ELSEVIER

Discrete Applied Mathematics 110 (2001) 25–40

DISCRETE APPLIED MATHEMATICS

# Efficient algorithms and codes for $k$-cardinality assignment problems

Mauro Dell'Amico[a], Andrea Lodi[b], Silvano Martello[b,*]

[a] *Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, Italy*
[b] *Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy*

**Abstract**

Given a cost matrix $W$ and a positive integer $k$, the $k$-cardinality assignment problem is to assign $k$ rows to $k$ columns so that the sum of the corresponding costs is a minimum. This generalization of the classical assignment problem is solvable in polynomial time, either by transformation to min-cost flow or through specific algorithms. We consider the algorithm recently proposed by Dell'Amico and Martello for the case where $W$ is dense, and derive, for the case of sparse matrices, an efficient algorithm which includes original heuristic preprocessing techniques. The resulting code is experimentally compared with min-cost flow codes from the literature. Extensive computational tests show that the code is considerably faster, and effectively solves very large sparse and dense instances. 

## 1. Introduction

A well-known problem in telecommunications is the *satellite-switched time-division multiple access* (SS/TDMA) time slot assignment problem (see, e.g., [14,9]). We are given $m$ transmitting stations and $n$ receiving stations, interconnected by a geostationary satellite through an on-board $k \times k$ switch ($k \leqslant \min(m,n)$), and an integer $m \times n$ *traffic matrix* $W = [w_{ij}]$, where $w_{ij}$ is the time needed for the transmission from station $i$ to station $j$. The problem is to determine a sequence of switching configurations which allow all transmissions to be performed in minimum total time, under the constraint that no preemption of a transmission occurs. A *switching configuration* is an $m \times n$ 0–1 matrix $X^\ell$ having exactly $k$ ones, with no two in the same row or column. Matrix $X^\ell$ is associated with a transmission time slot of duration $t^\ell = \max_{ij}\{X^\ell_{ij} w_{ij}\}$. The objective is thus to determine an integer $\tau$ and $X^\ell$ ($\ell = 1,\ldots,\tau$) so that $\sum_{\ell=1}^{\tau} t^\ell$ is a minimum.

* Corresponding author. Tel.: +39-051-644-3022; fax: +39-051-644-3073.
*E-mail addresses:* dellamico@unimo.it (M. Dell'Amico), alodi@deis.unibo.it (A. Lodi), smartello@deis.unibo.it (S. Martello).

0166-218X/01/$ - see front matter 
PII: S0166-218X(00)00301-2

The problem is known to be NP-hard. Several heuristic algorithms (see, e.g., [13,4]) determine, at each iteration, a switching configuration which minimizes the sum of the selected transmission times. In the following we consider efficient algorithms for exactly solving this local problem.

Given an integer $m \times n$ cost matrix $W$ and an integer $k$ $(k \leqslant \min(m,n))$, the *k-Cardinality Assignment Problem* ($k$-AP) is to select $k$ rows and $k$ columns of $W$ and to assign each selected row to a different selected column, so that the sum of the costs of the assignment is a minimum. Let $x_{ij}$ $(i=1,\ldots,m,\ j=1,\ldots,n)$ be a binary variable taking the value 1 if and only if row $i$ is assigned to column $j$. The problem can then be formulated as the following integer linear program:

$$z = \min \sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} x_{ij} \tag{1}$$

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

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

$$\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} = k, \tag{4}$$

$$x_{ij} \in \{0,1\} \quad (i = 1,\ldots,m,\ j = 1,\ldots,n). \tag{5}$$

In the special case where $m = n = k$, $k$-AP reduces to the well-known *Assignment Problem* (AP). (In this case the '$\leqslant$' sign in Eqs. (2) and (3) can be replaced by the '$=$' sign, and Eq. (4) can be dropped.)

It has been shown by Dell'Amico and Martello [5] that the constraint matrix of $k$-AP is totally unimodular, hence the polyhedron defined by the continuous relaxation of (1)–(5) has integral vertices. Thus, the optimal solution to the problem can be obtained in polynomial time through linear programming.

It is well known that AP is to find the minimum cost basis of the intersection of two partition matroids, hence $k$-AP is the optimum basis of the intersection of two matroids truncated to $k$. Since the two-matroid intersection algorithm runs in polynomial time, we know that $k$-AP is polynomial even for real-valued cost matrices. This also gives a possible dual algorithm for $k$-AP: start with an empty solution (no row assigned) and apply for $k$ times a *shortest augmenting path* technique (see, e.g., [11]) obtaining, at each iteration, a solution in which one more row is assigned. Another possible approach consists in solving a min-cost flow problem on an appropriate network (see Section 5).

In addition to the SS/TDMA context, $k$-AP has immediate applications in assigning workers to machines when there are multiple alternatives and only a subset of workers and machines has to be assigned. Although extensive literature exists on AP and related

problems (surveys can be found, e.g., in [12,1,6]), to our knowledge the only specific result on $k$-AP has been presented by Dell'Amico and Martello [5].

The algorithm in [5] was developed for solving instances in which matrix $W$ is complete. In this paper, we consider the case where $W$ is sparse, i.e., many entries are empty, or, equivalently, they have infinite cost. The sparse case is relevant for applications. Consider, e.g., the SS/TDMA time slot assignment previously discussed: it is clear that, in general, a transmitting station will not send messages to *all* receiving stations, so we can expect that real-world traffic matrices are sparse.

In the sparse case, the problem is more conveniently formulated through the following graph theory model. Let $G = (U \cup V, A)$ be a bipartite digraph where $U = \{1,\dots,m\}$ and $V = \{1,\dots,n\}$ are the two vertex sets and $A \subseteq U \times V$ is the arc set: $A = \{(i,j)\colon w_{ij} < +\infty\}$. Let $w_{ij}$ be the weight of an arc $(i,j) \in A$. We want to select $k$ arcs of $A$ such that no two arcs share a common vertex and the sum of the costs of the selected arcs is a minimum.

In the following we assume, without loss of generality, that $w_{ij} \geqslant 0\ \forall (i,j) \in A$. Since $k$-AP does not change if we swap sets $U$ and $V$, we also assume, without loss of generality, that $m \leqslant n$.

The problem in which the objective is to maximize the cost of the assignment, is solved by $k$-AP if each cost $w_{hl}$ $((h,l) \in A)$ is replaced by the nonnegative quantity $\tilde{w}_{hl} = \max_{(i,j)\in A}\{w_{ij}\} - w_{hl}$.

The aim of this article is to provide a specialized algorithm, and the corresponding computer code, for the exact solution of $k$-AP.

## 2. Complete matrices

The main phases of the approach in [5] are a heuristic preprocessing and an exact primal algorithm. Preprocessing finds a partial optimal solution, determines sets of rows and columns which must be assigned in an optimal solution and reduces the cost matrix. The partial solution is then heuristically completed, and the primal algorithm determines the optimum through a series of alternating paths.

