An algorithm for ranking assignments using reoptimization

Christian Roed Pedersen
Department of Operations Research
University of Aarhus
Ny Munkegade, Building 1530
DK-8000 Aarhus C
Denmark

Lars Relund Nielsen
Research Unit of Statistics and Decision Analysis
Department of Genetics and Biotechnology
University of Aarhus
P.O. Box 50
DK-8830 Tjele
Denmark

Kim Allan Andersen[*]
Department of Business Studies
University of Aarhus
Fuglesangs Allé 4
DK-8210 Aarhus V
Denmark

May 9, 2007

Abstract

We consider the problem of ranking assignments according to cost in the classical linear assignment problem. An algorithm partitioning the set of possible assignments, as suggested by Murty, is presented where, for each partition, the optimal assignment is calculated using a new reoptimization technique. Its computational performance is compared with all available implementations of algorithms with the same time complexity. The results are encouraging.

Keywords: Linear assignment problem, Ranking, K best solutions.

1 Introduction

The linear *assignment problem* (AP) is a well-known problem and may be considered as the problem of assigning n workers to n jobs. Each worker must be assigned to exactly one

[*]Corresponding author, e-mail: kia@asb.dk.

job. The objective is to minimize total cost.

In an annotated bibliography authored by Dell'Amico and Martello [5], more than 100 papers on the problem are mentioned. Kuhn [10, 11] suggested the first polynomial method for the solution of AP, called the *Hungarian method* with $\mathcal{O}(n^4)$ complexity. Since 1955 several other algorithms for AP have been developed. Some of the most efficient algorithms are the class of *successive shortest path procedures*[1] with an $\mathcal{O}(n^3)$ complexity (see e.g. Tomizawa [22] and Jonker and Volgenant [9]). An excellent survey is given by Dell'Amico and Toth [6] including comparative tests of several implementations of different AP algorithms.

As for other optimization problems, the assignment problem can be generalized to *ranking the best K assignments* in nondecreasing order of cost. Applications of ranking assignments are numerous. First, ranking assignments can be used to generate near optimal solutions which may be better from a practical point of view. Second, applications of AP may include constraints which are hard to specify formally and makes the problem difficult to optimize. Here an optimal assignment can be found by enumerating suboptimal assignments until an assignment satisfying the complicating constraints is found. Last but not least, ranking assignments appears as a subproblem within algorithms for solving the *bicriterion assignment problem* and related extensions, see for example Pedersen, Nielsen, and Andersen [17].

Several algorithms for ranking assignments have been suggested. They may be classified using two identifiers: a specific *branching technique* is used to partition the set of possible assignments into smaller subsets, and a *solution method* is used to find an optimal assignment for each subset.

Murty [13] suggested a branching technique where the set of possible assignments is partitioned into at most $n-1$ disjoint subsets for each additional ranking made. The Hungarian algorithm was used to find the best assignment for each subset resulting in an $\mathcal{O}(Kn^5)$ complexity. However, applying a successive shortest path procedure improves the overall complexity to $\mathcal{O}(Kn^4)$. Later, the branching technique in [13] was applied in the more general framework of finding the K best solutions to a discrete optimization problem by Lawler [12].

Hamacher and Queyranne [8] presented an alternative general framework for ranking solutions to combinatorial problems, later specialized for bipartite matchings by Chegireddy and Hamacher [4]. There an alternative branching technique is suggested, partitioning the current set into at most two subsets for each additional ranking. For each subset, the second best assignment has to be calculated. Different solution methods are suggested. One consists of identifying the second best assignment by a shortest cycle determination in an auxiliary network. The shortest cycle can be found by solving at most n shortest path problems resulting in an overall $\mathcal{O}(Kn^3)$ time complexity, the best known so far.

Recently, Pascoal, Captivo, and Clímaco [16] presented a ranking algorithm with the same branching technique as in [13]. However, by considering the subsets in reverse order when applying their solution method, they are able to reoptimize the solution from the

[1]Also known as shortest augmenting paths algorithms.

previous subset considered and find the best assignment by solving a single shortest path problem yielding the same time complexity as in [4].

The solution methods in all the above ranking algorithms use shortest path methods to find the best assignment for each subset. Methods based on shortest paths are *dual algorithms*. Dual feasibility exists and primal feasibility has to be reached. Tomizawa [22] noted that the original costs in the assignment may be replaced with the reduced costs when using successive shortest path procedures. Since the reduced costs are non-negative, the shortest path may be found using the algorithm of Dijkstra [7].

In spite of the connection between the dual variables and successive shortest path procedures, no one has considered updating the dual variables of the previous solution before the shortest path procedure is applied to a subset. We shall see that such an update yields an improvement in computational performance to the overall algorithm for ranking assignments. The new algorithm presented in this paper uses the branching technique of Murty [13]. For each subset, a solution method is used where only one single shortest path problem has to be solved. Hence, the overall time complexity of the proposed method is the same as in [4].

The main contributions of this paper can be shortly summarized as follows:

1. A new ranking algorithm for assignment problems using reoptimization is presented. Its time complexity is $\mathcal{O}(Kn^3)$.

2. The computational performance of our algorithm when the dual variables are updated before reoptimization is compared to the case where the dual variables are not updated. We show that updating yields a significant improvement.

3. We point out the impact on computational performance for different implementations of the ranking algorithm.

4. Comparative tests against other ranking algorithms known from the literature with complexity $\mathcal{O}(Kn^3)$ are carried out.

The paper is organized as follows. In Section 2 we provide the preliminaries. In Section 3 the new ranking algorithm is presented, and in Section 4 computational experiments are given.

2 Preliminaries

Let $G = (U \cup V, A)$ be a bipartite directed graph with node sets $U = V = \{1, \dots, n\}$, $m = n^2$ arcs in A and with cost c_{ij} on arc (i, j). Note that non-existing arcs can be represented as arcs having infinite cost. The *assignment problem* (AP) consists in assigning – with minimum total cost – each node in U to a node in V.

