574.TSP	48.00	53.42	32.25	87.25	113.15	tl	tl	11.07
U724.TSP	93.50	105.37	60.32	150.15	213.55	tl	tl	21.00
ULYSSE1.TSP	0.01	0.01	0.01	0.01	0.02	0.01	0.03	0.01
ULYSSE2.TSP	0.01	0.01	0.01	0.02	0.01	0.02	0.01	0.01
Average on the solved instances	11.08	12.31	7.86	20.98	21.87	202.61	154.83	2.51

The Machol Wien instances (see Table 10) are very difficult for all methods. The two pure auction methods are not able to solve the instances with $n > 400$ within the 1500 s of the time limit. The fastest of the other methods is CSA which has running times 3–4 times shorter than JV, which is the second best algorithm.

For the other benchmark problems we have added a row, at the bottom of each table, with the average time over all test instances of the same class (for the unsolved instances, a computing time equal to the time limit has been considered).

The symmetric TSP instances (see Table 11) are all very different from each other, both in size (n ranges from 17 to 1000) and in the structure of the cost matrix. However, the relative behavior of the algorithms is quite insensitive to the instance, with only one exception that we will describe in the following. If we do not consider the pure auction methods, the remaining algorithms solve the instances with $n \leqslant 400$ in less than 20 s, and the largest ones in a few minutes. Algorithm CSA is very fast, indeed its average running time over all instances is less than one half that of the second best code, namely JV, and about four times shorter than those of APC and CTCS. The remaining algorithms LAPm and NAUC are slower. The auction codes AFLP and AFR were not able to solve within the time limit about one third of the

Table 12
TSPLIB: Asymmetric instances, Sun Sparc Ultra 2 (time in seconds)

Name	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
BR17.ATS	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
FT53.ATS	0.02	0.04	0.01	0.02	0.01	0.05	0.01	0.07
FT70.ATS	0.02	0.04	0.02	0.03	0.02	0.40	0.01	0.13
FTV33.ATS	0.01	0.01	0.01	0.02	0.01	0.10	0.01	0.01
FTV35.ATS	0.01	0.02	0.01	0.01	0.01	0.03	0.01	0.01
FTV38.ATS	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.01
FTV44.ATS	0.01	0.01	0.02	0.02	0.01	0.02	0.02	0.01
FTV47.ATS	0.02	0.02	0.01	0.01	0.01	0.13	0.09	0.02
FTV55.ATS	0.02	0.02	0.01	0.02	0.01	0.18	0.37	0.03
FTV64.ATS	0.01	0.02	0.01	0.02	0.02	0.20	0.02	0.03
FTV70.ATS	0.02	0.02	0.01	0.02	0.02	0.23	0.23	0.02
FTV170.ATS	0.10	0.11	0.07	0.07	0.09	1.62	2.65	0.15
KRO124P.ATS	0.02	0.03	0.02	0.04	0.02	0.04	2.65	0.22
P43.ATS	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.03
RY48P.ATS	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.07
RBG323.ATS	0.48	0.20	0.32	0.17	0.43	17.30	0.50	0.95
RBG358.ATS	0.95	0.28	0.33	0.20	0.62	20.93	0.77	1.97
RBG403.ATS	1.20	0.35	0.38	0.23	0.65	32.87	0.77	3.22
RBG443.ATS	1.45	0.43	0.52	0.29	0.80	41.50	1.00	2.77
Average	0.23	0.09	0.09	0.06	0.15	6.09	0.48	0.51

Table 13
Vehicle Routing Problems, Sun Sparc Ultra 2 (time in seconds)

