An Analysis of Alternative Strategies for Implementing Matching Algorithms

Michael O. Ball
College of Business and Management, University of Maryland, College Park, Maryland 20742

Ulrich Derigs
Institut für Operations Research, Universität Bonn, West Germany

In this paper we explore implementation issues related to the solution of the weighted matching problem defined on an undirected graph $G = (V, E)$. We present algorithms based on the two different linear characterizations of the feasible solutions to the matching problem. Furthermore, we present two specialized implementations, one with an $O(|V|^3)$ time bound and one with an $O(|V||E| \log |V|)$ time bound. Both of these implementations have storage requirements that are linear in $|V|$ and $|E|$. We initially develop these algorithms as special implementations of the well-known primal-dual (blossom) algorithm and then show how the updates they perform have an interesting interpretation when the algorithms are viewed as methods that successively find shortest augmenting paths. Finally, we show that postoptimality analysis can be performed very efficiently within this setting.

I. INTRODUCTION

Given an undirected graph $G = (V, E)$, a *matching* $M \subseteq E$ is a subset of edges no two of which are incident with a common vertex. For any $M \subseteq E$, we define $V(M)$ as the set of vertices incident to some edge in M. *A perfect matching* is a matching M with $V(M) = V$. Given a cost c_{ij} for each $(i, j) \in E$, we define a *minimum-cost perfect matching* as a perfect matching M that minimizes $\Sigma_{(i,j)\in M} c_{ij}$. We assume without loss of generality that $c_{ij} \geq 0$ for all $(i, j) \in E$.

We call $G = (V, E, c)$ a *weighted graph*. If, in addition, we are given a cost a_i for each $i \in V$, we define a *minimum-cost matching* as a matching M that minimizes $\Sigma_{(i,j)\in M} c_{ij} + \Sigma_{i\in V-V(M)} a_i$. In this paper we describe algorithms for the minimum-cost perfect matching problem. We mention the minimum-cost matching problem since this is the version of the problem that arises in mass transit crew scheduling [4], an application that motivated some of our efforts to look into more efficient implementations. The algorithms we describe in this paper can easily be modified to solve this latter problem.

NETWORKS, Vol. 13 (1983) 517-549

CCC 0028-3045/83/040517-33 $04.30

We define the minimum-cost perfect matching problem as a mathematical program as follows:

MCPM:

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}$$

such that

$$\sum_{j:\,(i,j)\in E} x_{ij} = 1, \quad \text{for } i \in V, \tag{1}$$

$$x_{ij} \geqq 0, \quad \text{for } (i,j) \in E, \tag{2}$$

$$x_{ij} \text{ integer for } (i,j) \in E, \tag{3}$$

where we interpret

$$x_{ij} = \begin{cases} 1, & \text{if } (i,j) \in M, \\ 0, & \text{if } (i,j) \notin M. \end{cases}$$

It is well known that constraint (3) is not superfluous to MCPM.

In this paper, we focus on implementation issues related to the MCPM. The matching algorithms we study can be viewed as iterative applications of two basic operations: growing alternating trees and updating dual variables. We first describe the operations/subroutines for handling alternating trees and give a general framework for matching algorithms (Section II). In Section III, we discuss two dual updating formulas stemming from two different linear characterizations of the MCPM. We then give two "blossom algorithms" of complexity $O(|V|^2\,|E|)$ based on these formulas. The second characterization is the so-called "coboundary characterization." To our knowledge, this is the first published statement of an algorithm based on this characterization.