Defining the variables

$$x_{ij} = \begin{cases} 1, & \text{if node } i \text{ is assigned to node } j \\ 0, & \text{otherwise} \end{cases}$$

AP can, due to its totally unimodular constraint matrix, be formulated as the following continuous linear program.

$$\begin{aligned} \min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & \\ \text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} = 1, & i = 1,\dots,n \\ & \sum_{i=1}^{n} x_{ij} = 1, & j = 1,\dots,n \\ & x_{ij} \geq 0 & i,j = 1,\dots,n \end{aligned} \tag{1}$$

A feasible solution x to (1) is called an *assignment.* Using the network formulation, an assignment may alternatively be written as $a = \{(1, j_1), \dots, (n, j_n)\}$ where $(i, j) \in a$ if and only if $x_{ij} = 1$. A *partial primal solution* is a solution in which less than n variables are assigned value one and the constraints in (1) are satisfied with a $\leq$ sign instead of equality. Note that a partial primal solution corresponds to a *partial assignment* $a = \{(i_1, j_{i_1}), \dots, (i_q, j_{i_q})\}$.

By associating *dual variables* u_i and v_j with the constraints above, the corresponding dual problem becomes

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \\ \text{s.t.} \quad & u_i + v_j \leq c_{ij}, \qquad i,j = 1,\dots,n \end{aligned} \tag{2}$$

Given the reduced costs $\bar{c}_{ij} = c_{ij} - u_i - v_j$, $\forall\,(i,j) \in A$, *the complementary slackness optimality conditions* become

$$x_{ij}\bar{c}_{ij} = 0,\ \forall\,(i,j) \in A\ . \tag{3}$$

2.1 The successive shortest path procedure

Successive shortest path procedures for AP are dual methods. Dual feasibility exists and the optimal solution is built step by step by iteratively adding assignments to a current partial primal solution.

A successive shortest path procedure consists of two phases. In phase one, the cost matrix $[c_{ij}]$ is preprocessed and a partial primal solution (or partial assignment) and a dual feasible solution satisfying the complementary slackness optimality conditions (3) are determined. In phase two, the partial primal solution is augmented by adding one row/column assignment at a time until the solution becomes feasible. At each step in phase two, the dual solution is updated so that complementary slackness still holds. Hence, at the end of the second phase, the current primal and dual solutions are optimal.

```
procedure SuccSP()
    $(a, u, v)$ := Preprocess($[c_{ij}]$);
    while ($|a| < n$) do
        $\bar{G}(a)$ := BuildResNetwork($a, u, v$);
        $P$ := FindAugmentPath($\bar{G}(a)$);
        $a$ := AugmentSolution($P$);
        $(u, v)$ := AdjustDualSolution($P$);
    end while
end procedure
```

Figure 1: The successive shortest path procedure.

A pseudo-code for the successive shortest path procedure is given in Figure 1. Phase one is executed by function `Preprocess` which returns a partial assignment a and a dual feasible solution (u, v) satisfying (3). Phase two is executed on lines 3–8.

If $|a| < n$, then all nodes in U have not been assigned to a node in V and function `BuildResNetwork` builds the directed *residual network* $\bar{G}(a) = (U \cup V, A_f \cup A_b)$ constructed from G and the current partial solution a. In accordance with the introduction of the residual network in [1], let

$$A_f = \{(i,j) \ : \ (i,j) \in A \ \wedge \ (i,j) \notin a\} \text{ and}$$
$$A_b = \{(j,i) \ : \ (i,j) \in A \ \wedge \ (i,j) \in a\} \ .$$

Each *forward arc* (i, j) in A_f is assigned reduced cost $\bar{c}_{ij}$ and each *backward arc* (j, i) in A_b is assigned cost $-\bar{c}_{ij} = 0$ due to (3).

It is easy to see that any directed path in $\bar{G}(a)$ contains an arc in A_f and an arc in A_b, alternatingly. Such paths are called *alternating paths*. If the directed path P starts in an unassigned node in U and terminates with an unassigned node in V, it is called an *augmenting path*.

It is well known that, by removing assignments in a corresponding to the backward arcs in P and adding the forward arcs in P to a, the number of assignments in the resulting (partial) assignment $\bar{a}$ increases by one. Furthermore, since AP is a special instance of the minimum cost flow problem, the following result can be derived from the general optimality results given by Ahuja et al. [1].

Theorem 1. *Let a be a partial assignment and (u, v) the corresponding dual variables fulfilling the complementary slackness optimality conditions. Let P in $\bar{G}(a)$ be a shortest augmenting path and set*

$$\bar{a} = (a \setminus [P \cap A_b]) \cup (P \cap A_f) \ . \tag{4}$$

Then $\bar{a}$ is a minimum cost (partial) assignment with $|a| + 1$ assignments.

Hence, finding a minimum cost (partial) assignment $\bar{a}$ with $|a| + 1$ assignments can be done by finding the shortest augmenting path in $\bar{G}(a)$. As a consequence, AP can be solved by identifying at most n successive shortest augmenting paths. Since the reduced costs are non-negative, each path can be determined by Dijkstra's method running in $\mathcal{O}(n^2)$ time. Therefore, the overall computational complexity of a successive shortest path procedure is $\mathcal{O}(n^3)$.

In procedure `SuccSP` the shortest augmenting path P is found using function `Find-AugmentPath` and next the (partial) assignment a is updated as in (4) using function `Aug-mentSolution`. Finally, the dual variables are updated using function `AdjustDualSolution` such that (3) holds. For an efficient implementation of procedure `SuccSP` see for instance Jonker and Volgenant [9].

3 Ranking assignments

Consider the problem of ranking the best K assignments in nondecreasing order of cost, i.e. finding the K best assignments $a^1, \ldots, a^K$ satisfying

1. $c\left(a^i\right) \leq c\left(a^{i+1}\right), \ i = 1, \ldots, K-1$
2. $c\left(a^K\right) \leq c\left(a\right), \ \forall a \in \mathcal{A} \setminus \left\{a^1, \ldots, a^K\right\}$

where $c(a)$ denotes the cost of assignment $a = \{(1, j_1), \ldots, (n, j_n)\}$, and $\mathcal{A}$ denotes the set of all assignments.

