Mathematical Programming 33 (1985) 243-259
North-Holland

SOLVING MATCHING PROBLEMS WITH LINEAR PROGRAMMING

M. GRÖTSCHEL

Lehstuhl für Angewandte Mathematik II, Universität Augsburg, Memminger Str. 6, 8900 Augsburg, FR Germany

O. HOLLAND

Institut für Ökonometrie und Operations Research, Universität Bonn, Nassestr. 2, 5300 Bonn, FR Germany

Received 30 July 1984
Revised manuscript received 30 April 1985

In this paper we describe an implementation of a cutting plane algorithm for the perfect matching problem which is based on the simplex method. The algorithm has the following features:

- It works on very sparse subgraphs of K_n which are determined heuristically, global optimality is checked using the reduced cost criterion.
- Cutting plane recognition is usually accomplished by heuristics. Only if these fail, the Padberg-Rao procedure is invoked to guarantee finite convergence.

Our computational study shows that—on the average—very few variables and very few cutting planes suffice to find a globally optimal solution. We could solve this way matching problems on complete graphs with up to 1000 nodes. Moreover, it turned out that our cutting plane algorithm is competitive with the fast combinatorial matching algorithms known to date.

Key words: Matching, Linear Programming, Cutting Plane Algorithms, Computational Study.

1. Introduction

The development of the matching algorithm by Edmonds (1965a, 1965b) was one of the milestones in the theory of good algorithms. Edmonds characterized a polyhedron associated with the matchings of a graph. He used this characterization to design a combinatorial method (based on ideas from linear programming) for the solution of the matching problem with polynomial running time. The particular feature of the associated linear program is that it has a number of inequalities which is exponential in the number of edges. This seems prohibitive at first sight, but Edmonds' results show that the 'structure' of an inequality system is what matters and not the number of inequalities. Based on these results a number of further combinatorial algorithms for the matching problem have been developed, see for instance Edmonds (1965a), Lawler (1976), Cunningham and Marsh (1978), Burkard and Derigs (1980).

The current best worst-case bounds for the running time of weighted matching algorithms are $O(|V|^3)$ resp. $O(|V||E|\log|V|)$ for dense resp. sparse graphs $G = [V, E]$, see e.g. Lawler (1976), Derigs (1983), Ball and Derigs (1983).

Using the polyhedral results of Edmonds and the minimum odd cut algorithm of Padberg and Rao (1982) the ellipsoid method can be employed to obtain a quite different polynomial time algorithm for the matching problem, see e.g. Grötschel, Lovász and Schrijver (1981) and Padberg and Rao (1981). This method however appears to be impractical because of the many deficiencies of the ellipsoid method and the rather poor (though polynomial) worst-case bound on the running time.

Due to quite successful implementations of cutting plane algorithms for NP-hard combinatorial optimization problems (see Crowder and Padberg (1980) for the travelling salesman problem, and Grötschel, Jünger and Reinelt (1984) for the linear ordering problem) we were tempted to try this approach also for the well-solved matching problem. The basic idea is to use the theoretical cutting plane framework of the ellipsoid method, but to replace the ellipsoid method by the simplex algorithm. Moreover, the alglrithm resulting this way is enhanced with various heuristic features which empirically show considerable run time improvements.

The outcome of this development is a theoretically nonpolynomial cutting plane algorithm based on the simplex algorithm which—to our surprise—is competitive with the fastest combinatorial matching algorithms known to date (see Section 5 for a discussion of this 'comparableness').

More details about the theoretical background of the algorithm are given in Section 2. Section 3 contains a description of our algorithm and implementational details. Computational results are reported in Section 4.

2. Theoretical background

We denote an *(undirected)graph* by $G=[V, E]$ where V is the *node set* and E the *edge set.* Multiple edges and loops do not play a role in what follows, so we assume that all our graphs are *simple.* The *complete graph* on n nodes is denoted by K_n. If $G=[V, E]$ is a graph and $W \subseteq V$, then $E(W) := \{ij \in E \mid i, j \in W\}$ and $\delta(W) := \{ij \in E \mid i \in W, j \in V \setminus W\}$. For $v \in V$ we write $\delta(v)$ instead of $\delta(\{v\})$. Every edge set $\delta(W)$ is called a *cut.* If $|W|$ is odd then $\delta(W)$ is called *odd cut.*

A *matching* in a graph $G=[V, E]$ is a set of edges $M \subseteq E$ such that no two edges in M have a common endnode. A matching M is called *perfect* if every node of G is contained in an edge of M. If $c: E \to \mathbb{R}$ is a weight function on the edges of G, then, for any subset $M \subseteq E$,

$$c(M) := \sum_{e \in M} c_e$$

denotes the weight of M. The problem of determining a perfect matching in G of minimum weight is called the *(perfect) matching problem.* This problem is trivially equivalent to the problem of finding a maximum weight (possibly nonperfect) matching in G. We restrict our attention here to the case of perfect matchings, and therefore, we will often simply say 'matching' instead of 'perfect matching'.