In Section IV and V, we derive a set of updating formulas that provide more efficient implementations of the algorithms presented in Section III. In particular, the algorithm given in Section IV has an $O(|V|^3)$ time bound and the algorithm given in Section V has an $O(|V|\,|E|\log|V|)$ time bound. These implementations draw on the ideas of Lawler [13] and Galil, Micali, and Gabow [12]. However, certain important new features are added. In particular, the $O(|V|^3)$ algorithm has linear storage requirements whereas Lawler's algorithm has storage requirements proportional to $|V|^2$. The $O(|V|\,|E|\log|V|)$ algorithm differs from the Galil, Micali, and Gabow algorithm in that it requires a simpler priority queue and is based on the coboundary characterization mentioned above. One of the most interesting aspects of the update formulas given in Sections IV and V is that they can be interpreted in the context of the shortest augmenting path approach to the matching problem developed by Derigs [6]. This interpretation, which is given in Section VI, stimulated many of our ideas concerning efficient implementations and we feel that it should be quite useful to other researchers as well.

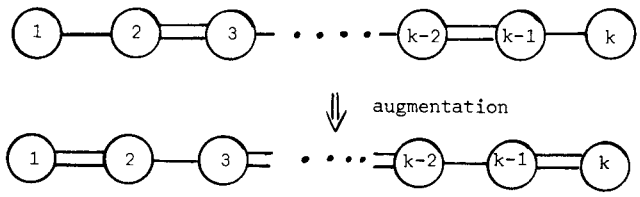


FIG. 1. (═ indicates an edge in M.)

Section VII shows that postoptimality analysis can be simply performed within this general framework.

Before describing the mathematical programming theory underlying the algorithm, we define certain graph structures used by the algorithm. We then show how to adapt this algorithm to achieve different levels of computational complexity. We assume a general knowledge of graphs; however, we now define some terminology that might be specific to this paper. A *path* is a sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$. A *simple path* is a path with $v_i \neq v_j$ for all $i \neq j$. A *circuit* is a path with $v_1 = v_k$ and $v_i \neq v_j$ for all other $i \neq j$. Given a matching M an *alternating path with respect to* M is a simple path whose edges are alternately in M and not in M. An *augmenting path* is an alternating path whose first and last vertices are not members of $V(M)$. The significance of an augmenting path P is that $|M|$ may be increased by 1 by setting $M = (M \backslash P) \cup (P \backslash M)$ (see Fig. 1).

A rooted alternating tree is a tree $T = (V(T), E(T))$ with distinguished root vertex $r \in V \backslash V(M)$ with the properties that the paths from r to each vertex in T are alternating paths and $E(T) \cap M$ is a perfect matching with respect to $V(T) \backslash \{r\}$. We designate vertices in a rooted alternating tree as even or odd depending on whether the number of edges in the path from r to the vertex is even or odd. Within the algorithm we will assign a label $l_i = +$ to even vertices i, $l_i = -$ to odd vertices i, and $l_i = 0$ to vertices not in the tree. If an unmatched vertex is adjacent to an even vertex of an alternating tree then an augmenting path can be obtained by appending the unmatched vertex to the tree (see Fig. 2).

The algorithms we describe iteratively increase $|M|$ by growing alternating tress and identifying augmenting paths with special cost properties. These tree growing activities can be thwarted by the presence of certain odd circuits C, which must be treated in a special way. Given an odd set of vertices $S \subseteq V$, we *shrink* S by forming the graph

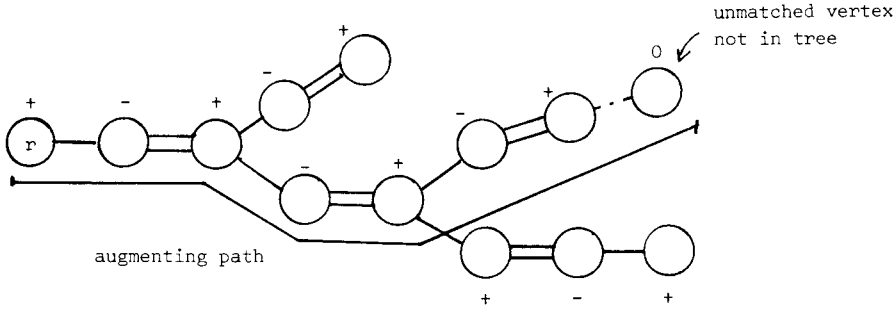
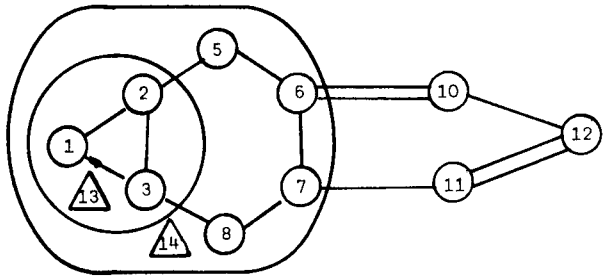


FIG. 2.