In this paper we use the branching technique described in [13] where the set $\mathcal{A}$ is partitioned into smaller subsets as follows. Given an optimal assignment $a^1 = \{(1, j_1), \ldots, (n, j_n)\}$, the set $\mathcal{A} \setminus \{a^1\}$ is partitioned into $n-1$ disjoint subsets $\mathcal{A}^i$, $i = 1, \ldots, n-1$ where

$$\begin{aligned} &\mathcal{A}^1 = \{a \in \mathcal{A} \ : \ (1, j_1) \notin a\} \ \text{ and} \\ &\mathcal{A}^i = \{a \in \mathcal{A} \ : \ \{(1, j_1), \ldots, (i-1, j_{i-1})\} \in a, (i, j_i) \notin a\}, \ i = 2, \ldots, n-1 \,. \end{aligned}$$

We say that $\{(1, j_1), \ldots, (i-1, j_{i-1})\}$ is *forced* to be in all assignments belonging to $\mathcal{A}^i$. Clearly, the second best assignment a^2 can be found by establishing the optimal assignment in the sets $\mathcal{A}^i$, $i = 1, \ldots, n-1$. Moreover, the branching technique can be applied recursively to subsets $\mathcal{A}^i \subset \mathcal{A}$.

The pseudo-code for the ranking algorithm, named `K-AP`, is shown in Figure 2. The algorithm implicitly maintains a candidate set Φ of pairs $(\bar{a}, \bar{\mathcal{A}})$ where $\bar{a}$ is the optimal assignment in (sub)set $\bar{\mathcal{A}}$. Assuming that the first $k-1$ assignments $a^1, \ldots, a^{k-1}$ have been found, the current candidate set represents the partition of $\mathcal{A} \setminus \{a^1, \ldots, a^{k-1}\}$. Assignment a^k is then found by selecting and removing the pair $(\hat{a}, \widehat{\mathcal{A}})$ containing the assignment with minimum cost in the candidate set (lines 5–7). Next, the branching technique is used to partition $\widehat{\mathcal{A}}$, possibly obtaining new pairs that are added to the candidate set (lines 8–11). In general, it is not necessary to consider all subsets in the partitioning of $\widehat{\mathcal{A}}$. Consider the case where (i, j_i) was forced to be in any assignment belonging to $\widehat{\mathcal{A}}$ in some previous partition. Therefore, $\widehat{\mathcal{A}}^i = \emptyset$ since (i, j_i) is not allowed in any assignment of $\widehat{\mathcal{A}}^i$, and it may be assumed that $\widehat{\mathcal{A}}^i$ is not generated by the algorithm.

```
procedure K-AP()
    $a :=$ SuccSP();
    $\Phi := \{(a, \mathcal{A})\}$;
    for ($k := 1$ to $K$) do
        $(\hat{a}, \widehat{\mathcal{A}}) := \arg\min\{c(\bar{a}) : (\bar{a}, \bar{\mathcal{A}}) \in \Phi\}$;
        if ($(\hat{a}, \widehat{\mathcal{A}}) =$ null) then STOP; else OUTPUT $a^k := \hat{a}$;
        $\Phi := \Phi \setminus \{(\hat{a}, \widehat{\mathcal{A}})\}$;
        for ($i := 1$ to $n - 1$) do
            $\hat{a}^* :=$ FindOptimal($\widehat{\mathcal{A}}^i$);
            if ($c(\hat{a}^*) < \infty$) then $\Phi := \Phi \cup \{(\hat{a}^*, \widehat{\mathcal{A}}^i)\}$;
        end for
    end for
end procedure
```

Figure 2: The ranking assignments algorithm.

Function `FindOptimal` represents the solution method applied to find the optimal assignment in a given subset. Consider partition $\widehat{\mathcal{A}}$ with optimal assignment $\hat{a} = \{(1, j_1), \ldots, (n, j_n)\}$ and assume that an assignment in $\widehat{\mathcal{A}}$ cannot contain $(l_1, t_1), \ldots, (l_q, t_q)$ due to previous partitions. Recall that $(1, j_1), \ldots, (i-1, j_{i-1})$ are forced to be in all assignments belonging to subset $\widehat{\mathcal{A}}^i$. Therefore, assuming that $\widehat{\mathcal{A}}^i$ is non-empty, the optimal assignment can be found solving an AP of size $n - (i-1)$ where

1. Rows $\{1, \ldots, i-1\}$ and columns $\{j_1, \ldots, j_{i-1}\}$ have been removed from the reduced cost matrix $[\bar{c}_{ij}]$, i.e. these indices are not considered in (1) and (2).
2. The reduced cost in cells (i, j_i) and $(l_1, t_1), \ldots, (l_q, t_q)$ is set to infinity.

Given a non-empty subset $\widehat{\mathcal{A}}^i$, let $AP(\widehat{\mathcal{A}}^i)$ denote the AP defined as above. If the successive shortest path procedure, `SuccSP`, is used as the solution method to find the optimal assignment to $AP(\widehat{\mathcal{A}}^i)$, $i = 1, \ldots, n-1$, the overall complexity of `K-AP` is $\mathcal{O}(Kn^4)$. However, the optimal assignment to $AP(\widehat{\mathcal{A}}^i)$ can be found using reoptimization, thereby reducing the complexity of the algorithm.

Let $\hat{a}$ denote the optimal assignment in subset $\widehat{\mathcal{A}}$ found by solving $AP(\widehat{\mathcal{A}})$, let $(\hat{u}, \hat{v})$ denote the corresponding dual values and let the partial assignment $a(i)$ be defined by removing from $\hat{a}$ the single assignment $\{(i, j_i)\}$,. Hence,

$$a(i) := \hat{a} \setminus \{(i, j_i)\} \; . \tag{5}$$

The following lemma is well known, Ahuja et al. [1].

Lemma 1. *$a(i)$ is a partial assignment of size $n-1$, and $(\hat{u}, \hat{v})$ remains dual feasible to $AP(\widehat{\mathcal{A}}^i)$ and satisfies the complementary slackness optimality conditions* (3)*.*

$$\begin{array}{c|cccc}
 & j_i & j_{i+1} & \cdots & j_n \\
\hline
i & \infty & c_{ij_{i+1}} - u_i - \hat{v}_{j_{i+1}} & \cdots & c_{ij_n} - u_i - \hat{v}_{j_n} \\
i+1 & c_{i+1j_i} - \hat{u}_{i+1} - v_{j_i} & \hat{c}_{i+1j_{i+1}} & \cdots & \hat{c}_{i+1j_n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
n & c_{nj_i} - \hat{u}_n - v_{j_i} & \hat{c}_{nj_{i+1}} & \cdots & \hat{c}_{nj_n}
\end{array}$$

Figure 3: The reduced cost matrix to $AP(\widehat{\mathcal{A}}^i)$.