The preprocessing phase can be summarized as follows.

*Step* 0: For each row $i$ determine the first, second and third smallest cost in the row $(\rho_i, \rho_i', \rho_i''$, respectively) and let $c(i), c'(i), c''(i)$ be the corresponding columns. For each column $j$ determine the first, second and third smallest cost in the column $(\gamma_j, \gamma_j', \gamma_j''$, respectively) and let $r(j), r'(j), r''(j)$ be the corresponding rows. Reorder the columns of $W$ by nondecreasing $\gamma_j$ values, and the rows by nondecreasing $\rho_i$ values.

*Step* 1: Determine a value $g_1 \leqslant k$, as high as possible, such that a partial optimal solution in which $g_1$ rows are assigned can be heuristically determined. This is obtained through a heuristic algorithm which, for increasing values of $j$, assigns row $r(j)$ to column $j$ if this row is currently unassigned; if instead row $r(j)$ is already assigned, under certain conditions the algorithm assigns column $j+1$ instead of $j$ (and then proceeds to column $j+2$) or assigns the current column $j$ to row $r'(j)$ instead of $r(j)$.

If $g_1 = k$ then terminate (the solution is optimal); otherwise, execute a greedy heuristic to determine an approximate solution of cardinality $k$ and let $UB_1$ be the corresponding value.

*Step* 2: Repeat Step 1 over the transposed cost matrix. Let $g_2$ and $UB_2$ be the values obtained.

*Step* 3: $UB := \min(UB_1, UB_2)$; $g := \max(g_1, g_2)$.

*Step* 4: Starting from the partial optimal solution of cardinality $g$, determine a set $R$ of rows and a set $C$ of columns that must be selected in an optimal solution to $k$-AP. Compute a lower bound $LB$ on the optimal solution value.

*Step* 5: If $UB > LB$ then reduce the instance by determining pairs $(i, j)$ for which $x_{ij}$ must assume the value zero.

The overall time complexity of the above preprocessing phase is $O(mn + kn \log n)$.

Once Step 5 has been executed, the primal phase starts with the $k \times k$ submatrix of $W$ induced by the sets of rows and columns selected in the heuristic solution of value $UB$, and exactly solves an AP over this matrix. Then $m+n-2k$ iterations are performed by adding to the current submatrix one row or column of $W$ at a time, and by re-optimizing the current solution through alternating paths on a particular digraph induced by the submatrix.

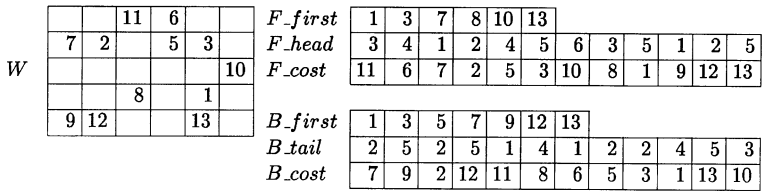
## 3. Sparse matrices

In this section we discuss the most relevant features of the algorithm for the sparse case.

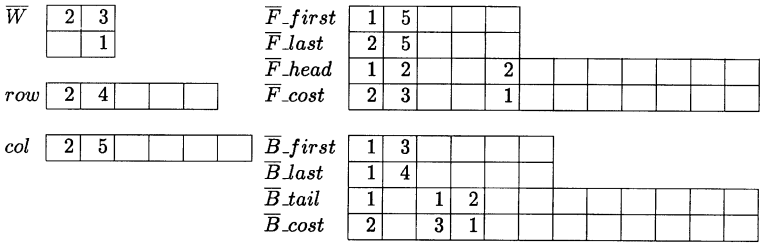
### 3.1. Data structures

Let $\mu$ be the number of entries with finite cost. We store matrix $W$ through a *forward star* (see Fig. 1(a)) consisting of an array, $F\_first$, of $m+1$ pointers, and two arrays, $F\_head$ and $F\_cost$, of $\mu$ elements each. The entries of row $i$, and the corresponding columns, are stored in $F\_cost$ and $F\_head$, in locations $F\_first(i), \ldots, F\_first(i+1)-1$ (with $F\_first(1) = 1$ and $F\_first(m+1) = \mu + 1$). Since the algorithm needs to scan $W$ both by row and by column, we duplicate the information into a *backward star*, similarly defined by arrays $B\_first$ (of $n+1$ pointers), $B\_tail$ and $B\_cost$, storing the entries of column $j$, and the corresponding rows, in locations $B\_first(j), \ldots, B\_first(j+1) - 1$.

In the primal phase a local matrix $\bar{W}$ is initialized to the $k \times k$ submatrix of $W$ obtained from preprocessing, and iteratively enlarged by addition of a row or a column. Let $row(i)$ (resp. $col(j)$) denote the row (resp. column) of $W$ corresponding to row $i$ (resp. column $j$) of $\bar{W}$. Matrix $\bar{W}$ is stored through a modified forward star. In this case we have two arrays, $\bar{F}\_head$ and $\bar{F}\_cost$, of length $\mu$, and two arrays, $\bar{F}\_first$ and $\bar{F}\_last$, of $m$ pointers each. For each row $i$ currently in $\bar{W}$: (a) the elements of the current columns, and the corresponding current column indices, are stored in




(a) Input matrix.




(b) Local matrix.

Fig. 1. Data structures.

$\bar{F}\_cost$ and $\bar{F}\_head$, in locations $\bar{F}\_first(i),\ldots,\bar{F}\_last(i)$; (b) locations $\bar{F}\_last(i)+1,\ldots,\bar{F}\_first(i)+L-1$ (where $L=F\_first(row(i)+1)-F\_first(row(i))$ denotes the number of elements in row $row(i)$ of $W$) are left empty for future insertions. The backward star for $\bar{W}$ is similarly defined by arrays $\bar{B}\_tail$, $\bar{B}\_cost$, $\bar{B}\_first$ and $\bar{B}\_last$. Fig. 1(b) shows the structures for a $2\times 2$ submatrix $\bar{W}$ of $W$.

### 3.2. Approximate solution

All computations needed by the preprocessing of Section 2 were efficiently implemented through the data structures described in the previous section, but the greedy solution of Step 1 (and 2) required a specific approach. Indeed, due to sparsity, in some cases a solution of cardinality $k$ does not exist at all. Even if $k$-cardinality assignments exist, in many cases the procedure adopted for the complete case proved to be highly inefficient in finding one. For the same reason, a second greedy approach was needed at Step 3. These approaches are discussed in Section 3.3.