An *incidence vector* of a subset $M \subseteq E$ is a vector $\chi^M \in \mathbb{R}^E$ satisfying $\chi_e^M = 1$ if $e \in M$ and $\chi_e^M = 0$ if $e \notin M$. The convex hull of the incidence vectors of the perfect matchings of a graph is called the *matching polytope* $P(G)$ of G, i.e.

$$P(G) = \text{conv}\{\chi^M \in \mathbb{R}^E \mid M \subseteq E \text{ perfect matching}\}.$$

The vertices of $P(G)$ correspond in a 1-1 way to the perfect matchings in G. Edmonds (1965b) proved

(2.1) Theorem. *For every graph $G = [V, E]$, $P(G)$ is the solution set of the following system of equations and inequalities*:

$$x(\delta(v)) = 1 \quad \textit{for all } v \in V, \tag{2.2}$$

$$x(E(W)) \leqslant \tfrac{1}{2}(|W| - 1) \quad \textit{for all } W \subseteq V,\ |W| \textit{ odd}, \tag{2.3}$$

$$x_e \geqslant 0 \quad \textit{for all } e \in E. \qquad \square \tag{2.4}$$

It is easy to see that $P(G)$ can also be described by the system of equations and inequalities (2.2), (2.4) and

$$x(\delta(W)) \geqslant 1 \quad \text{for all } W \subseteq V,\ |W| \text{ odd}. \tag{2.5}$$

Thus the matching problem can be solved by solving the linear programming problem

$$\min\ c^T x, \quad x \text{ satisfies (2.2), (2.3), (2.4) (resp. } x \text{ satisfies (2.2), (2.4), (2.5))}. \tag{2.6}$$

It follows from the ellipsoid method (see Grötschel, Lovász and Schrijver (1981), Padberg and Rao (1981)) that the linear program (2.6) can be solved in polynomial time if and only if the following separation problem for $P(G)$ can be solved in polynomial time.

(2.7) Separation problem for $P(G)$. Given a vector $y \in \mathbb{Q}^E$, decide whether $y \in P(G)$, and if y is not in $P(G)$ find a hyperplane separating y from $P(G)$.

Since $P(G)$ is given by the system (2.2), (2.3), (2.4) the separation problem for $P(G)$ can solved by deciding whether a given vector $y \in \mathbb{Q}^E$ satisfies (2.2), (2.3), (2.4) and if this is not the case to find one of these equations resp. inequalities which is violated. Using Edmonds' algorithm for finding a minimum weight perfect matching and the ellipsoid method a polynomial time separation algorithm can be designed, but Padberg and Rao (1982) found a much more effective combinatorial algorithm to solve this problem.

In fact, given $y \in \mathbb{Q}^E$ we can check by substitution whether y satisfies (2.2) and (2.4). If not then we have a desired separating hyperplane (cutting plane). The number of inequalities in (2.3) (resp. in the equivalent system (2.5)) is exponential in $|E|$, so it is prohibitive to check all these inequalities one by one. But the separation

problem for (2.5) can be turned into an optimization problem solvable in polynomial time as follows.

Given $y \in \mathbb{Q}^E$ (we may assume that y satisfies (2.2) and (2.4)), consider the values y_e, $e \in E$ as capacities on the edges of G. Determine an odd cut $\delta(W^*)$ of G of minimum wieght $y(\delta(W^*))$. Clearly, if $y(\delta(W^*)) \geq 1$ then y satisfies all inequalities (2.5), otherwise $y(\delta(W^*)) < 1$ and $x(\delta(W^*)) \geq 1$ gives a cutting plane. Now Padberg and Rao (1982) showed that a variant of the Gomory-Hu procedure to determine a minimum cut in a graph solves this odd cut problem. The running time of the Padberg-Rao algorithm is $(n-1)$ times the complexity of the max-flow algorithm used to solve certain subproblems. So for dense graphs, the running time of the minimum odd cut algorithm is $O(n^4)$.

Incorporating the Padberg-Rao algorithm as separation subroutine in the ellipsoid method yields a polynomial time algorithm for the solution of (2.6), and (since vertex solutions can be found by the ellipsoid method) also for the matching problem.

This method is a theoretical polynomial time algorithm, but we have serious doubts that it will work in practice, in particular for large problems. Still, the scheme seem appealing and we describe in the next section how we have modified it to make it practical.

3. Description of the algorithm

Since the real-world problems we have at hand are defined on complete graphs (and since all the combinatorial algorithms which we knew of and wanted to use for comparisons are designed for complete graphs) we decided to write the code for such graphs. However, this is only a superficial restriction, since we actually work on very sparse subgraphs of K_n which we extract from K_n by certain heuristic rules. It needs almost no changes to modify the code for the treatment of general graphs. So we assume that the input to our algorithm is a list $c_{12}, c_{13}, \ldots, c_{1n}, c_{23}, c_{24}, \ldots, c_{n-1,n}$ of (integral) edge weights of K_n, n even.

We also point out that our code has been developed for solving large problems (e.g. at least 300 nodes which means 44 850 and more variables). The largest problems we report about here have 1000 nodes, i.e. 499 500 variables.

As mentioned before we replace the ellipsoid method by the simplex method and obtain what is usually called a cutting plane procedure. In each iteration a linear program has to be solved to optimality. The optimum solution is analyzed, and either new cutting planes or new variables have to be added. Clearly, the first case comes up if the LP-solution is nonintegral, the second case may occur since we do not work on the whole set of variables. The restriction to a subset of the variables is a computational necessity, since otherwise the linear programs (e.g. with several hundred thousands of variables) would not be manageable. Furthermore, our computational experience has shown that for most problems relatively few variables suffice to produce a globally optimum solution.

In Fig. 3.1 we give a schematic description of our algorithm which shows the general strategy. Tactical issues will be discussed by describing the contents of the boxes of our flow chart.

We now describe the details necessary to understand the boxes of the flow chart given in Fig. 3.1.

Box 1. A set $E' \subseteq E$ of so-called 'candidate edges' to be used as the variables of the initial LP is chosen. For each node we determine the NN $(1 \leq NN \leq |V|-1)$ shortest (with respect to their objective function value) edges incident to it. E' is the union of these edges. The algorithm has been tested for various values of NN. It turns out (see Section 4) that

$$5 \leq NN \leq 10$$

is a good choice. For problems of size 500–1000 this means that the number of variables actually used is less then 1% of the total number of variables.