However, by updating the dual variables according to the following scheme

$$\begin{aligned}
u_i &= \hat{u}_i + \min_{j\in V\setminus\{j_1,\ldots,j_i\}} \{c_{ij} - \hat{u}_i - \hat{v}_j\} \\
u_r &= \hat{u}_r,\ r \in \{i+1,\ldots,n\} \\
v_{j_i} &= \hat{v}_{j_i} + \min_{r\in\{i+1,\ldots,n\}} \{c_{rj_i} - \hat{u}_r - \hat{v}_{j_i}\} \\
v_j &= \hat{v}_j,\ j \in V \setminus \{j_1,\ldots,j_i\}\,,
\end{aligned} \tag{6}$$

the following revised version of Lemma 1 can be derived. As we will see in Section 4, this provides a speed-up of the overall algorithm due to the present implementation of the method to find a shortest augmenting path.

Lemma 2. *$a(i)$ is a partial assignment of size $n-1$ and (u,v) defined in (6) is a dual feasible solution to $AP(\widehat{\mathcal{A}}^i)$, satisfying the complementary slackness optimality conditions (3).*

Proof. Given $(\hat{u},\hat{v})$, let $\hat{c}$ denote the corresponding non-negative reduced costs. The reduced cost matrix to $AP(\widehat{\mathcal{A}}^i)$ using (u,v) from (6) can be seen in Figure 3 where it is utilized that $u_r = \hat{u}_r$, $r \in \{i+1,\ldots,n\}$, and $v_j = \hat{v}_j$, $j \in V \setminus \{j_1,\ldots,j_i\}$.

Only the reduced costs in column j_i and row i have changed. Due to (6), the reduced costs in row i satisfy

$$c_{ij} - u_i - v_j \geq c_{ij} - (\hat{u}_i + (c_{ij} - \hat{u}_i - \hat{v}_j)) - \hat{v}_j = 0,\ \ \forall j \in V \setminus \{j_1,\ldots,j_i\}\ .$$

Similar results hold for the reduced costs in column j_i. Hence, (u,v) is a dual feasible solution to $AP(\widehat{\mathcal{A}}^i)$. Moreover, since the reduced costs corresponding to the elements in assignment $a(i)$ have not changed, the complementary slackness optimality conditions (3) still hold. □

A pseudo-code for the reoptimization algorithm is given in Figure 4. Here, the partial assignment $a(i)$ and the dual values (u,v) are calculated first due to (5) and (6), respectively. As an alternative to updating the dual variables, one could substitute the dual

```
1 procedure FindOptimal($\widehat{\mathcal{A}}^i$)
2     $a(i)$ := CreatePartial($\hat{a}$);
3     $(u, v)$ := ModifyDual($\hat{u}, \hat{v}$);
4     $\bar{G}(a(i))$ := BuildResNetwork($a(i), u, v$);
5     $P$ := FindAugmentPath($\bar{G}(a(i))$);
6     $a$ := AugmentSolution($P$);
7 end procedure
```

Figure 4: Finding the optimal solution for a subset.

variables by $(\hat{u}, \hat{v})$ in line 3 of Figure 4. Next, the residual network corresponding to $a(i)$ and the dual solution is built. Finally, the shortest augmenting path and the corresponding solution are found. Due to Theorem 1 and the fact that the length of the partial assignment $a(i)$ is $n - 1$, the following result holds true.

Theorem 2. *Using partial assignment $a(i)$ and dual values (u, v) (or $(\hat{u}, \hat{v})$), the optimal assignment in subset $\widehat{\mathcal{A}}^i$ can be found by solving a single shortest path problem.*

Using Dijkstra's method to find the shortest path, function `FindOptimal` runs in $\mathcal{O}(n^2)$. Therefore, the following time complexity of `K-AP` is obtained, which is equal to the best known time complexity for ranking the K best assignments.

Theorem 3. *The K best assignments using procedure `K-AP` can be found in $\mathcal{O}(Kn^3)$ time.*

4 Computational experiments

In this section, computational experiments for different versions of the algorithm presented in this paper are given.

All tests were performed on an Intel Xeon 2.67 GHz computer with 6 GB RAM using a Red Hat Enterprize Linux operating system version 4.0. In order to minimize the impact of other processes, interrupts etc., we ran the benchmarks on a system with as little workload as possible. Since the results may still vary slightly, we repeated each run on a test instance five times, removed the fastest and slowest CPU time and then used the average of the remaining.

In all implementations of our algorithm the candidate set Φ of pairs $(\bar{a}, \bar{\mathcal{A}})$ is maintained implicitly using a binary tree as described in Nielsen [14], p137. In each node of the branching tree, no information on the solution is stored apart from the solution value. Therefore, a given solution must be recalculated before branching on this solution can be performed. On the downside, this results in an increased running time of the algorithm. However, on the positive side, the memory requirements are much smaller.

The successive shortest path procedure implementation of Jonker and Volgenant [9] and related sub-procedures are used in a slightly modified version allowing problems of varying size to be solved.

The algorithms were all implemented in C++ and compiled with the GNU C++ compiler version 3.4.5 using optimize option `-O3`.

4.1 Test instances

To yield consistency with the literature, the algorithms were tested on two separate classes of test instances known from the literature where only for the first class, different test instances are considered for a given problem size. Therefore, the main part of the results presented in this section are on the first class of test instances.

The first class of test instances are the ones used by Pascoal et al. [16] plus a few larger instances not reported on in that paper. The test instances consist of assignment problems on complete bipartite networks of size $n \in \{50, 100, \dots, 300\}$. Costs c_{ij} are drawn uniformly at random in $\{0, \dots, 99\}$, which is a much smaller cost range than previously stated in [16].[2] For each problem size, ten instances generated with different seeds are available. As a consequence average results will be reported.

The second class of test instances were taken from the OR library[3] and were first used in Beasley [2]. The instances are complete bipartite networks of size $n \in \{100, 200, \dots, 800\}$ and costs c_{ij} drawn uniformly at random in $\{1, \dots, 100\}$. Only one instance of each problem size is available, i.e. 8 instances in total are considered.

4.2 Some comparative statistics

Denote by $\delta_{1,k}$ the relative percentage increase in cost from the optimal solution a^1 to the kth best solution a^k, i.e.

$$\delta_{1,k} = \frac{c(a^k) - c(a^1)}{c(a^1)} \cdot 100 \ ,$$

provided that $c(a^1) \neq 0$ (otherwise $\delta_{1,k}$ is not defined).