In certain cases, the execution of the greedy algorithms can be replaced by the following transformation which provides a method to determine a $k$-cardinality assignment, or to prove that no such assignment exists. Let us consider the bipartite digraph $G=(U\cup V,A)$ introduced in Section 1, and define a digraph $G'=(\{s\}\cup\{\bar{s}\}\cup U\cup V\cup\{t\},A')$, where $A'=A\cup\{(s,\bar{s})\}\cup\{(\bar{s},i)\colon i\in U\}\cup\{(j,t)\colon j\in V\}$. Now assign unit capacity to all arcs of $A'$ except $(s,\bar{s})$, for which the capacity is set to $k$, and send the maximum flow $\vartheta$ from $s$ to $t$. It is clear that this will either provide a $k$-cardinality assignment or prove that no such assignment is possible. If $\vartheta<k$, we know that no

solution to $k$-AP exists. If $\vartheta = k$, the arcs of $A$ carrying flow provide a $k$-cardinality assignment, which is suboptimal since the arc costs have been disregarded.

Computational experiments proved that the above approach is only efficient when the instance is very sparse, i.e., $|A| \leqslant 0.01\ mn$, and $k > 0.9\ m$. For different cases, much better performance was obtained through the following greedy approaches.

### 3.3. Greedy approximate solutions

We recall that at steps 1 and 2 of the preprocessing phase described in Section 2 we have partial optimal solutions of cardinality $g_1$ and $g_2$, respectively. Starting from one of these solutions, the following algorithm tries to obtain an approximate solution of cardinality $k$ by using the information computed at Step 0 (the first, second and third smallest cost in each row and column). In a first phase (steps (a) and (b) below) we try to compute the solution using only first minima. If this fails, we try using second and third minima (step (c)). Parameter $g_h$ can be either $g_1$ or $g_2$.

*Algorithm GREEDY*($g_h$):

(a) $\bar{g}_h := g_h$;
find the first unassigned row, $i$, such that column $c(i)$ is unassigned and set $\tilde{\rho} := \rho_i$ ($\tilde{\rho} := +\infty$ if no such row);
find the first unassigned column, $j$, such that row $r(j)$ is unassigned and set $\tilde{\gamma} := \gamma_j$ ($\tilde{\gamma} := +\infty$ if no such column);

(b) **while** $\bar{g}_h < k$ **and** ($\tilde{\rho} \neq +\infty$ **or** $\tilde{\gamma} \neq +\infty$) **do**
  $\bar{g}_h := \bar{g}_h + 1$;
  **if** $\tilde{\rho} \leqslant \tilde{\gamma}$ **then**
    assign row $i$ to column $c(i)$;
    find the next unassigned row, $i$, such that column $c(i)$ is unassigned and set $\tilde{\rho} := \rho_i$ ($\tilde{\rho} := +\infty$ if no such row)
  **else**
    assign column $j$ to row $r(j)$;
    find the next unassigned column, $j$, such that row $r(j)$ is unassigned and set $\tilde{\gamma} := \gamma_j$ ($\tilde{\gamma} := +\infty$ if no such column)
  **endif**
**endwhile**;

(c) **if** $\bar{g}_h < k$ **then**
  execute steps similar to (a) and (b) above, but looking, at each search, for the first or next row $i$ (resp. column $j$) such that column $c'(i)$ or $c''(i)$ (resp. row $r'(j)$ or $r''(j)$) is unassigned, and setting
  $\tilde{\rho} := \rho'(i)$ if $c'(i)$ is unassigned, or $\tilde{\rho} := \rho''(i)$ otherwise
  (resp. $\tilde{\gamma} := \gamma'(j)$ if $r'(j)$ is unassigned, or $\tilde{\gamma} := \gamma''(j)$ otherwise).

(d) **if** $\bar{g}_h = k$ **then** $UB_h :=$ solution value **else** $UB_h := +\infty$;
**return** $\bar{g}_h$ and $UB_h$.

Since it is possible that both $\bar{g}_1$ and $\bar{g}_2$ are less than $k$, Step 3 of Section 2 was modified so as to have in any case an approximate solution of cardinality $k$, either through elements other than the first three minima, or, if this fails, through dummy elements. The resulting Step 3 is as follows.

Step 3. $UB$:=min($UB_1, UB_2$); $g$:=max($g_1, g_2$); $\bar{g}$:=max($\bar{g}_1, \bar{g}_2$);

**if** $\bar{g} = k$ **then go to** Step 4;
**for each** unassigned column $j$ **do**
  **if** $\exists\ w_{ij}$ such that $i$ is unassigned **then**
    assign row $i$ to column $j$, update $UB$ and set $\bar{g}$:=$\bar{g} + 1$;
    **if** $\bar{g} = k$ **then go to** Step 4
  **endif;**
**while** $\bar{g} < k$ **do**
  let $i$ be the first unassigned row and $j$ the first unassigned column;
  add to $W$ a dummy element $w_{ij} = M$ (a very large value);
  assign row $i$ to column $j$, update $UB$ and set $\bar{g}$:=$\bar{g} + 1$
**endwhile**.

## 4. Implementation

The overall algorithm for sparse matrices was coded in FORTRAN 77. The approximate solution of Section 3.2 was obtained using the FORTRAN implementation of the Dinic [7] algorithm proposed by Goldfarb and Grigoriadis [8].

The AP solution needed at the beginning of the primal phase was determined through a modified version of the Jonker and Volgenant [10] code SPJV for the sparse assignment problem. The original Pascal code was translated into FORTRAN and modified so as to: (i) handle our data structure for the current matrix $\bar{W}$ (see Section 3.1); (ii) manage the particular case arising when the new Step 3 (see Section 3.3) adds dummy elements to the cost matrix. These elements are not explicitly inserted in the data structure but implicitly considered by associating a flag to the rows having fictitious assignment.

The alternating paths needed, at each iteration of the primal phase, to re-optimize the current solution have been determined through the classical Dijkstra algorithm, implemented with a binary heap for the node labels. Two shortest path routines were coded: RHEAP (resp. CHEAP) starts from a given row (resp. column) and uses the forward (resp. backward) data structure of Section 3.1. Both routines implicitly consider the possible dummy elements (see above).

The overall structure of the algorithm is the following.

*Algorithm SKAP*:

**if** $|A| \leqslant 0.01\ mn$ **and** $k > 0.9\ m$ **then**
  execute the Dinic algorithm (Section 3.2) and set $LB$:= $-1$;
  **if** the resulting flow is less than $k$ **then return** "no solution"

**else**
perform the sparse version of Steps 0–2 of preprocessing (Section 2);
execute the new Step 3 (Section 3.3) and let $UB$ denote the resulting value;
execute the sparse version of Step 4 and let $LB$ denote the lower bound value;
**if** $UB = LB$ **then return** the solution **else** reduce the instance (Step 5)
**endif**;
initialize the data structure for matrix $\bar{W}$ (Section 3.1);
execute the modified SPJV code for AP and let $UB$ denote the solution value;
**if** $UB = LB$ **then return** the solution;
**for** $i$:=1 **to** $m - k$ **do**
add a row to $\bar{W}$ and re-optimize by executing RHEAP;
add a column to $\bar{W}$ and re-optimize by executing CHEAP
**endfor**;
**for** $i$:=1 **to** $n - m$ **do**
add a column to $\bar{W}$ and re-optimize by executing CHEAP;
**if** the solution includes dummy elements **then return** "no solution"
**else return** the solution.