Box 2. The edges of E' are sorted via Quicksort in nondecreasing order with respect to their weights. Then we find a matching $M \subseteq E'$ in a greedy fashion using the above ordering. If M is not a perfect matching, arbitrary pairs of yet unmatched nodes are connected by taking edges from $E \backslash E'$. To guarantee feasibility of the subproblem defined by E' we add the edges $M \backslash E'$ to E'. The matching M found this way does not only guarantee feasibility of the initial LP, but is also used for optimality checks and set-up purposes.

Box 3. The first linear program to be solved is the trivial relaxation

$$\begin{aligned} \min \quad & c_{E'}^{\mathrm{T}} x_{E'} \\ & x_{E'}(\delta(v)) = 1 \quad \text{for all } v \in V, \\ & x_e \geq 0 \quad \text{for all } e \in E', \end{aligned}$$

of the subproblem induced by E'. This LP has the desirable feature that the set of feasible integral solutions of it is the (nonempty) set of all incidence vectors of perfect matchings in $[V, E']$.

For an initial basic solution of this LP we use the variables corresponding to the matching M determined in Box 2.

Box 4. The linear programs are solved with IBM-MPSX/370 using PRIMAL. In a general step, the optimum solution obtained in the previous step is used as a starting basis. This is computationally profitable since it avoids total reoptimization of the whole LP. If new cuts have been added, this basis is infeasible for the primal problem but feasible for the dual. In this case (the usual one) a dual variant DUAL of the simplex method is used before calling PRIMAL. Let x^* be the optimum solution of the current LP.

Box 5. We construct a capacitated graph G_{x^*} which represents the LP-solution and serves as input for the cutting-plane recognition phase of the algorithm as follows. Set $G_{x^*} = [V, E_{x^*}]$ where $E_{x^*} := \{ij \in E' \mid x_{ij}^* > 0\}$, and define edge capacities $k: E_{x^*} \to \mathbb{R}$ by $k_{ij} := x_{ij}^*$. At the same time, we test whether x^* is integral (to avoid scanning x^* twice).

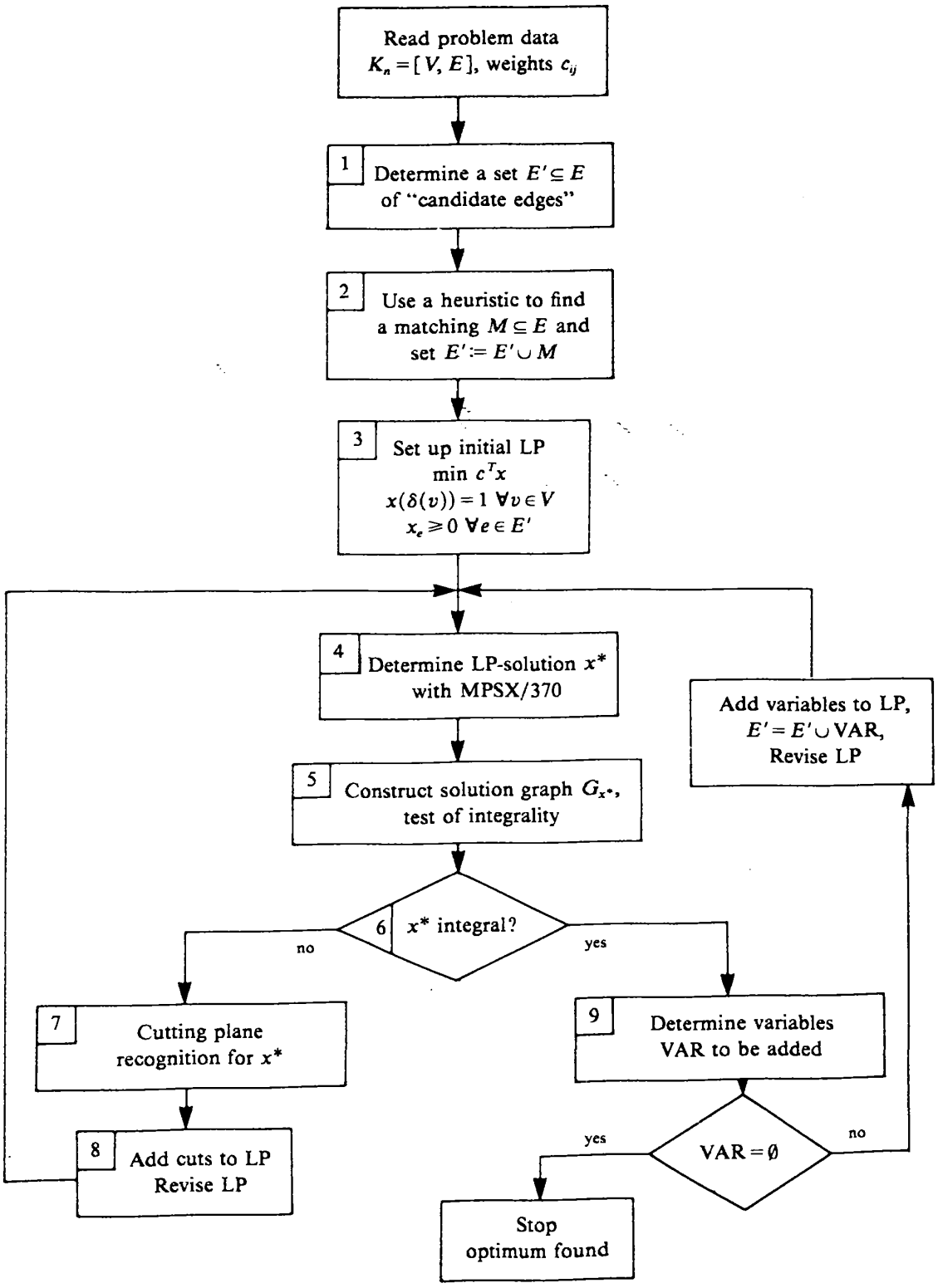
Read problem data
$K_n = [V, E]$, weights c_{ij}
1
Determine a set $E' \subseteq E$ of "candidate edges"
2
Use a heuristic to find a matching $M \subseteq E$ and set $E' := E' \cup M$
3
Set up initial LP
min $c^T x$
$x(\delta(v)) = 1 \ \forall v \in V$
$x_e \geq 0 \ \forall e \in E'$
4
Determine LP-solution x^* with MPSX/370
5
Construct solution graph G_{x^*}, test of integrality
6
x^* integral?
no
yes
7
Cutting plane recognition for x^*
8
Add cuts to LP
Revise LP
9
Determine variables VAR to be added
VAR = ∅
yes
no
Add variables to LP,
$E' = E' \cup$ VAR,
Revise LP
Stop
optimum found