Also, let $a_{max} \in \mathcal{A}$ be the worst assignment in terms of cost, i.e. $a_{max} := \arg\max\{c(a) : a \in \mathcal{A}\}$. Notice that a_{max} can be found by computing the best assignment in a modified AP in which all cost entries, c_{ij}, are substituted by $\tilde{c}_{ij} = c_{max} - c_{ij}$ where c_{max} is the maximal cost entry for the original AP.

Table 1 provides some statistics for the first class of test instances. For each problem size, we display the average of the optimal solution values, the average of the relative percentage increase in cost for $K = 500$ and $K = 1000$, and the average worst solution value. The relative percentage increase in cost tends to decrease with the dimension, so an increasing number of the $n!$ feasible solutions becomes alternative optima.

It is worth noticing that $c(a^1)$ decreases when n increases. This suggests that it becomes easier to find an assignment with a small value as n increases. This is not surprising since the number of possible assignments grows much faster with n than the number of possible values of $c(a^1)$.

[2]The correction is due to Pascoal [15].

[3]`http://people.brunel.ac.uk/~mastjjb/jeb/info.html` (see also [3]).

Size	$c(a^1)$	$\delta_{1,500}$	$\delta_{1,1000}$	$c(a_{max})$
50	125.2	4.23	5.06	4811.0
100	112.7	1.62	1.70	9787.5
150	87.5	0.94	1.29	14759.2
200	72.0	0.00	0.28	19726.3
250	52.8	0.19	0.19	24697.8
300	37.4	0.00	0.00	29661.7

Table 1: Statistics (class one).

Since the cost of the worst assignment is much higher than the optimal solution value, the importance of choosing an optimal or near optimal solution, by ranking, is justified.

Analyzing the second class of test instances again shows the number of alternative optima to be high. In fact, only for the instance of size 100, $c(a^1) \neq c(a^{1000})$ indicating that the number of alternative optima is larger compared to class one.

4.3 Results - Dual updating

Two versions of the ranking assignment algorithm `K-AP` were implemented, both using the reoptimization solution method given in Figure 4. In the `DU1` procedure `ModifyDual` updates the dual variables according to (6), whereas, in `NoDU`, the optimal dual variables $(\hat{u}, \hat{v})$ of $\widehat{\mathcal{A}}^i$ are used. That is, nothing else than the computation of the dual variables differs in the two versions. In both versions `FindAugmentPath` finds the shortest augmenting path using the modified implementation of Jonker and Volgenant [9] and an interval heap is used to maintain the candidate set. Further details about the candidate set are provided in Section 4.4.

Size	DU1	NoDU	Ratio
50	0.39	0.42	1.08
100	5.04	5.52	1.09
150	18.55	21.28	1.15
200	41.66	48.58	1.17
250	63.44	75.61	1.19
300	77.78	91.57	1.18

Table 2: Average CPU times and ratios (class one, $K = 1000$).

The effect of updating the dual variables according to (6) is displayed numerically in Table 2 giving results for ranking the $K = 1000$ best assignments in each of the class one test instances. It is evident that updating the dual variables improves the algorithm. This is a result of the present implementation of procedure `FindAugmentPath` using the specialized Dijkstra's method in [9]. Updating the dual variables means decreasing the reduced costs corresponding to the unassigned column j_i. Therefore, this column enters the list of indices to be scanned next faster than if the reduced costs corresponding to column j_i had not been decreased. Since all reduced costs are non-negative, this leads to a faster termination of procedure `FindAugmentPath`.

The same observations hold for class two test instances. Here the ratio varies between 1.11 and 1.21.

4.4 Results - Candidate set implementations

For each iteration of the ranking algorithm an assignment with minimal cost must be picked from the candidate set (procedure `K-AP` line 5 in Figure 2). A priority queue can be used to efficiently sort the costs and retrieve the pair with minimal $c(\bar{a})$.

Heaps are often used to implement priority queues. A *heap* is a tree where each node contains a *key* (the cost in our case). The heap satisfies the *heap property*, i.e. if node B is a child node of A, then $key(A) \leq key(B)$. This implies that the minimal element, with respect to the key, is always in the root node. A node is inserted into the heap by adding it at the bottom of the tree and then letting it "shift" up until the heap property is satisfied. This operation is based on a *compare relation* $\prec$ where two nodes are swapped if $key(A) \prec key(B)$. In our case the compare relation can be either $<$ or $\leq$. For further details about heaps see Tarjan [21].

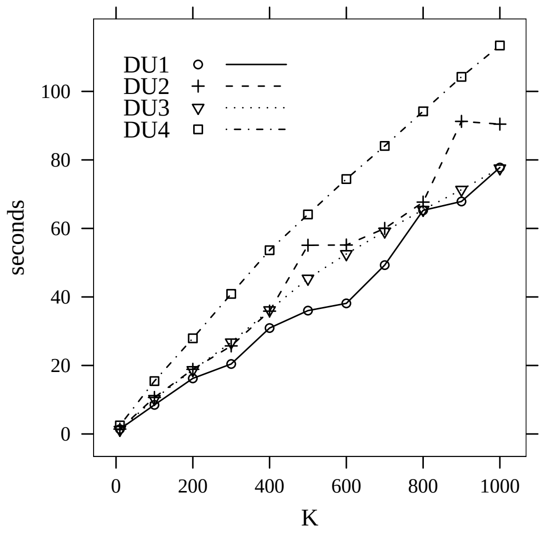
Note that in this study we are only interested in finding the K best assignments. As a result the candidate set only needs to contain at most K pairs resulting in lower memory usage. Therefore, line 10 of procedure `K-AP` in Figure 2 can be modified to

```
if ($c(\hat{a}^*) < \infty$ and $|\Phi| < K$) then $\Phi := \Phi \cup \{(\hat{a}^*, \widehat{\mathcal{A}^i})\}$;
else if ($c(\hat{a}^*) \leq c(a^{max})$) do
    $\Phi := \Phi \cup \{(\hat{a}^*, \widehat{\mathcal{A}^i})\}$;                    (7)
    $\Phi := \Phi \setminus \{(a^{max}, \mathcal{A}^{max})\}$;
end else if
```

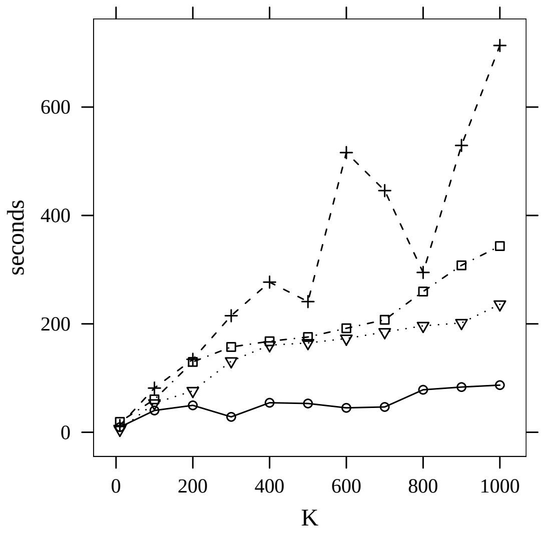
where $(a^{max}, \mathcal{A}^{max})$ is a pair with maximum cost among the pairs in the candidate set.