Name	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
Eil22.VRP	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02
Eil23.VRP	0.01	0.01	0.01	0.02	0.01	0.02	0.01	0.01
Eil30.VRP	0.02	0.01	0.01	0.01	0.02	0.07	0.07	0.02
Eil33.VRP	0.02	0.01	0.01	0.01	0.02	0.05	0.07	0.02
Eil51.VRP	0.02	0.03	0.02	0.05	0.04	0.08	0.02	0.02
Eil76.VRP	0.05	0.07	0.05	0.17	0.08	0.10	0.10	0.02
Eil101.VRP	0.10	0.12	0.08	0.33	0.20	1.57	0.20	0.10
Fis45.VRP	0.01	0.02	0.01	0.04	0.03	1.34	0.11	0.05
Fis72.VRP	0.04	0.05	0.02	0.10	0.03	0.27	0.07	0.10
Fis135.VRP	0.32	0.35	0.17	0.78	0.46	2.53	1.77	0.35
M-N101.VRP	0.17	0.15	0.09	0.40	0.29	2.15	1.43	0.18
M-N121.VRP	0.32	0.35	0.20	0.60	0.55	1.47	3.05	0.23
M-N151.VRP	0.41	0.46	0.25	1.16	0.68	4.55	0.96	0.23
M-N200.VRP	0.92	1.04	0.55	2.34	1.62	4.32	2.85	0.53
Average	0.17	0.19	0.11	0.43	0.29	1.32	0.77	0.13

instances, and the running times for some of the solved instances are very long. A very exceptional case is DSJ1000, an instance with 1000 rows and columns, which is quite difficult for all methods, but for AFR which solves it with a running time that is surprisingly short.

The asymmetric TSP instances (see Table 12) have at most 443 rows and columns and are generally easy for all algorithms. Indeed, the maximum running time is about

Table 14
OR-Library, Sun Sparc Ultra 2 (time in seconds)

Name	APC	CTCS	JV	LAPm	NAUC	AFLP	AFR	CSA
A100.TXT	0.03	0.02	0.02	0.02	0.03	0.07	0.03	0.05
A200.TXT	0.15	0.12	0.11	0.09	0.18	0.15	0.20	0.18
A300.TXT	0.18	0.12	0.12	0.08	0.18	0.93	0.13	0.17
A400.TXT	1.60	0.45	0.60	0.28	1.70	20.96	0.60	1.40
A500.TXT	1.79	0.67	0.85	0.45	2.77	38.45	1.79	3.48
A600.TXT	2.68	0.98	1.20	0.55	2.90	114.04	2.12	7.63
A700.TXT	4.50	1.34	1.42	0.75	3.46	237.43	6.15	9.12
A800.TXT	6.60	1.77	1.90	0.95	4.35	499.25	5.15	12.20
Average	2.19	0.68	0.78	0.40	1.95	113.91	2.02	4.28

three seconds, if we exclude code AFLP which runs up to twenty times slower than the other codes. The fastest algorithms are LAPm, JV and CTCS.

The Vehicle Routing Problems (see Table 13) are also not very large (at most 200 rows and columns) and are easy. Indeed, algorithms APC, CTCS, JV and CSA solve each instance in less than one second, whereas the maximum running time, due to AFLP, is smaller than five seconds. The benchmark problems by Augerat et al. [6], not reported in the tables, were solved within at most 0.2 s by all the algorithms.

To solve the benchmark instances from the OR-Library (see Table 14) the best code is LAPm, which determines the optimal solution of each instance in less than one second. Algorithms CTCS and JV are also very fast, whereas AFLP is again very slow (up to 400 times slower than LAPm).

6. Conclusions

From the computational results it is not possible to obtain a precise ranking of the eight algorithms considered, but it is possible to evaluate their relative behavior.

We can first note that AFLP has almost always the longest computing times, and often exceeds the time limit. AFR has a similar behavior, although it is sometimes competitive with other algorithms. Moreover, several instances cannot be solved with this code, due to the restrictions on the values of the cost matrix. Hence, we can state that both AFLP and AFR are not competitive when solving dense instances, and we will not consider them any more in the following.

The other auction algorithm, NAUC, is beaten, on average, by three or four other codes on each class of instances, and in no entry is it the winner. Algorithm CTCS shows a better average performance, indeed it is generally the third or the fourth best code on each class, but it is never the winner in a class. APC is the fastest code for the two cost class, and has a behavior, on average, similar to that of CTCS for the other classes. Algorithm LAPm is the winner for the uniform random and the geometric classes, and for the instances from the OR-library. No dominance with respect to NAUC, CTCS and APC exists for the remaining classes. Code JV has a good and stable average performance for all the classes, and it is the best algorithm for the uniform random (together with LAPm) and for the single-depot class. Finally, the performances of CSA strongly depends on the class, indeed it is certainly the winner for classes no-wait flow-shop, randomized Machol Wien, Machol Wien and Symmetric TSP. On the other classes, it is either the second best code, either the worst one (classes uniform random, two cost and OR-library).