Fig. 3.1.

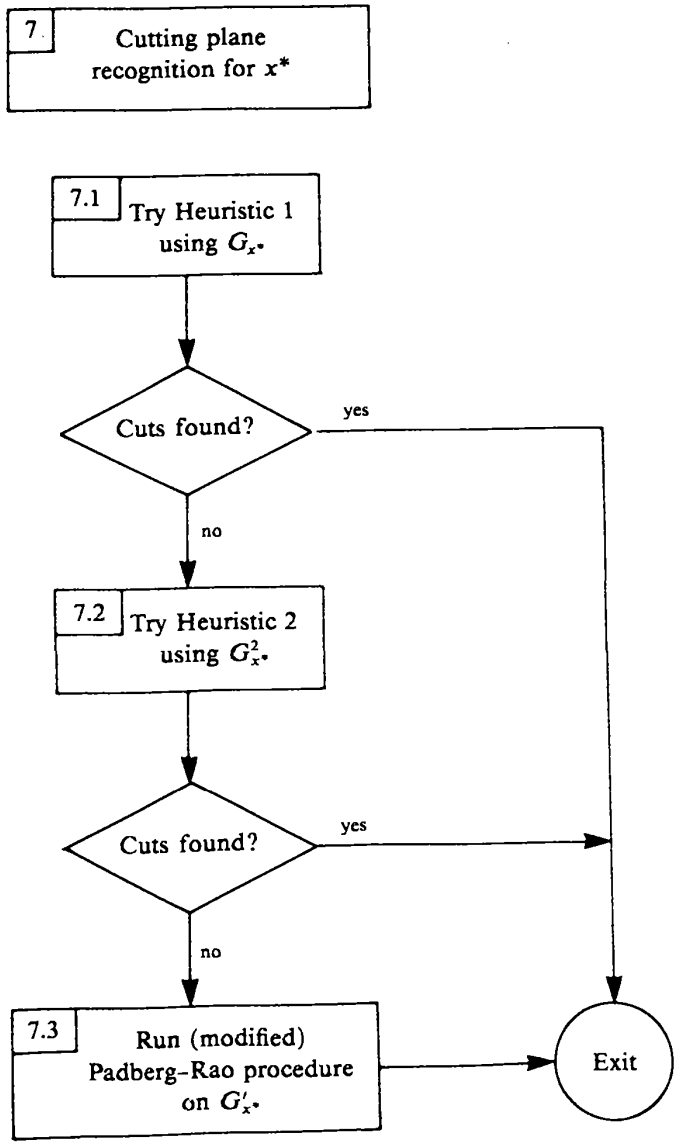
7
Cutting plane recognition for x^*
7.1
Try Heuristic 1 using G_{x^*}
Cuts found?
yes
no
7.2
Try Heuristic 2 using $G^2_{x^*}$
Cuts found?
yes
no
7.3
Run (modified) Padberg-Rao procedure on G'_{x^*}
Exit

Fig. 3.2.

Box 6. If x^* is integral, it induces an optimum perfect matching for the subproblem considered at present. In this case global optimality is checked (Box 9). Otherwise cutting planes chopping off x^* are determined (Box 7).

Boxes 7 and 8. A detailed description of Box 7 (the 'key box') is given later. The algorithms of Box 7 guarantee that at least one inequality of type (2.3) (resp. (2.5)) which is violated by the current fractional solution x^* is found. All cutting planes found in Box 7 are added in Box 8 to the current LP using the MPSX procedure REVISE. Hereafter we return to Box 4.

Box 9. The optimum solution of the current LP is integer and therefore determines a minimum weight perfect matching of $[V, E']$. This may not be an optimum perfect matching of K_n! To check this we calculate for each edge $e \in E \setminus E'$ the reduced cost of the variable corresponding to it. Let VAR denote the set of edges which could lead to an improvement. If all reduced costs have the correct sign, i.e. VAR $= \emptyset$, the current solution also determines an optimum perfect matching of K_n, thus we can stop having found an optimum solution. In VAR $\neq \emptyset$, then we set $E' := E' \cup$ VAR and add the corresponding variables—after generating their columns—to the current LP through REVISE. Afterwards we return to Box 4.