A priority deque can be used to sort the candidate set for efficiently implementing (7). As opposed to a priority queue a priority deque, holds the possibility to get both the smallest and the largest element using the same data structure. Different algorithms for a priority deque have been implemented and tested by Skov and Olsen [20]. Among these an interval heap (see [23]) was found to be most efficient. This is also the case in this paper. Interval heaps resemble ordinary heaps except that an interval of two keys is maintained in each node of the heap tree instead of a single key. The largest interval (i.e. the minimal and maximal key) is always in the root node. As a result the smallest and the largest element can be found in constant time.

Another important feature which will affect the performance of the ranking algorithm is how we handle pairs in the candidate set with the same cost. As illustrated in Table 1,



(a) Class one $n = 300$.



(b) Class two $n = 800$.

Figure 5: CPU times for different candidate set implementations.

there may exist a huge amount of assignments with the same cost. To improve performance, the goal is to branch on a pair with many rows/columns fixed since fast computations can then be made. Consider two pairs, $(a^i, \mathcal{A}^i)$ and $(a^j, \mathcal{A}^j)$, with equal cost obtained when branching on $\mathcal{A}$. If $j > i$, then $\mathcal{A}^j$ is the most constrained subset and a heuristic rule is always to branch on $\mathcal{A}^j$ before $\mathcal{A}^i$. This is obtained using the $\leq$ compare relation when inserting the keys $c(a^i)$ and $c(a^j)$ into the heap tree. Moreover, it is important to use $c(\hat{a}^*) \leq c(a^{max})$ in (7) (and not $c(\hat{a}^*) < c(a^{max})$), since we then remove the subsets with least rows/columns fixed. The following implementations of the candidate set are considered.

`DU1`: Interval heap using the $\leq$ compare relation.

`DU2`: Interval heap using the $<$ compare relation.

`DU3`: 4-heap using the $\leq$ compare relation.

`DU4`: 4-heap using the $<$ compare relation.

Note that for `DU3` and `DU4` the size of the candidate set is not limited to K since line 10 of procedure `K-AP` in Figure 2 is used instead of (7). For further details about 4-heaps see Tarjan [21].

In Figure 5, the CPU time against K is displayed for both class one and two. First, note that using the $<$ compare relation decreases the performance since we do not branch on the pairs with relatively many rows/columns fixed. Second, if using an interval heap, then for instance $\text{CPU}_{K=800} < \text{CPU}_{K=600}$ if we consider `DU2` in Figure 5(b). This strange behavior is due to the fact that at most K elements are kept in the candidate set. Hence, a pair with many rows/columns fixed may not be added to the candidate set if many pairs with cost equal to $c(a^{max})$ exist. However, for a larger value of K this pair may be added resulting in smaller CPU times when branching on this pair. Indeed this is the case in the second class, which has more alternative assignments with minimum cost. Using the $\leq$ compare relation produces a more stable curve. However, CPU_{K_1} may still be less than CPU_{K_2} ($K_1 > K_2$). Finally, note that `DU1` has the best performance. Hence, this algorithm will be used in the remainder of the paper.

4.5 Results - Other ranking algorithms

In the literature, few other algorithms exist for ranking assignments. The three included below are, to the best of our knowledge, the only available implementations with time complexity $\mathcal{O}(Kn^3)$. Only original codes implemented by the original authors are considered, i.e. we compare algorithm implementations (and therefore also each programmer's skills) and hence the test results must be interpreted with caution.

`VMA1`: An executable version of the algorithm from Pascoal et al. [16] was provided to us by the authors. In `VMA1`, a label correcting algorithm is used for solving the shortest path problems. An internal upper bound on the allowable number of assignment nodes to be stored imposes an implicit limit on the instance size that can be solved by the current implementation of this algorithm.

`VMA2`: An implementation of the algorithm from [16] was provided to us by Przybylski [18].

`CH`: An implementation of the ranking algorithm suggested by Chegireddy and Hamacher [4] was provided to us by Przybylski, Gandibleux, and Ehrgott [19]. Using a binary search tree branching technique [8], the solution method solves shortest path problems on bipartite graphs as a subprocedure.

The algorithm `VMA1` has been implemented in C and compiled using the GNU C++ compiler version 3.3.5 with optimize option `-O4`. Both `VMA2` and `CH` are implemented in C and have been compiled with the GNU C++ compiler version 3.4.5 using optimize option `-O3`.

In Figure 6, the CPU time against K is displayed for algorithms `DU1`, `VMA1`, `VMA2`, and `CH` for the largest problem size ($n = 250$) from the first class of test instances on which all algorithms were capable of ranking the 100 best assignments. The algorithm `VMA1` was unable to rank the 100 best assignments for any of the size $n = 300$ instances because the upper bound limit of number of assignment nodes was reached. The problem sizes not displayed show similar results as the one shown in Figure 6. The two algorithms `CH` and `VMA1` are significantly slower than the remaining two algorithms.