Acknowledgements

We thank an anonymous referee for his detailed comments which improved upon a first version of the paper. Partially supported by MURST (Italian Ministry of University and Scientific Research) and by CNR (National Council for the Research).

References

[1] R.K. Ahuja, J.B. Orlin, The scaling network simplex algorithm, Oper. Res. (Suppl. 1) (1992) S5–S13.
[2] M. Akgül, A sequential dual simplex algorithm for the linear assignment problem, Oper. Res. Lett. 7 (1988) 155–158.
[3] M. Akgül, Erratum. A sequential dual simplex algorithm for the linear assignment problem, Oper. Res. Lett. 8 (1989) 117.
[4] M. Akgül, The linear assignment problem, in: M. Akgül, H.W. Hamacher, S. Tüfekçi (Eds.), Combinatorial Optimization NATO ASI Series F, vol. 82, Springer, Berlin, 1992, pp. 85–122.
[5] M. Akgül, A genuinely polynomial primal simplex algorithm for the assignment problem, Discr. Appl. Math. 45 (1993) 93–115.
[6] P. Augerat, J.M. Belenguer, E. Benavent, A. Corberán, D. Naddef, G. Rinaldi, Computational results with a branch and cut code for the capacitated vehicle routing problem, Tech. Rep. RR949-M, IMAG, Grenoble, France, 1995.
[7] E. Balas, D. Miller, J. Pekny, P. Toth, A parallel augmenting shortest path algorithm for the assignment problem, J. ACM 38 (1991) 985–1004.
[8] M.L. Balinski, The Hirsch conjecture for dual transportation polyhedra, Math. Oper. Res. 9 (1984) 629–633.
[9] M.L. Balinski, Signature methods for the assignment problem, Oper. Res. 33 (1985) 527–536.
[10] M.L. Balinski, A competitive (dual) simplex method for the assignment problem, Math. Program. 34 (1986) 125–141.
[11] R.S. Barr, F. Glover, D. Klingman, The alternating basis algorithm for assignment problems, Math. Program 13 (1977) 1–13.
[12] J.E. Beasley, Linear programming on Cray supercomputers, J. Oper. Res. Soc. 41 (1990) 133–139.
[13] J.E. Beasley, Or-library: distributing test problems by electronic mail, J. Oper. Res. Soc. 41 (1990) 1069–1072.
[14] D.P. Bertsekas, A new algorithm for the assignment problem, Math. Program. 21 (1981) 152–171.
[15] D.P. Bertsekas, Linear Network Optimization: Algorithms and Codes, The MIT Press, Cambridge, MA, 1991.
[16] D.P. Bertsekas, J. Eckstein, Dual coordinate step methods for linear network flow problems, Math. Program. 42 (1988) 203–243.
[17] D.P. Bertsekas, D.A. Castañon, Parallel synchronous and asynchronous implementations of the auction algorithm, Parallel Comput. 17 (1991) 707–732.
[18] R.E. Burkard, U. Derigs, Assignment and Matching Problems: Solution Methods with FORTRAN Programs, Springer, Berlin, 1980. R.E. Burkard, E. Cela, Linear assignment and extensions, Tech. Rep. 127, Institut für Mathematik, Technische Universität Graz (1998).