The crucial part of our algorithm is the cutting plane recognition procedure. To outline our strategy in more detail we have expanded Box 7 of Fig. 3.1 into the flow chart shown in Fig. 3.2. We now describe the contents of Box 7 of Fig. 3.1, resp. of the boxes of Fig. 3.2.

Box 7. The optimum solution x^* of the current LP is fractional and therefore not in $P(G)$. We have to find a cutting plane and would like to do this as fast as possible.

First observe that the systems of inequalities (2.3) and (2.5) are theoretically equivalent, but (2.3) has some numerical advantages. Namely, if $W \subseteq V$ is an odd set, then the three inequalities $x(E(W)) \leq \frac{1}{2}(|W|-1)$, $x(E(V \setminus W)) \leq \frac{1}{2}(|V \setminus W|)-1$ and $x(\delta(W)) \geq 1$ determine one and the same face of $P(G)$. So, if x^* violates one of these inequalities it also violates the other two. For numerical reasons we want to keep our LP as sparse as possible, and it is easy to see that (with respect to K_n) always one of the two inequalities of type (2.3) has fewer nonzeros than the equivalent inequality of (2.5). This counting may not apply to our sparse graphs. So one should actually test which of the inequalities has the fewest nonzeros and add this one. However, this causes additional computation and the use of another data structure for the second type of inequalities. Therefore we have evaluated the number of nonzeros in the two types of equivalent cuts empirically, and it turned out that in the vast majority of cases the inequalities of type (2.5) were less sparse than those of type (2.3). Thus we enlarge our LP by adding cutting planes of type (2.3) only.

We have mentioned before that the Padberg–Rao procedure to solve the separation problem for the inequality system (2.5) (and hence for (2.3)) runs in $O(n^4)$ time. This is prohibitive for large problems and therefore we try to avoid the use of this procedure by invoking cutting plane recognition heuristics.

Box 7.1. Using Depth-First-Search we check whether the graph G_{x^*} (defined in Box 5) has odd components (i.e. connected components with an odd number of nodes). If so, each odd component, say $[V_i, E_i]$, of G_{x^*} yields an inequality

$$x(E_i) \leqq \tfrac{1}{2}(|V_i| - 1)$$

which is violated by x^*. We do not continue the search for cutting planes and go to Box 8 adding all cutting planes found. Note that the complexity of this step is $O(|E_{x^*}|)$. Empirically it has turned out that G_{x^*} is quite sparse (not only because of our choice of E'), so the empirical complexity of this step is $O(n)$.

Box 7.2. If no violated inequality of type (2.3) has been found in Box 7.1, we determine a new graph $G^2_{x^*}$ by eliminating all edges from E_{x^*} whose capacity is less than $\varepsilon > 0$. In our code we have chosen $\varepsilon = 0.3$. Then we apply the algorithm of Box 7.1 to $G^2_{x^*}$. (Roughly speaking, we determine 'weak' components of G_{x^*}.) If this algorithm finds odd components of $G^2_{x^*}$ we have to check whether x^* violates the corresponding inequalities (this may not be the case!). If so, cutting planes have been found and we proceed to Box 8 adding all cutting planes found. Again, this heuristic runs in $O(n)$ time empirically.

Our empirical observation is that the two heuristics work surprisingly well. In 50 out of the 58 problems on which we report in Section 4, the cutting planes found by the two heuristics described above were sufficient to determine an optimum perfect matching. Furthermore, the total number of cutting planes added to the 58 problems was 1309, and only 46 of these were not recognized by our heuristics.

Only in case our heuristics fail to produce a cutting plane we call the Padberg–Rao procedure.

Box 7.3. Let V_1 denote the set of nodes of G_{x^*} which are incident to an edge $e \in E_{x^*}$ with capacity one. Remove V_1 and all edges incident to a node in V_1 from G_{x^*}. Denote the new graph by $G'_{x^*} = [V', E'_{x^*}]$. Clearly, each minimum capacity odd cut in G'_{x^*} determines a minimum capacity odd cut in G_{x^*}, and vice versa. (Note also that the number of nodes of G'_{x^*} is even.)

As shown by Padberg and Rao (1982) a minimum capacity odd cut of G'_{x^*} can be found by determining the Gomory–Hu tree of G'_{x^*} and finding an edge of this tree having minimum capacity among those edges whose removal splits the tree into two components of odd size.

Since x^* is not in the matching polytope this procedure must yield at least one odd cut, say $\delta(W)$, of G'_{x^*} of capacity less than one. So $x(E(W)) \leqq \frac{1}{2}(|W| - 1)$ is an inequality of type (2.3) violated by x^* (we add the inequality corresponding to the smaller of the two sets X, $V \setminus W$).

The Gomory–Hu procedure to determine the Gomory–Hu tree uses a max-flow algorithm as subroutine. We have compared a number of max-flow algorithms available to us on flow-problems of the type that come up in our cutting plane algorithm. Among the algorithms we have tested were variants of the Ford–Fulkerson (1957) algorithm, the Dinic (1970) algorithm, the Karzanov (1972) algorithm, the

3-Indians (Malhorta et al. (1978)) algorithm and the primal network flow algorithm (Glover et al. (1979)). The primal algorithm turned out to be the clear winner with respect to empirically observed running times.

To save time spent in this computationally expensive step 7.3 we do not grow the full Gomory-Hu tree unless necessary. As soon as we have found an odd cut of capacity less than one in one of the intermediate steps we stop and go to Box 8.

Altogether in the 58 problems discussed later, 46 cuts were found in this step 7.3, and for this a total of 212 max-flow calculations had to be performed. Determining these 46 cuts by growing the full Gomory-Hu tree approximately 9000 max-flow calculations would have been necessary.