$G' = (V', E')$ where $V' = V \backslash S \cup \{v'\}$ and $E' = E \backslash \{(i, j)$: i or $j \in S\} \cup \{(v', j)$: $\exists$ an $(i, j) \in E$ with $i \in S$ and $j \notin S\}$.

We call v' the *pseudovertex induced by* S. Given a pseudovertex k we *expand* k by reversing the process described above. During the course of the algorithm we may perform the shrink operation a number of times. In particular, the shrink operation may be invoked recursively, in the sense that set S may contain pseudovertices. We call the current set V' of vertices, i.e., vertices not contained within a pseudovertex, *exterior vertices*; all vertices that are contained within some pseudovertex are called *interior vertices*. The algorithm does not explicitly delete interior vertices and their adjacent edges. In particular, the sets E and V will always denote the original edge and vertex sets. For any real vertex i (i.e., $i \in V$) we denote by $b(i)$ the exterior vertex that contains i. For any exterior vertex k, we denote by REAL(k) the set of real vertices contained in k. Note that if a real vertex i is an exterior vertex then $b(i) = i$ and REAL(i) = $\{i\}$. Figure 3(a) illustrates these definitions.

Applying the shrinking procedure iteratively produces a graph $G' = (V', E')$ where several sets of $S \subseteq V$ of odd cardinality have been shrunk to pseudovertices. These sets are associated with systems of "nested" odd cycles and have the special property that whenever any vertex in S is matched with a vertex outside S the remaining set of vertices in S can be matched using edges having both ends in S only. The signif-



b(1) = b(2) = b(3) = b(5) = b(6) = b(7) = b(8) = 14 ; b(10) = 10 ; b(11) = 11 ;
b(12) = 12 .

REAL(14) = {1,2,3,5,6,7,8} ; REAL(10) = {10} ; REAL(11) = {11} ; REAL(12) = {12} .

FIG. 3(a).

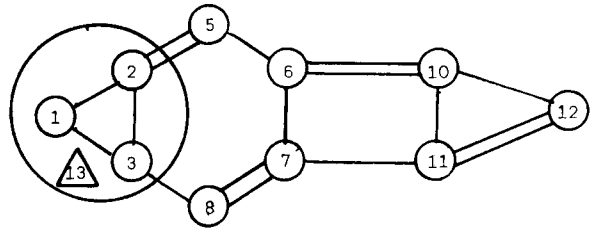


FIG. 3(b). Graph from (a) after vertex 14 is expanded (with matching extended).

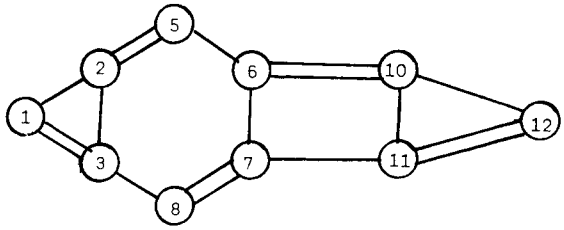


FIG. 3(c). Graph from (b) after vertex 13 is expanded (with matching extended).

icance of these so-called "hypomatchable" sets was first investigated by Edmonds and Pulleyblank [11]. The importance of these structures comes from the fact that whenever a perfect matching in $G' = (V', E')$ can be found, this matching can be canonically extended to a perfect matching in the original graph $G = (V, E)$.

Figure 3 illustrates how a perfect matching in a graph with pseudovertices can be extended to a perfect matching in the original graph. Note that for any pseudovertex, once $i \in S$ to be covered by an edge not in C is determined, then the members of C to be included in M are also determined.