### 4.1. Running the code

The algorithm was implemented as a FORTRAN 77 subroutine, called SKAP and available at http://www.or.deis.unibo.it/ORCodes/. The whole subroutine is completely self-contained and communication with it is achieved solely through the parameter list. The subroutine can be invoked with the statement

```
CALL SKAP(M,N,F_FIRST,F_COST,F_HEAD,B_FIRST,B_COST,B_TAIL,K,
    OPT,Z,ROW,COL,U,V,MU,F_COST2,B_COST2,F_HEAD2,B_TAIL2)
```

The input parameters are

M, N = number of rows and columns $(m, n)$;
F_FIRST, F_COST, F_HEAD = forward star data structure (see Section 3.1): these arrays must be dimensioned at least at $m + 1, |A|$ and $|A|$, respectively;
B_FIRST, B_COST, B_TAIL=backward star data structure (see Section 3.1): these arrays must be dimensioned at least at $n + 1, |A|$ and $|A|$, respectively;
K = cardinality of the required solution;

The output parameters are

OPT = return status: value 0 if the instance is infeasible; value 1 otherwise;
Z = optimal solution value;
ROW = the set of entries in the solution is {(ROW(j), j): ROW(j)>0, j=1,..., N}: this array must be dimensioned at least at $n$;
COL = the set of entries in the solution is {(i,COL(i)): COL(i)>0, i=1,..., M}: this array must be dimensioned at least at $m$.

Local arrays U, V (dimensioned at least at $m$ and $n$, respectively) and scalar MU are internally used to store the dual variables (see, Dell'Amico and Martello [5]); local

arrays F_COST2, F_HEAD2, B_COST2 and B_TAIL2 are internally used to store the restricted matrix (see Section 3.1), and must be dimensioned at least at $|A|$.

## 5. Computational experiments

Algorithm SKAP was computationally tested on a Silicon Graphics Indy R10000sc 195 MHz.

We compared its performance with the following indirect approach. Consider digraph $G' = (\{s\} \cup \{\bar{s}\} \cup U \cup V \cup \{t\}, A')$ introduced in Section 3.2, and assign cost $w_{ij}$ to each arc $(i,j) \in A$, and zero cost to the remaining arcs. It is then clear that sending a min-cost $s-t$ flow of value $k$ on this network will give the optimal solution to the corresponding $k$-AP. This approach was implemented by using two different codes for min-cost flow. The first one is the FORTRAN code RELAXT IV of Bertsekas and Tseng, see [2], called RLXT in the sequel, and the second one is the C code CS2 developed by Cherkassky and Goldberg, see Goldberg [3]. Parameter CRASH of RELAXT IV was set to one, since preliminary computational experiments showed that this value produces much smaller computing times for our instances.

Test problems were obtained by randomly generating the costs $w_{ij}$ from the uniform distributions $[0, 10^2]$ and $[0, 10^5]$. This kind of generation is often encountered in the literature for the evaluation of algorithms for AP.

For increasing values of $m$ and $n$ (up to 3000) and various degrees of density (1%, 5% and 10%), 10 instances were generated and solved for different values of $k$ (ranging from $\frac{20}{100}m$ to $\lfloor \frac{99}{100}m \rfloor$).

Density $d$% means that the instance was generated as follows. For each pair $(i,j)$, we first generate a uniformly random value $r$ in $[0,1)$: if $r \leqslant d$ then the entry is added to the instance by generating its value $w_{ij}$ in the considered range; otherwise the next pair is considered. In order to ensure consistency with the values of $m$ and $n$, we then consider each row $i$ and, if it has no entry, we uniformly randomly generate a column $j$ and a value $w_{ij}$. The same is finally done for columns.

Tables 1–3 show the results for sparse instances. In each block, each entry in the first six lines gives the average CPU time computed over 10 random instances, while the last line gives the average over all $k$ values considered. The results show that the best of the two min-cost flow approaches is faster than SKAP for some of the easy instances of Table 1, while both are clearly slower for all the remaining instances.

More specifically, for the 1% density (Table 1) RLXT is faster than SKAP in the range $[0, 10^2]$ for $m$ and $n$ less than 3000, while CS2 is faster than SKAP in the range $[0, 10^5]$ for $500 \leqslant m \leqslant 1000$ and $1000 \leqslant n \leqslant 2000$. It should be noted, however, that SKAP solves all these instances in less than one second on average, and that it is less sensitive to the cost range.

For the 5% instances (Table 2), SKAP is always the fastest code. For large values of $m$ and $n$ it is two–three times faster than its best competitor. This trend is confirmed by Table 3 (10% density), where, for large instances, the CPU times of SKAP are about one order of magnitude smaller than the others.

Table 1
Density 1%. Average computing times over 10 problems, Silicon Graphics INDY R10000sc seconds