[19] G. Carpaneto, M. Dell'Amico, P. Toth, Exact solution of large-scale, asymmetric traveling salesman problems, ACM Trans. Math. Software 21 (1995) 394–409.
[20] G. Carpaneto, S. Martello, P. Toth, Algorithms and codes for the assignment problem, in: B. Simeone, P. Toth, G. Gallo, F. Maffioli, S. Pallottino (Eds.), Fortran Codes for Network Optimization, Ann. Oper. Res., Vol. 13, Baltzer, Basel, 1988, pp. 193–223.
[21] G. Carpaneto, P. Toth, Solution of the assignment problem, ACM, Trans. Math. Software 6 (1980) 104–111.
[22] G. Carpaneto, P. Toth, Primal-dual algorithms for the assignment problem, Discrete Appl. Math. 18 (1987) 137–153.
[23] D.A. Castañon, Reverse auction algorithms for assignment problems, in: D.S. Johnson, C.C. McGeoch (Eds.), Network Flows and Matching: First DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1993, pp. 407–430.
[24] N. Christofides, S. Eilon, An Algorithm for the vehicle dispatching problem, Oper. Res. Quart. 20 (1969) 309–318.
[25] N. Christofides, A. Mingozzi, P. Toth, The vehicle routing problem, in: N. Christofides, A. Mingozzi, P. Toth, C. Sandi (Eds.), Combinatorial Optimization, Wiley, Chichester, 1979, pp. 318–338.
[26] W.H. Cunningham, A network simplex method, Math. Program. 11 (1976) 105–116.
[27] M. Dell'Amico, M. Fischetti, P. Toth, Heuristic algorithms for the multiple depot vehicle scheduling problem, Manage. Sci. 39 (1993) 115–125.
[28] M. Dell'Amico, S. Martello, Linear assignment, in: M. Dell'Amico, F. Maffioli, S. Martello (Eds.), Annotated Bibliographies in Combinatorial Optimization, Wiley, Chichester, 1997, pp. 355–371.
[29] U. Derigs, The shortest augmenting path method for solving assignment problems – motivation and computational experience, in: C.L. Monma, (Ed.), Algorithms and Software for Optimization – Part I, Ann. Oper. Res., Vol. 4, Baltzer, Basel, 1985, pp. 57–102.
[30] M. Fisher, Optimal solution of vehicle routing problem using minimum k-trees, Oper. Res. 42 (4) (1994) 626–642.
[31] A.V. Goldberg, R. Kennedy, An efficient cost scaling algorithm for the assignment problem, Math. Program. 71 (1995) 153–177.
[32] A.V. Goldberg, S.A. Plotkin, P. Vaidya, Sublinear-time parallel algorithms for matching and related problemst, J. Algorithms 14 (1993) 180–213.
[33] A.V. Goldberg, R.E. Tarjan, Finding minimum-cost circulation by successive approximation, Math. Oper. Res. 15 (1990) 430–466.
[34] M.S. Hung, A polynomial simplex method for the assignment problem, Oper. Res. 31 (1983) 595–600.
[35] D.S. Johnson, C.C. McGeoch (Eds.), Network Flows and Matching: First DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1993.
[36] R. Jonker, A. Volgenant, Improving the Hungarian assignment algorithm, Oper. Res. Lett. 5 (1986) 171–175.
[37] R. Jonker, A. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems, Computing 38 (1987) 325–340.
[38] J.L. Kennington, Z. Wang, An empirical analysis of the dense assignment problem: Sequential and parallel implementations, ORSA J. Comput. 3 (1991) 299–306.
[39] H.W. Kuhn, The Hungarian method for the assignment problem, Naval Res. Logistics Quart. 2 (1955) 83–97.
[40] H.W. Kuhn, Variants of The Hungarian method for the assignment problem, Naval Res. Logistics Quart. 3 (1956) 253–258.
[41] E.L. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, New York, 1976.
[42] R.E. Machol, M. Wien, A hard assignment problem, Oper. Res. 24 (1976) 190–192.
[43] R.E. Machol, M. Wien, Errata, Oper. Res. 24 (1977) 364.
[44] S. Martello, P. Toth, Linear assignment problems, in: S. Martello, G. Laporte, M. Minoux, C. Ribeiro, (Eds.), Surveys in Combinatorial Optimization, Annals of Discrete. Mathematics, Vol. 31, North-Holland, Amsterdam, 1987, pp. 259–282.
[45] L.F. McGinnis, Implementation and testing of a primal-dual algorithm for the assignment problem, Oper. Res. 31 (1983) 277–299.

[46] J.B. Orlin, On the simplex algorithm for networks and generalized networks, Math. Program. Stud. 24 (1985) 166–178.

[47] J.B. Orlin, R.K. Ahuja, New scaling algorithms for the assignment and minimum cycle mean problems, Math. Program. 54 (1992) 41–56.

[48] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[49] K.G. Ramakrishnan, N.K. Karmarkar, A.P. Kamath, An approximate dual projective algorithm for solving assignment problems, in: D.S. Johnson, C.C. McGeoch (Eds.), Network Flows and Matching: First DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1993, pp. 431–452.