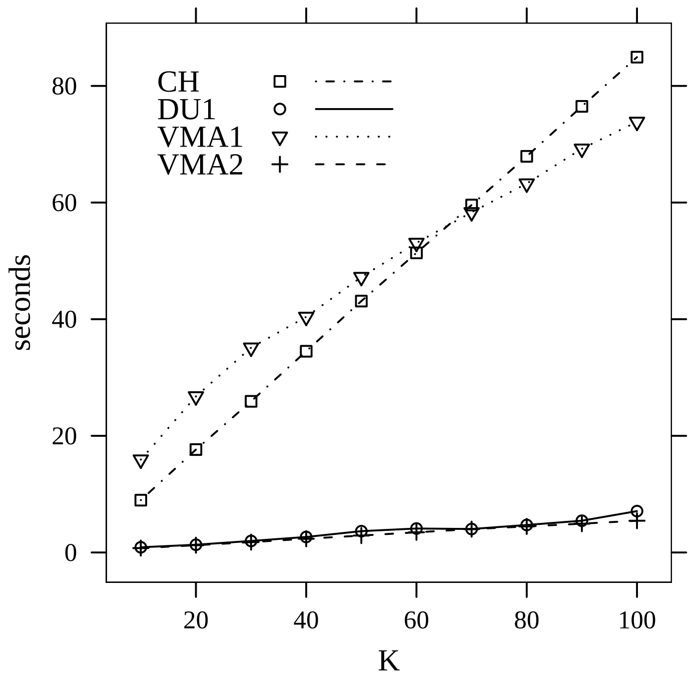


Figure 6: CPU times for different algorithms (class one, $n = 250$).

The reason that `VMA1` is slow may be due to insufficient data structures compared to the data structures used in `VMA2`. However, the exact reason cannot be pointed out since only an executable of `VMA1` was provided to us.

The algorithms `CH` and `VMA2` were both implemented by Przybylski [18]. The results indicate that using a revised version of the branching technique ascribed to Murty [13] outperforms the binary search tree algorithms of Hamacher and Queyranne [8]. This is supported by the numerical results presented in Pascoal et al. [16]. However, to substantiate this will require further analysis and testing.

Since the implementations `VMA2` and `DU1` perform best, the remaining test results cover these two algorithms solely.

The CPU time against K for some representative problem sizes from the first and the second class of instances are shown in Figures 7 and 8, respectively. For small sizes of the problem (≤ 250) neither of the two algorithms are capable of outperforming the other in their present implementation. Remember though that in `DU1` the times to recalculate the optimal assignment for a given subset taken out of the candidate set Φ are included; this is not the case for `VMA2`. For larger problem sizes implementation `DU1` has a better performance, which is mainly due to the candidate set implementation. If all runs on the test instances are considered, then on average the CPU time is 232 per cent higher using `VMA2` instead of `DU1`.

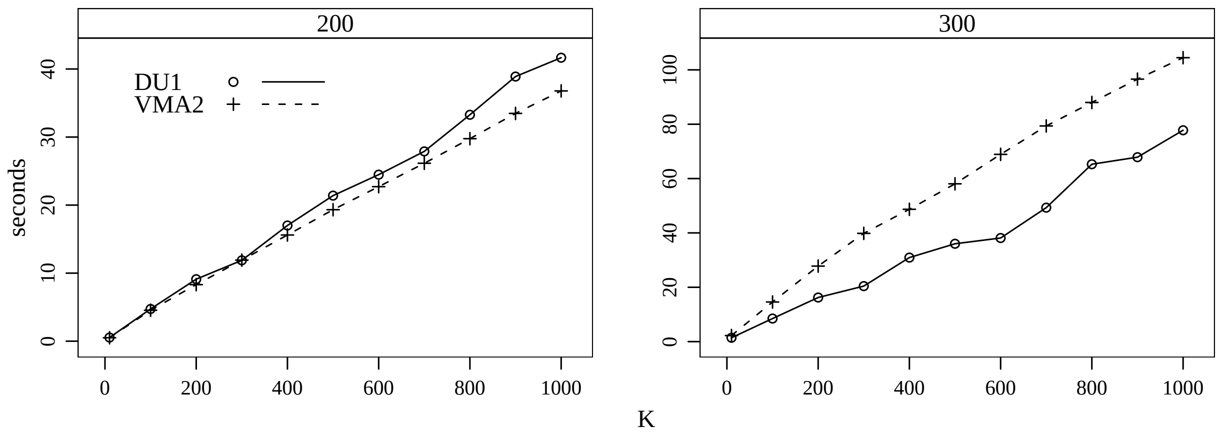


Figure 7: CPU times (class one).