II. GENERAL FRAMEWORK FOR MATCHING ALGORITHMS

We now describe four subprocedures which constitute the basic components of our algorithm. The algorithm maintains an alternating tree; these subprocedures perform certain operations relative to this tree. The operations are the GROW operation, which adds a matched pair of vertices to the tree; the SHRINK operation, which shrinks an odd circuit into a pseudovertex; the EXPAND operation, which expands a pseudovertex; and the AUGMENT operation, which augments the matching over an augmenting path through the tree. These operations are specified below and illustrated using the graph of Figure 4(a).

In the specifications, note that each subprocedure calls an update routine. These update routines update dual variables and related labels. They will be described in detail for several different implementations. The value D passed to the subprocedure is not relevant to the current discussion but will be used later in the paper.

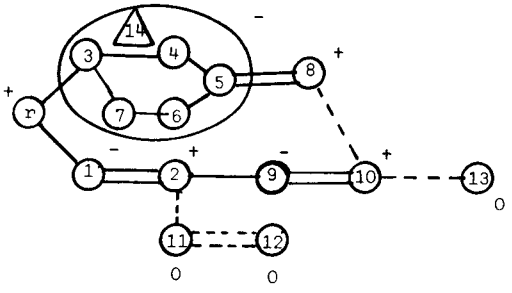


FIG. 4(a). Initial graph: (——) edges in tree, (- -) edges not in tree, (═══) edges in matching.

GROW(i, j, D)
Let k be the exterior vertex matched to i;
attach i to the tree at j and k to the tree at i;
set $l_i = -$ and $l_k = +$;
G-UPDATE(i, k, D).

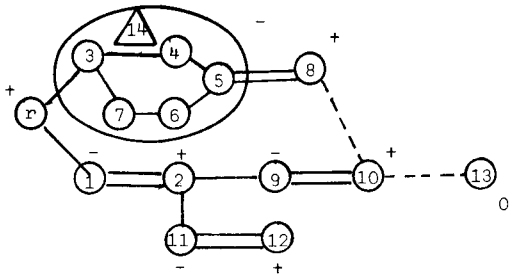


FIG. 4(b). Result of GROW(11, 2, D).

SHRINK(i, j, D)
Find CYC, the circuit of exterior vertices formed by adding edge (i, j) to the tree;
let k be a free pseudovertex index;
S-UPDATE(CYC, k, D);
shrink CYC into pseudovertex k.

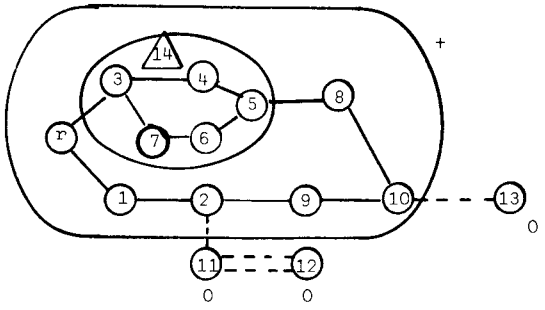


FIG. 4(c). Result of SHRINK(8, 10, D).

EXPAND(i, D)
Expand pseudovertex i;
set $l_k = +, -$, or 0 for new exterior vertices k as appropriate;
let SA be the new exterior vertices k with $l_k = +$,
TA the new exterior vertices k with $l_k = -$, and
UA the new exterior vertices k with $l_k = 0$;
E-UPDATE(i, SA, TA, UA, D);
add i to the free pseudovertex list.

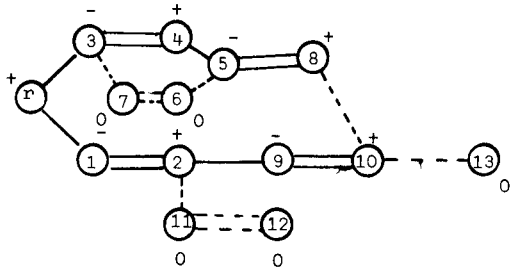


FIG. 4(d). Result of EXPAND(14, D): $SA = \{4\}$, $TA = \{3, 5\}$, $UA = \{7, 6\}$.