This finishes the description of our algorithm for the perfect matching problem.

4. Performance of the algorithm

The algorithm was run on the IBM 4331 model 2 of the Institut für Ökonometrie und Operations Research of the University of Bonn under VM/BSEP. The program was written in PL/I and ECL standard control language of MPSX. To get comparisons, this program was also executed on an IBM 4361 model LK5 (Institut für Ökonometrie und Operations Research, Universität Bonn), an IBM 370/168, and an IBM 3081 (both of the Rechenzentrum der Universität Bonn). The running times on the 4331 reported here were about 3, 15 resp. 45 times as long as those on the other three machines.

4.1. Test problems

Real-world instances of the matching problem in its pure form are rather rare. Matching problems, however, come up frequently as subproblems of hard problems; and a number of combinatorial problems (resp. certain aspects of these problems) can be formulated as matching problems. For instance, the matching problem plays a role in the solution of certain shortest path and max-cut problems (cf. Grötschel and Pulleyblank (1981)), and the Christofides-heuristic for the travelling salesman problem needs a perfect matching algorithm for complete graphs as a subroutine. We are using our code and modifications of it in several ways to solve travelling salesman problems (e.g. in the Christofides heuristic), therefore we have chosen seven real-world TSP-problems with

$$24, 42, 48, 96, 120, 202, 666$$

nodes available to us to demonstrate the performance of our code for this type of 'real-world problems'.

Furthermore, we report here about 10 random problems with the following numbers of nodes:

$$100, 200, \ldots, 1000.$$

The integer objective function values for these problems were distributed uniformly in the interval 1 to 5000. Each problem was run with different values for NN, see Box 1 in Section 3. All problems were run on NN = 5, 10, some problems also on NN = 2, 15 and $n-1$.

4.2. Computational results

Table 4.1 shows the running times in CPU-minutes (counting input, output, set-up etc.) on the IBM 4331/2 depending on the number of next neighbour variables NN, see Section 3, Box 1.

For the seven real-world problems, the shortest running times were obtained for NN = 5 (three times) or NN = 10 (four times). For the random problems, NN = 5 always was the best choice with one exception ($n = 200$). On the average (also counting further problems not reported about here) the code performed superior with NN = 5.

It turned out that, for all problems, the edges used in the optimum matching found were among the edges given by the ten next neighbours of any node, cf. SP in Tables 4.2, 4.3. Therefore, it is clear that increasing NN to a number larger than 10 resulted in poorer running times.

The first experiments with NN = 2 showed that the gap between the optimum solution of the first subproblem solved and the global optimum solution was so large that it caused nearly all variables to enter the next subproblem (cf. Section 3,

Table 4.1
Running times (depending on NN)

n \ NN	2	5	10	15	$n-1$*
24	—	0:14	0:15	0:16	0:18
42	—	0:40	0:34	0:42	1:09
48	—	0:33	0:30	0:55	1:03
96	—	1:07	1:09	1:27	5:34
120	—	6:18	5:38	7:19	34:50
202	—	3:24	3:29	4:32	7:52
666	—	73:01	62:31	70:51	108:09
100	0:59	0:40	0:48	—	—
200	3:46	1:45	1:28	—	—
300	5:43	3:45	3:56	—	—
400	10:51	6:06	7:26	—	—
500	17:27	9:19	13:04	—	—
600	21:17	14:15	18:28	—	—
700	31:08	23:42	35:18	—	—
800	56:30	33:55	47:46	—	—
900	77:37	50:22	63:35	—	—
1000	113:46	61:09	142:51	—	—

* For $n = 202$ we used NN = 30, for $n = 666$ we used NN = 25.

Table 4.2

n	NN	VAR	LP	C	HC	MF	SP
24	5	80	1	0	0	0	1
	10	150	1	0	0	0	1
	15	217	1	0	0	0	1
	23	276	1	0	0	0	1
42	5	135	8	12	12	0	2
	10	259	6	10	10	0	1
	15	389	6	10	10	0	1
	41	861	4	10	10	0	1
48	5	153	4	4	4	0	2
	10	290	3	4	4	0	1
	15	433	3	4	4	0	1
	47	1128	3	4	4	0	1
96	5	302	8	16	16	0	2
	10	712	7	16	16	0	1
	15	1063	7	16	16	0	1
	95	4560	7	16	16	0	1
120	5	379	39	63	44	77	2
	10	712	29	54	45	46	1
	15	1063	29	54	45	46	1
	119	7140	24	50	46	20	1
202	5	639	12	36	36	0	2
	10	1262	10	34	34	0	1
	15	1891	10	34	34	0	1
	30	3928	10	34	34	0	1
666	5	2101	27	162	162	0	3
	10	4055	37	184	184	0	1
	15	6083	37	184	184	0	1
	25	10 201	37	184	184	0	1

Box 9). Therefore, in this case we restricted the size of VAR to $n+500$ to keep the problem manageable. This resulted in a large number of subproblems to be solved (cf. SP in Tables 4.2, 4.3) but also to unexpectedly large total numbers of variables used, and the running times were considerably larger than those for NN = 5. In particular, the time consuming Box 9 was entered more often than in all other cases. There may be more intelligent ways to select new variables VAR (e.g. choosing a limited number of new variables taking the values of the reduced costs into account), but we believe that NN = 2 is just a bad initial choice. Our computational experience shows that in the problem range we have considered ($24 \leq n \leq 1000$) a good choice for the initial set E' of variables is obtained by setting $5 \leq \mathrm{NN} \leq 10$.