| | | | Range $[0, 10^2]$ | | | Range $[0, 10^5]$ | | |
|---|---|---|---|---|---|---|---|---|
| $m$ | $n$ | $k\frac{100}{m}$ | RLXT | CS2 | SKAP | RLXT | CS2 | SKAP |
| | | 20 | 0.018 | 0.029 | 0.005 | 0.031 | 0.038 | 0.008 |
| | | 50 | 0.020 | 0.029 | 0.032 | 0.053 | 0.038 | 0.037 |
| | | 80 | 0.024 | 0.031 | 0.039 | 0.071 | 0.043 | 0.035 |
| 250 | 500 | 90 | 0.023 | 0.030 | 0.039 | 0.067 | 0.042 | 0.038 |
| | | 95 | 0.027 | 0.034 | 0.037 | 0.081 | 0.041 | 0.037 |
| | | 99 | 0.023 | 0.031 | 0.041 | 0.068 | 0.045 | 0.039 |
| | | Average | 0.023 | 0.031 | 0.032 | 0.062 | 0.041 | 0.032 |
| | | 20 | 0.033 | 0.049 | 0.056 | 0.082 | 0.062 | 0.063 |
| | | 50 | 0.054 | 0.056 | 0.061 | 0.155 | 0.064 | 0.068 |
| | | 80 | 0.070 | 0.057 | 0.078 | 0.218 | 0.068 | 0.087 |
| 500 | 500 | 90 | 0.067 | 0.055 | 0.077 | 0.197 | 0.073 | 0.088 |
| | | 95 | 0.076 | 0.058 | 0.073 | 0.165 | 0.074 | 0.077 |
| | | 99 | 0.084 | 0.061 | 0.050 | 0.152 | 0.078 | 0.052 |
| | | Average | 0.064 | 0.056 | 0.066 | 0.162 | 0.070 | 0.073 |
| | | 20 | 0.045 | 0.090 | 0.029 | 0.189 | 0.120 | 0.061 |
| | | 50 | 0.059 | 0.096 | 0.115 | 0.171 | 0.110 | 0.126 |
| | | 80 | 0.075 | 0.095 | 0.118 | 0.227 | 0.125 | 0.141 |
| 500 | 1000 | 90 | 0.077 | 0.097 | 0.131 | 0.316 | 0.120 | 0.164 |
| | | 95 | 0.072 | 0.100 | 0.182 | 0.364 | 0.122 | 0.202 |
| | | 99 | 0.077 | 0.099 | 0.153 | 0.326 | 0.123 | 0.159 |
| | | Average | 0.068 | 0.096 | 0.121 | 0.266 | 0.120 | 0.142 |
| | | 20 | 0.089 | 0.156 | 0.231 | 0.358 | 0.204 | 0.231 |
| | | 50 | 0.178 | 0.171 | 0.227 | 0.605 | 0.205 | 0.256 |
| | | 80 | 0.200 | 0.181 | 0.294 | 0.987 | 0.219 | 0.351 |
| 1000 | 1000 | 90 | 0.199 | 0.188 | 0.271 | 1.109 | 0.232 | 0.326 |
| | | 95 | 0.218 | 0.200 | 0.239 | 0.875 | 0.229 | 0.300 |
| | | 99 | 0.237 | 0.217 | 0.142 | 0.796 | 0.244 | 0.202 |
| | | Average | 0.187 | 0.186 | 0.234 | 0.788 | 0.222 | 0.278 |
| | | 20 | 0.191 | 0.399 | 0.323 | 1.695 | 0.489 | 0.448 |
| | | 50 | 0.196 | 0.430 | 0.446 | 0.911 | 0.489 | 0.471 |
| | | 80 | 0.330 | 0.435 | 0.455 | 1.344 | 0.497 | 0.571 |
| 1000 | 2000 | 90 | 0.257 | 0.439 | 0.519 | 1.530 | 0.511 | 0.683 |
| | | 95 | 0.270 | 0.461 | 0.735 | 1.688 | 0.506 | 0.779 |
| | | 99 | 0.249 | 0.454 | 0.612 | 1.831 | 0.493 | 0.615 |
| | | Average | 0.249 | 0.436 | 0.515 | 1.500 | 0.498 | 0.595 |
| | | 20 | 0.477 | 1.015 | 0.905 | 2.485 | 1.190 | 0.913 |
| | | 50 | 0.599 | 1.047 | 0.858 | 3.264 | 1.234 | 1.029 |
| | | 80 | 0.867 | 1.203 | 1.078 | 7.862 | 1.315 | 1.620 |
| 2000 | 2000 | 90 | 0.876 | 1.256 | 1.007 | 6.087 | 1.405 | 1.435 |
| | | 95 | 0.883 | 1.333 | 0.895 | 5.828 | 1.432 | 1.312 |
| | | 99 | 0.900 | 1.427 | 0.503 | 4.274 | 1.547 | 0.893 |
| | | Average | 0.767 | 1.214 | 0.874 | 4.967 | 1.354 | 1.200 |

Table 1 (*continued*)

| | | | Range $[0, 10^2]$ | | | Range $[0, 10^5]$ | | |
|---|---|---|---|---|---|---|---|---|
| $m$ | $n$ | $k\frac{100}{m}$ | RLXT | CS2 | SKAP | RLXT | CS2 | SKAP |
| | | 20 | 0.928 | 1.078 | 0.038 | 7.412 | 1.410 | 1.026 |
| | | 50 | 0.795 | 1.271 | 0.996 | 3.714 | 1.511 | 1.077 |
| | | 80 | 0.613 | 1.307 | 0.983 | 1.723 | 1.490 | 1.262 |
| 1500 | 3000 | 90 | 0.754 | 1.370 | 1.112 | 2.656 | 1.560 | 1.673 |
| | | 95 | 0.752 | 1.363 | 1.476 | 3.448 | 1.570 | 1.853 |
| | | 99 | 0.663 | 1.340 | 1.252 | 2.803 | 1.551 | 1.420 |
| | | Average | 0.751 | 1.288 | 0.976 | 3.626 | 1.515 | 1.385 |
| | | 20 | 1.088 | 2.476 | 0.069 | 7.119 | 2.941 | 2.083 |
| | | 50 | 1.431 | 2.743 | 2.012 | 5.509 | 3.220 | 2.387 |
| | | 80 | 2.826 | 3.064 | 2.543 | 28.727 | 3.397 | 3.926 |
| 3000 | 3000 | 90 | 2.239 | 3.207 | 2.328 | 13.689 | 3.594 | 3.428 |
| | | 95 | 2.160 | 3.438 | 1.975 | 14.763 | 3.766 | 3.034 |
| | | 99 | 2.096 | 3.656 | 1.194 | 17.572 | 3.950 | 1.973 |
| | | Average | 1.973 | 3.097 | 1.687 | 14.563 | 3.478 | 2.805 |

Table 2
Density 5%. Average computing times over 10 problems, Silicon Graphics INDY R10000sc seconds

| | | | Range $[0, 10^2]$ | | | Range $[0, 10^5]$ | | |
|---|---|---|---|---|---|---|---|---|
| $m$ | $n$ | $k\frac{100}{m}$ | RLXT | CS2 | SKAP | RLXT | CS2 | SKAP |
| | | 20 | 0.035 | 0.076 | 0.005 | 0.168 | 0.091 | 0.022 |
| | | 50 | 0.034 | 0.080 | 0.032 | 0.089 | 0.093 | 0.039 |
| | | 80 | 0.040 | 0.082 | 0.042 | 0.080 | 0.107 | 0.051 |
| 250 | 500 | 90 | 0.035 | 0.075 | 0.049 | 0.155 | 0.092 | 0.053 |
| | | 95 | 0.040 | 0.080 | 0.049 | 0.102 | 0.107 | 0.057 |
| | | 99 | 0.043 | 0.077 | 0.043 | 0.072 | 0.093 | 0.049 |
| | | Average | 0.038 | 0.078 | 0.037 | 0.111 | 0.097 | 0.045 |
| | | 20 | 0.054 | 0.145 | 0.048 | 0.225 | 0.176 | 0.070 |
| | | 50 | 0.063 | 0.147 | 0.078 | 0.176 | 0.186 | 0.088 |
| | | 80 | 0.098 | 0.155 | 0.101 | 0.560 | 0.193 | 0.122 |
| 500 | 500 | 90 | 0.111 | 0.168 | 0.107 | 0.489 | 0.199 | 0.125 |
| | | 95 | 0.114 | 0.170 | 0.093 | 0.497 | 0.204 | 0.108 |
| | | 99 | 0.114 | 0.176 | 0.066 | 0.388 | 0.211 | 0.085 |
| | | Average | 0.092 | 0.160 | 0.082 | 0.389 | 0.195 | 0.100 |
| | | 20 | 0.176 | 0.358 | 0.016 | 1.376 | 0.450 | 0.099 |
| | | 50 | 0.165 | 0.405 | 0.137 | 0.721 | 0.501 | 0.144 |
| | | 80 | 0.165 | 0.419 | 0.153 | 0.419 | 0.508 | 0.200 |
| 500 | 1000 | 90 | 0.170 | 0.432 | 0.168 | 0.326 | 0.499 | 0.237 |
| | | 95 | 0.172 | 0.424 | 0.180 | 0.302 | 0.502 | 0.223 |
| | | 99 | 0.171 | 0.435 | 0.173 | 0.320 | 0.488 | 0.198 |
| | | Average | 0.170 | 0.412 | 0.138 | 0.577 | 0.491 | 0.184 |