AUGMENT(i, j, D)

Augment the matching along the path that starts at i, proceeds to j, and then follows the tree path from j to r;

A-UPDATE(D);

Set all data structures associated with the current tree to their null values.

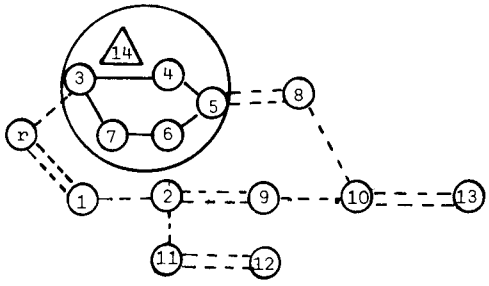


FIG. 4(e). Result of AUGMENT(13, 10, D).

We now describe MATCH, a general version of the matching algorithm. In addition to the four subprocedures just described this algorithm requires I-UPDATE(r; D) and MIN-I(; i, D). (Note that in the variable list for subprocedures, the variables preceding the semicolon are input variables and the variables following the semicolon are output variables.) I-UPDATE initializes certain labels and MIN-I either chooses the edge or vertex which determines each successive step of the algorithm or determines that no augmenting path rooted at r exists. If the next step of the algorithm is to be an EXPAND then MIN-I finds the appropriate pseudovertex i to be expanded. If the next step is to be a GROW then MIN-I sets i to the exterior vertex to be added to the tree and $p(i)$ to the *real* vertex in the tree to which i is to be attached. If the next step is to be a SHRINK, then MIN-I sets i to be one of the vertices in the tree and $p(i)$ equal to j where j is the real vertex to which i is to be attached to form the new pseudovertex. If the next step is to be an AUGMENT, MIN-I sets i to be the unmatched

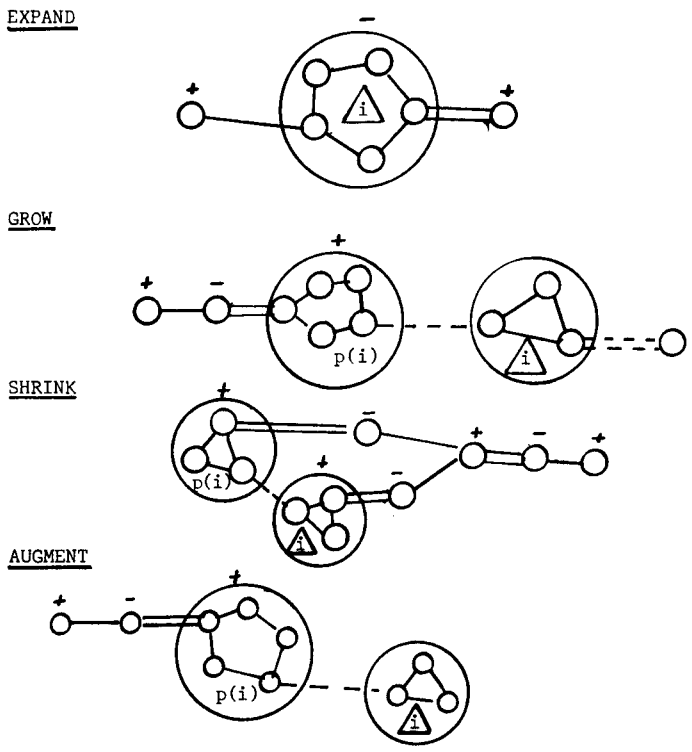


FIG. 5.

vertex that starts the augmenting path and $p(i)$ to be the real vertex within the tree that starts the tree portion of the augmenting path. If no augmenting path rooted at r exists then MIN-I sets i to zero. This indicates that no perfect matching exists so the algorithm immediately terminates. Figure 5 illustrates these definitions.