In Tables 4.2 and 4.3 we report results which—in our opinion—are at least as significant as the running times. The rows of these tables correspond to the test problems (described before), the column labels have the following meaning:

Table 4.3

n	NN	VAR	LP	C	HC	MF	SP
100	2	203	3	2	2	0	2
	5	314	2	2	2	0	1
	10	574	2	2	2	0	1
200	2	1011	4	5	4	3	4
	5	631	4	5	4	3	1
	10	1128	4	5	4	3	1
300	2	1285	5	2	2	0	4
	5	958	2	0	0	0	2
	10	1772	1	0	0	0	1
400	2	1735	6	4	4	0	6
	5	1279	2	0	0	0	2
	10	2378	1	0	0	0	1
500	2	1916	11	12	12	0	5
	5	1568	3	2	2	0	2
	10	2942	2	2	2	0	1
600	2	2406	9	8	8	0	5
	5	1903	4	6	6	0	1
	10	3563	4	6	6	0	1
700	2	2781	7	6	6	0	4
	5	2237	2	0	0	0	2
	10	4121	1	0	0	0	1
800	2	3144	17	22	18	14	5
	5	2554	5	6	6	0	2
	10	4711	4	6	6	0	1
900	2	3238	8	4	4	0	6
	5	2856	3	2	2	0	2
	10	5295	2	2	2	0	1
1000	2	4602	9	4	4	0	7
	5	3174	4	4	4	0	2
	10	5876	2	2	2	0	1

NN ≙ number of next neighbour edges chosen (see Box 1, Section 3),
VAR ≙ total number of variables (initial ones plus all the ones added later) used in the linear programming phase,
LP ≙ total number of linear programs solved (calls of MPSX),
C ≙ total number of cuts added to the initial linear program of Box 3,
HC ≙ number of cuts found by the heuristics described in Boxes 7.1 and 7.2,
MF ≙ total number of max-flow calculations (in the Padberg–Rao algorithm),
SP ≙ number of LP-subproblems solved (number of times Box 9 is entered).

Tables 4.2 and 4.3 show that, for instance, for NN = 5 in all cases less than 1% of all variables were sufficient to find an optimum integral LP-solution and to prove global optimality. The total number of linear programs solved (LP) is extremely

small, in particular for all random problems and all large problems ($n \geq 200$). The total number of cuts (C) added is at most $n/2$, and moreover, almost all cuts were found by the heuristics (HC). Only in 8 of the 58 problems documented in Tables 4.2 and 4.3 the Padberg-Rao procedure to identify violated constraints had to be invoked. But also in these 8 problems very few calls of the max-flow algorithm (MF) were sufficient to determine a violated cut. The number of subproblems solved (SP), i.e. calls of Box 9, with the costly calculation of reduced costs is very small. For NN = 5 no more than 2 (exception $n = 666$) subproblems were generated. For NN = 10 the inital set of variables always contained the optimum perfect matching, so SP = 1 in all cases with NN $\geq$ 10.

Summing up our computational experience with the Simplex-based cutting plane algorithm described before we can state the following empirical observations.

For the problems considered the running times grew roughly linearly with the number of edges. This result has been surprising to us since we implemented an algorithm with nonpolynomial worst-case time bound. The reasons for this seem to be the following

- less than 1% of the total variables were needed to provide an optimum solution,
- the number of subproblems (optimality checks) has been very small,
- the total number of cuts needed has been less than $|V|/2$,
- nearly all cuts were identified by fast heuristics,
- the modification of the Padberg-Rao procedure used worked quite effectively.
- And the simplex algorithm is fast on the average!

The performance of the LP-code used (IBM's MPSX) was, however, not totally satisfactory. For instance, we observed that for each of the two linear programs which had to be solved for the 1000 nodes problem with NN = 5 took about 23 minutes. About 90% of this time was spent on I/O-activities. Thus the numerical calculations were performed in about 2.5 minutes. The reason for this behaviour seems to lie in the design of MPSX which has been developed in the late sixties to run on machines with considerably less memory than available today. So it might be possible to get a speed-up by a factor 10 in the pure LP phase. Moreover, the LP-update facilities (MPSX-procedure REVISE) for adding rows and variables are quite time consuming due to their design for occasional use. Again, substantial speed-up may be obtained in these phases of the algorithm.

In our opinion there is need for new and better LP-codes which are adequate to the requirements of a cutting plane procedure, which also have an interface to a language 'faster' than PL/I, and which take more advantage of the storage capacities available today. With such an LP-code our algorithm would probably show an even better computational behaviour.

5. Comparisons with other codes and conclusions

As mentioned in Section 1 there are several combinatorial matching-codes available. They have been compared by U. Derigs with a code published in Burkard and Derigs

(1980). This code turned out to be superior to all other combinatorial codes used in this study. Thus we could restrict ourselves to a comparison with the Burkard & Derigs procedure, called SMP, which is implemented in FORTRAN IV, see also Derigs (1983).

Table 5.1 reports the results we obtained. T(SMP) is the total CPU-time in minutes (on the same IBM 4331/2) needed with the combinatorial code SMP. The right column ($T(5)$) contains the running times for the cutting plane procedure with NN = 5 (cf. Chapter 3, Box 1, and Tables 4.2 and 4.3). The times include total computation, input, output, etc.

This table, in fact, was the real surprise of our computational study. It shows that the performance of our cutting plane procedure is comparable with the best combinatorial code for the perfect matching problem available today.

Clearly, for small problems the overhead costs (set-up etc.) of the LP-code are significant, and usually a combinatorial code has terminated before the first LP has been started. But for $n \geq 300$ our code is really competitive and sometimes faster. (Our initial guess was that it would be about 10 to 50 times slower.) We should however be careful with such a conclusion since it may not only depend on the codes, it may also depend on the computers used (due to special processors favouring the operations of one of the codes).