Table 2 (*continued*)

| | | | Range $[0, 10^2]$ | | | Range $[0, 10^5]$ | | |
|---|---|---|---|---|---|---|---|---|
| $m$ | $n$ | $k\frac{100}{m}$ | RLXT | CS2 | SKAP | RLXT | CS2 | SKAP |
| | | 20 | 0.175 | 0.928 | 0.028 | 1.360 | 1.135 | 0.263 |
| | | 50 | 0.296 | 1.035 | 0.267 | 1.213 | 1.230 | 0.339 |
| | | 80 | 0.529 | 1.126 | 0.365 | 4.883 | 1.285 | 0.584 |
| 1000 | 1000 | 90 | 0.583 | 1.165 | 0.389 | 3.024 | 1.315 | 0.572 |
| | | 95 | 0.586 | 1.215 | 0.361 | 4.574 | 1.365 | 0.486 |
| | | 99 | 0.611 | 1.257 | 0.274 | 3.728 | 1.428 | 0.357 |
| | | Average | 0.463 | 1.121 | 0.281 | 3.130 | 1.293 | 0.434 |
| | | 20 | 1.044 | 2.161 | 0.045 | 9.425 | 2.560 | 0.524 |
| | | 50 | 0.735 | 2.430 | 0.052 | 4.206 | 2.862 | 0.549 |
| | | 80 | 0.994 | 2.700 | 0.582 | 2.404 | 3.065 | 0.840 |
| 1000 | 2000 | 90 | 1.068 | 2.752 | 0.621 | 1.841 | 3.070 | 1.042 |
| | | 95 | 0.970 | 2.772 | 0.667 | 1.519 | 2.981 | 1.011 |
| | | 99 | 0.912 | 2.652 | 0.675 | 1.900 | 2.918 | 0.830 |
| | | Average | 0.954 | 2.578 | 0.440 | 3.549 | 2.909 | 0.799 |
| | | 20 | 1.914 | 4.775 | 0.085 | 6.455 | 5.419 | 1.054 |
| | | 50 | 2.115 | 5.460 | 0.173 | 4.212 | 6.062 | 1.483 |
| | | 80 | 2.676 | 6.048 | 1.502 | 33.195 | 6.475 | 3.074 |
| 2000 | 2000 | 90 | 3.624 | 6.334 | 1.778 | 29.435 | 6.662 | 2.648 |
| | | 95 | 3.661 | 6.598 | 1.646 | 35.922 | 6.759 | 2.261 |
| | | 99 | 3.132 | 6.943 | 1.237 | 36.167 | 7.402 | 1.671 |
| | | Average | 2.854 | 6.026 | 1.070 | 24.231 | 6.463 | 2.032 |
| | | 20 | 3.296 | 5.459 | 0.099 | 29.304 | 6.607 | 1.184 |
| | | 50 | 1.406 | 5.958 | 0.098 | 13.933 | 7.282 | 1.249 |
| | | 80 | 3.062 | 6.531 | 1.296 | 6.015 | 7.462 | 2.213 |
| 1500 | 3000 | 90 | 2.207 | 6.898 | 1.383 | 4.832 | 7.594 | 2.997 |
| | | 95 | 2.529 | 6.932 | 1.488 | 4.720 | 7.447 | 2.763 |
| | | 99 | 2.345 | 6.613 | 1.488 | 5.935 | 7.179 | 2.097 |
| | | Average | 2.474 | 6.399 | 0.975 | 10.790 | 7.262 | 2.084 |
| | | 20 | 9.503 | 11.850 | 0.171 | 14.187 | 13.428 | 2.395 |
| | | 50 | 3.867 | 12.669 | 0.175 | 9.093 | 13.938 | 3.710 |
| | | 80 | 12.990 | 15.402 | 3.757 | 74.084 | 16.270 | 8.534 |
| 3000 | 3000 | 90 | 12.098 | 15.735 | 4.696 | 94.893 | 16.925 | 7.384 |
| | | 95 | 10.224 | 16.366 | 4.446 | 96.993 | 17.938 | 5.911 |
| | | 99 | 7.818 | 17.444 | 3.581 | 105.391 | 19.175 | 4.629 |
| | | Average | 9.417 | 14.911 | 2.804 | 65.774 | 16.279 | 5.427 |

In Table 4, we consider complete matrices and compare the previous approaches and the specialized algorithm PRML, developed, for dense $k$-APs, by Dell'Amico and Martello [5]. The results show that SKAP is clearly superior to both min-cost flow

Table 3
Density 10%. Average computing times over 10 problems, Silicon Graphics INDY R10000sc seconds