[50] G. Reinelt, Tsplib – A travelling salesman problem library, ORSA J. Comput. 4 (1991) 376–384.

[51] B.L. Schwartz, A computational analysis of the auction algorithm, European. J. Oper. Res. 42 (1994) 161–169.

[52] B. Simeone, P. Toth, G. Gallo, F. Maffioli, S. Pallottino (Eds.), Fortran Codes for Network Optimization, Annals of Operation Research, Vol. 13, Baltzer, Basel, 1988.

[53] A. Volgenant, Linear and semi-assignment problems: a core oriented approach, Comput. Oper. Res. 23 (1996) 917–932.

[54] H. Zaki, A comparison of two algorithms for the assignment problem, Comput. Opt. Appl. 41 (1995) 23–45.

Update

Discrete Applied Mathematics

Volume 140, Issue 1–3, 15 May 2004, Page ix–x

: https://doi.org/10.1016/S0166-218X(04)00158-1

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Discrete Applied Mathematics 140 (2004) ix–x
DISCRETE APPLIED MATHEMATICS
www.elsevier.com/locate/dam

ERRATUM

Editorial Note: DAM Software Section☆

Dr. C.J. Leonard, Publishing Editor

Due to an oversight by the Publisher, the following articles have appeared in *Discrete pplied Mathematics* without any indication that these papers were supposed to be in e Mathematical Software Section of the journal. The Publisher wishes to apologise r this oversight to the authors of the articles and also to the Mathematical Software ection editors.

he construction of cubic and quartic planar maps with prescribed face degrees
iscrete Applied Mathematics, Volume 128, Issues 2–3, 1 June 2003, Pages 41–554
unnar Brinkmann, Thomas Harmuth and Oliver Heidemeier
-ref: 10.1016/S0166-218X(02)00549-8

ocal search algorithms for the *k*-cardinality tree problem
iscrete Applied Mathematics, Volume 128, Issues 2–3, 1 June 2003, Pages 11–540
hristian Blum and Matthias Ehrgott
-ref: 10.1016/S0166-218X(02)00548-6

euristics and meta-heuristics for 2-layer straight line crossing minimization
iscrete Applied Mathematics, Volume 127, Issue 3, 1 May 2003, Pages 665–678
afael Martí and Manuel Laguna
-ref: 10.1016/S0166-218X(02)00397-9

ortran subroutines for computing approximate solutions of weighted MAX-SAT roblems using GRASP
iscrete Applied Mathematics, Volume 100, Issues 1–2, 15 March 2000, Pages 5–113
auricio G.C. Resende , Leonidas S. Pitsoulis and Panos M. Pardalos
-ref: 10.1016/S0166-218X(99)00171-7

lgorithms and codes for dense assignment problems: the state of the art
iscrete Applied Mathematics, Volume 100, Issues 1–2, 15 March 2000, Pages 7–48

☆ PII of original article S0166-218X(02)00549-8 S0166218X02005486 S0166218X02003979 0166218X99001717 S0166218X99001729 S0166218X99000505 S0166218X99000487

Iauro Dell'Amico and Paolo Toth
-ref: 10.1016/S0166-218X(99)00172-9

eparating lifted odd-hole inequalities to solve the index selection problem
iscrete Applied Mathematics, Volume 92, Issues 2–3, June 1999, Pages 111–134
lberto Caprara and Juan José Salazar González
-ref: 10.1016/S0166-218X(99)00050-5

software package of algorithms and heuristics for disjoint paths in Planar Networks
iscrete Applied Mathematics, Volume 92, Issues 2–3, June 1999, Pages 91–110
lrik Brandes, Wolfram Schlickenrieder, Gabriele Neyer, Dorothea Wagner and Karsten
'eihe
-ref: 10.1016/S0166-218X(99)00048-7

Please note that submissions for the Mathematical Software Section are welcomed nd should be addressed to Professors S. Martello and P. Toth at smartello@deis.unibo.it nd ptoth@deis.unibo.it

Further information on submission for the Software Section and other sections can e found in the Guide for Authors at the back of this journal, or at the Elsevier Author ateway http://authors.elsevier.com/