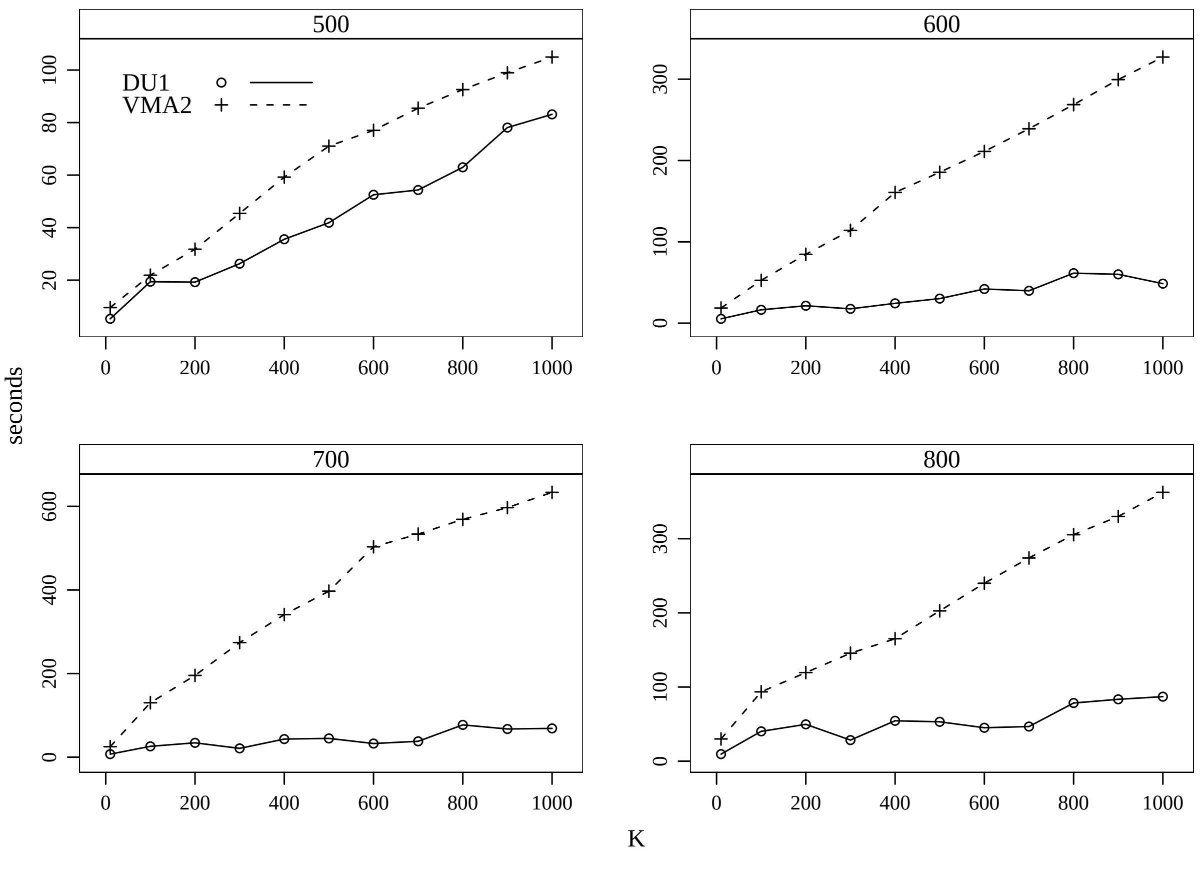
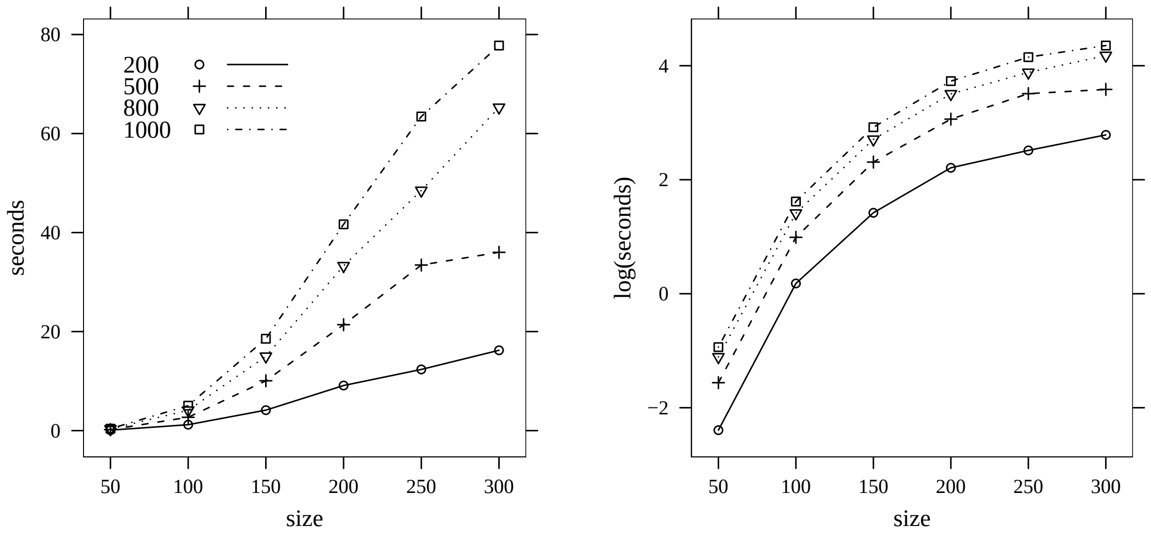


Figure 8: CPU times (class two).



(a) CPU time (b) Logarithm of CPU time

Figure 9: CPU against size for algorithm `DU1` (class one, $K = 200, 500, 800$, and 1000).

Finally, for the first class of test instances, the CPU times for ranking $K = 200, 500, 800$, and 1000 assignments with `DU1` are displayed against problem size in Figure 9(a). The algorithm shows more than a linear growth in CPU time with increasing n. However, it is less than exponential growth, as can be seen in Figure 9(b) displaying logarithms of CPU times.

Acknowledgements

First we thank Jonker and Volgenant for providing efficient source code for solving AP. Next, we would like to thank Pascoal et al. for providing the executable version `VMA1` and Przybylski et al. for the source code to versions `VMA2` and `CH`. Finally, we would like to thank two anonymous referees for improving the paper.

References

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, New Jersey, 1993.

[2] J.E. Beasley. Linear programming on cray supercomputers. *The Journal of the Operational Research Society*, 41(2):133–139, 1990.

[3] J.E. Beasley. OR-library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41(2):1069–1072, 1990.

[4] C.R. Chegireddy and H.W. Hamacher. Algorithms for finding K-best perfect matchings. *Discrete Applied Mathematics*, 18(2):155–165, 1987.

[5] M. Dell'Amico and S. Martello. Linear assignment. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*, pages 355–371. Wiley, Chichester, 1997.

[6] M. Dell'Amico and P. Toth. Algorithms and codes for dense assignment problems: the state of the art. *Discrete Applied Mathematics*, 100(1):17–48, 2000.

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

[8] H.W. Hamacher and M. Queyranne. K best solutions to combinatorial optimization problems. *Annals of Operations Research*, 6(4):123–143, 1985.

[9] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[10] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[11] H.W. Kuhn. Variants of the Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 3:253–258, 1956.

[12] E.L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path. *Management Science*, 18(7):401–405, 1972.

[13] K.G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[14] L.R. Nielsen. *Route Choice in Stochastic Time-Dependent Networks.* PhD thesis, Department of Operations Research, University of Aarhus, 2004.

[15] M.M.B. Pascoal, 2006. Personal communication.

[16] M.M.B. Pascoal, M.E. Captivo, and J.C.N. Clímaco. A note on a new variant of Murty's ranking assignments algorithm. *4OR: Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(3):243–255, 2003.

[17] C.R. Pedersen, L.R. Nielsen, and K.A. Andersen. On the bicriterion multi modal assignment problem. Working Paper No. 2005/3, Department of Operations Research, University of Aarhus, 2005.

[18] A. Przybylski, 2006. Personal communication.

[19] A. Przybylski, X. Gandibleux, and M. Ehrgott. The biobjective assignment problem. Research Report N$^{\underline{o}}$ 05.07, lina – Laboratoire d'Informatique de Nantes Atlantique, 2005.

[20] S. Skov and J.H. Olsen. A comparative analysis of three different priority deques. CPH STL report series 2001-14, Department of Computer Science, University of Copenhagen, 2001.

[21] R.E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* SIAM, Philadelphia, PA, 1983.

[22] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1:173–194, 1971.

[23] J. van Leeuwen. Interval heaps. *The Computer Journal*, 36(3):209–216, 1993.