| | | | Range $[0, 10^2]$ | | | Range $[0, 10^5]$ | | |
|---|---|---|---|---|---|---|---|---|
| $m$ | $n$ | $k\frac{100}{m}$ | RLXT | CS2 | SKAP | RLXT | CS2 | SKAP |
| | | 20 | 0.062 | 0.126 | 0.006 | 0.479 | 0.157 | 0.027 |
| | | 50 | 0.062 | 0.140 | 0.044 | 0.243 | 0.172 | 0.045 |
| | | 80 | 0.074 | 0.147 | 0.049 | 0.123 | 0.170 | 0.058 |
| 250 | 500 | 90 | 0.071 | 0.141 | 0.062 | 0.103 | 0.166 | 0.069 |
| | | 95 | 0.065 | 0.144 | 0.061 | 0.105 | 0.170 | 0.066 |
| | | 99 | 0.066 | 0.150 | 0.062 | 0.118 | 0.171 | 0.064 |
| | | Average | 0.067 | 0.141 | 0.047 | 0.195 | 0.168 | 0.055 |
| | | 20 | 0.069 | 0.338 | 0.011 | 0.380 | 0.421 | 0.065 |
| | | 50 | 0.109 | 0.372 | 0.080 | 0.719 | 0.442 | 0.098 |
| | | 80 | 0.180 | 0.380 | 0.129 | 1.194 | 0.450 | 0.183 |
| 500 | 500 | 90 | 0.207 | 0.393 | 0.133 | 1.099 | 0.464 | 0.175 |
| | | 95 | 0.211 | 0.414 | 0.130 | 1.104 | 0.476 | 0.166 |
| | | 99 | 0.219 | 0.423 | 0.097 | 0.794 | 0.478 | 0.124 |
| | | Average | 0.166 | 0.387 | 0.097 | 0.882 | 0.455 | 0.135 |
| | | 20 | 0.362 | 0.848 | 0.027 | 3.299 | 1.068 | 0.116 |
| | | 50 | 0.267 | 0.932 | 0.022 | 1.587 | 1.166 | 0.163 |
| | | 80 | 0.399 | 1.029 | 0.168 | 0.894 | 1.188 | 0.236 |
| 500 | 1000 | 90 | 0.365 | 1.045 | 0.210 | 0.692 | 1.202 | 0.321 |
| | | 95 | 0.356 | 1.039 | 0.222 | 0.550 | 1.209 | 0.313 |
| | | 99 | 0.347 | 1.030 | 0.237 | 0.488 | 1.167 | 0.259 |
| | | Average | 0.349 | 0.987 | 0.148 | 1.252 | 1.167 | 0.235 |
| | | 20 | 0.491 | 1.951 | 0.037 | 2.177 | 2.335 | 0.303 |
| | | 50 | 0.642 | 2.155 | 0.069 | 2.122 | 2.548 | 0.382 |
| | | 80 | 0.892 | 2.503 | 0.474 | 9.234 | 2.722 | 0.843 |
| 1000 | 1000 | 90 | 1.085 | 2.578 | 0.569 | 8.973 | 2.789 | 0.817 |
| | | 95 | 1.218 | 2.546 | 0.546 | 8.413 | 2.869 | 0.767 |
| | | 99 | 1.107 | 2.748 | 0.436 | 6.904 | 3.049 | 0.577 |
| | | Average | 0.906 | 2.414 | 0.355 | 6.304 | 2.719 | 0.615 |
| | | 20 | 1.980 | 4.610 | 0.070 | 25.197 | 5.472 | 0.501 |
| | | 50 | 1.009 | 4.933 | 0.080 | 7.292 | 5.796 | 0.639 |
| | | 80 | 1.898 | 5.400 | 0.583 | 4.349 | 5.962 | 0.996 |
| 1000 | 2000 | 90 | 1.840 | 5.559 | 0.760 | 4.198 | 5.948 | 1.630 |
| | | 95 | 1.596 | 5.518 | 0.839 | 4.648 | 5.832 | 1.587 |
| | | 99 | 1.671 | 5.397 | 0.872 | 3.666 | 5.745 | 1.191 |
| | | Average | 1.666 | 5.236 | 0.534 | 8.225 | 5.793 | 1.091 |
| | | 20 | 6.281 | 9.746 | 0.133 | 9.597 | 10.952 | 1.216 |
| | | 50 | 2.105 | 10.495 | 0.149 | 7.715 | 11.145 | 1.670 |
| | | 80 | 6.176 | 12.512 | 2.019 | 46.621 | 12.472 | 4.882 |
| 2000 | 2000 | 90 | 7.773 | 12.580 | 3.071 | 66.271 | 13.377 | 4.491 |
| | | 95 | 8.017 | 13.426 | 2.624 | 49.976 | 13.855 | 3.745 |
| | | 99 | 5.773 | 14.087 | 2.317 | 67.158 | 14.363 | 2.762 |
| | | Average | 6.021 | 12.141 | 1.719 | 41.223 | 12.694 | 3.128 |

Table 3 (*continued*)

| $m$ | $n$ | $k\frac{100}{m}$ | Range $[0, 10^2]$ | | | Range $[0, 10^5]$ | | |
|---|---|---|---|---|---|---|---|---|
| | | | RLXT | CS2 | SKAP | RLXT | CS2 | SKAP |
| | | 20 | 5.124 | 11.413 | 0.155 | 54.703 | 12.920 | 1.232 |
| | | 50 | 3.890 | 12.410 | 0.161 | 20.098 | 13.715 | 1.427 |
| | | 80 | 7.012 | 13.495 | 0.168 | 12.327 | 14.986 | 2.510 |
| 1500 | 3000 | 90 | 4.381 | 13.674 | 1.491 | 9.505 | 14.774 | 4.529 |
| | | 95 | 5.027 | 14.092 | 1.880 | 14.312 | 14.470 | 4.390 |
| | | 99 | 4.666 | 13.821 | 1.957 | 14.103 | 14.433 | 3.168 |
| | | Average | 5.017 | 13.151 | 0.969 | 20.841 | 14.216 | 2.876 |
| | | 20 | 27.022 | 24.249 | 0.290 | 18.440 | 26.891 | 2.789 |
| | | 50 | 10.271 | 25.223 | 0.312 | 14.506 | 26.967 | 3.907 |
| | | 80 | 15.036 | 28.650 | 5.850 | 117.946 | 29.787 | 13.780 |
| 3000 | 3000 | 90 | 73.373 | 31.067 | 10.134 | 141.552 | 33.177 | 12.530 |
| | | 95 | 25.598 | 34.414 | 11.014 | 137.860 | 35.357 | 10.124 |
| | | 99 | 19.422 | 34.894 | 7.139 | 158.249 | 36.366 | 7.952 |
| | | Average | 28.454 | 29.750 | 5.790 | 98.092 | 31.424 | 8.514 |

Table 4
Complete matrices. Average computing times over 10 problems, Silicon Graphics INDY R10000sc seconds

| $m$ | $n$ | $k\frac{100}{m}$ | Range $[0, 10^2]$ | | | | Range $[0, 10^5]$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | RLXT | CS2 | PRML | SKAP | RLXT | CS2 | PRML | SKAP |
| | | 20 | 0.263 | 2.323 | 0.029 | 0.032 | 5.838 | 2.577 | 0.043 | 0.063 |
| | | 50 | 0.347 | 2.431 | 0.040 | 0.039 | 2.594 | 2.806 | 0.117 | 0.128 |
| | | 80 | 0.403 | 2.543 | 0.046 | 0.041 | 2.279 | 2.846 | 0.212 | 0.145 |
| 250 | 500 | 90 | 0.482 | 2.585 | 0.049 | 0.052 | 2.678 | 2.905 | 0.297 | 0.175 |
| | | 95 | 0.498 | 2.645 | 0.122 | 0.119 | 2.782 | 2.811 | 0.386 | 0.182 |
| | | 99 | 0.503 | 2.693 | 0.132 | 0.136 | 2.777 | 2.938 | 0.457 | 0.243 |
| | | Average | 0.416 | 2.537 | 0.070 | 0.070 | 3.158 | 2.814 | 0.252 | 0.156 |
| | | 20 | 1.255 | 5.042 | 0.074 | 0.068 | 2.923 | 5.370 | 0.131 | 0.221 |
| | | 50 | 1.224 | 5.197 | 0.083 | 0.076 | 4.087 | 5.549 | 0.295 | 0.278 |
| | | 80 | 1.451 | 5.424 | 0.262 | 0.253 | 16.758 | 6.003 | 0.860 | 0.609 |
| 500 | 500 | 90 | 1.958 | 5.528 | 0.281 | 0.730 | 16.187 | 6.044 | 1.186 | 1.116 |
| | | 95 | 2.732 | 5.876 | 0.292 | 0.775 | 17.136 | 6.631 | 1.318 | 1.062 |
| | | 99 | 2.691 | 6.195 | 0.347 | 0.784 | 17.211 | 6.764 | 1.458 | 0.904 |
| | | Average | 1.885 | 5.544 | 0.223 | 0.448 | 12.384 | 6.060 | 0.875 | 0.698 |