We are sure that the combinatorial codes have not reached their limits yet. One reason is certainly that most of these codes were designed and programmed by people who only occasionally implement algorithms. (In contrast to this, most of the commercial LP-codes like MPSX have been implemented by professionals, and our algorithm certainly benefits from this.) Some possibilities for speed up are the following. All the combinatorial codes we know use the full set of variables all the

Table 5.1
Comparison with the best combinatorial code

n	T(SMP)	$T(5)$
24	0:02	0:14
42	0:03	0:40
48	0:04	0:33
96	0:11	1:07
100	0:10	0:40
120	0:21	6:18
200	1:02	1:45
202	1:52	3:24
300	4:07	3:45
400	5:04	6:06
500	10:31	9:19
600	24:43	14:15
666	56:09	73:01
700	51:04	23:42
800	69:47	33:55
900	59:05	50:22
1000	66:35	61:09

time. We believe that reducing the number of variables heuristically, performing optimality checks by calculating reduced costs, and applying sparse graph techniques could considerably improve the empirical (but not the theoretical) running times. Moreover, only few of these combinatorial codes have heuristic features (e.g. intermediate search for a good matching, or using a good matching as a starting solution) which usually result in a better performance on the average. Implementing these ideas might give the lead again to the combinatorial codes, but these gains might be matched by our algorithm through the use of better LP-codes as indicated in Section 4.

Overall, it is our opinion that LP-codes provide more flexibility, in particular if one wants to solve problems which are not pure matching problems (this is the usual case in practice). Even if the additional constraints destroy integrality of the LP, good bounds (depending on the type of problem) for a (fast) branch & bound procedure may be obtained. Furthermore, cutting-plane procedures can be used as subroutines for difficult (e.g. NP-complete) problems.

Within this conceptual framework we are currently working on a special code for the perfect 2-matching (with applications to the travelling salesman problem) and for the general capacitated b-matching problem, and we have good hopes for a reasonable numerical behaviour of these codes.

There is, however, one drawback of the cutting plane LP-codes which we should mention. They are by far not as easy to code and to make reliable as it might appear from our description.

Note added in proof. After completion of our study U. Derigs, see Derigs (1984), has applied the sparse graph techniques and heuristic enhancements described in the present paper to his primal combinatorial matching codes. This resulted in substantial running time improvements which make his new codes faster than our cutting plane procedure.

References

M. Ball and U. Derigs, "An analysis of alternate strategies for implementing matching algorithms", *Networks* 13 (1983) 517–549.

R.E. Burkard and U. Derigs, *Assignment and Matching Problems : Solution Methods with* FORTRAN-*Programs* (Springer Lecture Notes in Economics and Mathematical Systems, No. 184, Berlin, 1980).

H. Crowder and M.W. Padberg, "Solving large-scale symmetric travelling salesman problems", *Management Science* 26 (1980) 495–509.

W. Cunningham and A. Marsh, "A primal algorithm for optimum matching", *Mathematical Programming Study* 8 (1978) 50–72.

U. Derigs, "Solving matching problems via shortest path techniques", Report No. 83263-OR, Institut für Ökonometrie und Operations Research, Universität Bonn (Bonn, 1983).

U. Derigs, "Solving large scale matching problems efficiently—A new primal matching approach", Report No. 84346-OR, Institut für Ökonometrie und Operations Research, Universität Bonn (Bonn, 1984).

E.A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation", *Soviet Mathematics Doklady* 11 (1970) 1277–1280.

J. Edmonds, "Paths, trees and flowers", *Canadian Journal of Mathematics* 17 (1965) 449–467.

J. Edmonds, "Maximum matching and a polyhedron with 0, 1 vertices", *Journal of Research National Bureau of Standards* 69B (1965) 125–130.

L.R. Ford and D.R. Fulkerson, "A simple algorithm for finding maximal flows and an application to the Hitchcock problem", *Canadian Journal of Mathematics* 9 (1957) 210–218.

F. Glover, D. Klingman, J. Mote and D. Whitman, "A primal simplex variant for the maximum flow problem", Center for Cybernetic Studies, CCS 362 (Austin, TX, 1979).

M. Grötschel, M. Jünger and G. Reinelt, "A cutting plane algorithm for the linear ordering problem", *Operations Research* 32 (1984) 1195–1220.

M. Grötschel, L. Lovász and A. Schrijver, "The ellipsoid method and its consequences in combinatorial optimization", *Combinatorica* 1 (1981) 169–197.

M. Grötschel and W.R. Pulleyblank, "Weakly bipartite graphs and the max-cut problem", *Operations Research Letters* 1 (1981) 23–27.

A.V. Karzanov, "Determining the maximal flow in a network by the method of preflows", *Soviet Mathematics Doklady* 15 (1972) 434–437.

E.L. Lawler, *Combinatorial optimization: Networks and matroids* (Holt, Rinehart and Winston, New York, 1976).

V.M. Malhorta, M.P. Kumar and S.N. Maheshwari, "An $O(|V|^3)$ algorithm for finding maximum flows in networks", *Information Processing Letters* 7 (1978) 277–278.

M.W. Padberg and M.R. Rao, "Odd minimum cut-sets and b-matchings", *Mathematics of Operations Research* 7 (1982) 67–80.

M.W. Padberg and M.R. Rao, "The Russian method for linear inequalities III: Bounded integer Programming", Preprint, GBA New York University, New York, May 1981.