approaches. In addition, in many cases, it is even faster than the specialized code. This is probably due to the new heuristic algorithm embedded in SKAP.

Tables 1–4 also give information about the relative efficiency of codes RELAXT IV and CS2, when used on our type of instances. It is quite evident that the two codes are sensitive to the cost range: RELAXT IV is clearly faster than CS2 for the range $[0, 10^2]$, while the opposite holds for the range $[0, 10^5]$.

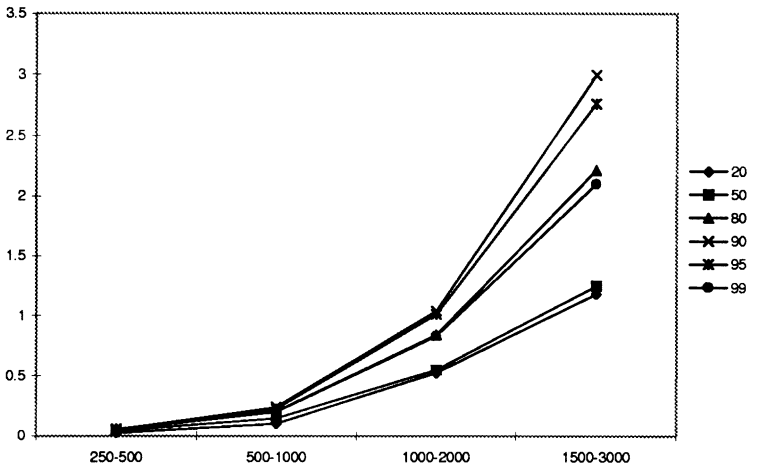



Fig. 2. Density 5%, range $[0, 10^5]$, rectangular instances. Average CPU time (in seconds) as a function of the instance size.

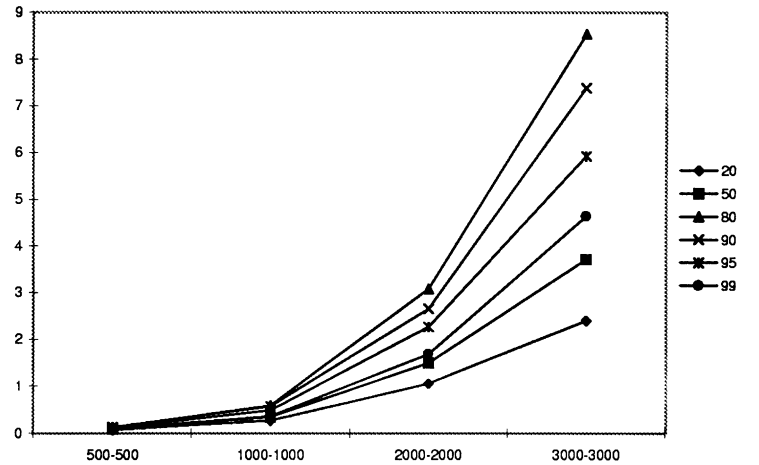



Fig. 3. Density 5%, range $[0, 10^5]$, square instances. Average CPU time (in seconds) as a function of the instance size.

Finally, Figs. 2 and 3 show the behavior of the average CPU time requested by SKAP, for each value of $k$, as a function of the instance size. We consider the case of range $[0, 10^5]$ and 5% density, separately for rectangular and square instances. The functions show that SKAP has, for fixed $k$ value, a regular behavior. There is instead no regular dependence of the CPU times on the value of $k$: small values produce easy instances, while the most difficult instances are encountered for $k$ around $\frac{80}{100}m$ or $\frac{90}{100}m$.

## Acknowledgements

This work was supported by Ministero dell'Università e della Ricerca Scientifica e Tecnologica, and by Consiglio Nazionale delle Ricerche, Italy.

## References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[2] D.P. Bertsekas, P. Tseng, Relaxation methods for minimum cost ordinary and generalized network flow problems, Oper. Res. 36 (1988) 93–114.
[3] A.V. Goldberg, An efficient implementation of a scaling minimum-cost flow algorithm, J. Algorithms 22 (1997) 1–29.
[4] M. Dell'Amico, F. Maffioli, M. Trubian, New bounds for optimum traffic assignment in satellite communication, Comput. Oper. Res. 25 (1998) 729–743.
[5] M. Dell'Amico, S. Martello, The $k$-cardinality assignment problem, Discrete Appl. Math. 76 (1997) 103–121.
[6] 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.
[7] E.A. Dinic, Algorithms for solution of a problem of maximum flow in networks with power estimation, Soviet Math. Dokl. 11 (1970) 1277–1280.
[8] D. Goldfarb, M.D. Grigoriadis, A computational comparison of the Dinic and network simplex methods for maximum flow, in: B. Simeone, P. Toth, G. Gallo, F. Maffioli, S. Pallottino (Eds.), FORTRAN Codes for Network Optimization, Annals of Operational Research, Vol. 13, Baltzer, Basel, 1988, pp. 83–103.
[9] R. Jain, J. Werth, J.C. Browne, A note on scheduling problems arising in satellite communication, J. Oper. Res. Soc. 48 (1997) 100–102.
[10] R. Jonker, T. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems, Computing 38 (1987) 325–340.
[11] E.L. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart & Winston, New York, 1976.
[12] S. Martello, P. Toth, Linear assignment problems, in: S. Martello et al. (Eds.), Surveys in Combinatorial Optimization, Annals of Discrete Mathematics, Vol. 31, North-Holland, Amsterdam, 1987, pp. 259–282.
[13] C.A. Pomalaza-Ráez, A note on efficient SS/TDMA assignment algorithms, IEEE Trans. Commun. 36 (1988) 1078–1082.
[14] C. Prins, An overview of scheduling problems arising in satellite communications, J. Oper. Res. Soc. 45 (1994) 611–623.