Throughout this paper we assume that graph G is connected.

MATCH (V, E, c)
  Initialize dual variables and graphical structures;
  while an unmatched vertex r exists:
    set $l_r = +$ and $l_k = 0$ for all exterior vertices $k \neq r$,
    I-UPDATE(r; D),
    MIN-I(; i, D),
    while i is matched and $i \neq 0$:
      if $l_i = 0$ then GROW(i, $b(p(i))$, D),
      if $l_i = +$ then SHRINK(i, $b(p(i))$, D),
      if $l_i = -$ then EXPAND(i, D),
      MIN-I(; i, D),
    if $i \neq 0$ then AUGMENT(i, $b(p(i))$, D),
    otherwise, stop, no perfect matching exists,
  a minimum-cost perfect matching has been found in the reduced graph, extend it to a perfect matching in G.

The key to the efficiency of MATCH is the efficient implementation of MIN-I. MIN-I, I-UPDATE, and the update routines previously mentioned will be varied to achieve different time bounds on the algorithm.

III. BLOSSOM ALGORITHMS

The now well-known blossom algorithm [10, 13] makes use of a linear characterization of the solutions to MCPM. We describe two algorithms which are based on two different characterizations. Before giving the constraint sets we define $\mathcal{S} = \{S \subseteq V: |S| \geq 3, \text{odd}\}$ and for all $S \subseteq V$

$$\delta(S) = \{(i,j) \in E: i \in S \text{ and } j \notin S\},$$
$$\gamma(S) = \{(i,j) \in E: i,j \in S\}.$$

The two sets of linear constraints are

$$\sum_{(i,j)\in\gamma(S)} x_{ij} \leqq \frac{(|S|-1)}{2}, \quad \text{for all } S \in \mathcal{S}, \tag{I}$$

$$\sum_{(i,j)\in\delta(S)} x_{ij} \geqq 1, \quad \text{for all } S \in \mathcal{S}. \tag{II}$$

The importance of these sets of constraints is that any extreme point solution to

LI:

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}$$

such that (1), (2), (I), or to

LII:

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}$$

such that (1), (2), (II), is a solution to MCPM.

In fact, the algorithms we describe, while they do not explicitly deal with the entire constraint sets of LI or LII, do rely very heavily on the structure of LI and LII.

To understand the manner in which the algorithms use this structure we must consider the dual linear programs and the related complementarity conditions. The dual of LI is

DI:

$$\max \sum_{i\in V} y_i - \sum_{S\in\mathcal{S}} \tfrac{1}{2}(|S|-1)\, y_S$$

such that

$$y_i + y_j - \sum_{\substack{S \in \delta \text{ with} \\ (i,j) \in \gamma(S)}} y_S \leqq c_{ij}, \quad \text{for } (i,j) \in E, \tag{4}$$

$$y_S \geqq 0, \quad \text{for } S \in \delta. \tag{5}$$

The usual linear programming reduced cost is defined by

$$c'_{ij} = c_{ij} - y_i - y_j + \sum_{\substack{S \in \delta \text{ with} \\ (i,j) \in \gamma(S)}} y_S,$$

and the complementarity conditions are

either $$x_{ij} = 0 \quad \text{or} \quad c'_{ij} = 0, \quad \text{for } (i,j) \in E, \tag{6}$$

either $$y_S = 0 \quad \text{or} \quad \sum_{(i,j) \in \gamma(S)} x_{ij} = \frac{(|S| - 1)}{2}, \quad \text{for } S \in \delta. \tag{7}$$

The dual of LII is

DII:

$$\max \sum_{i \in V} y_i + \sum_{S \in \delta} y_S$$

such that

$$y_i + y_j + \sum_{\substack{S \in \delta \text{ with} \\ (i,j) \in \delta(S)}} y_S \leqq c_{ij}, \quad \text{for } (i,j) \in E, \tag{8}$$

$$y_S \geqq 0, \quad \text{for } S \in \delta. \tag{9}$$

The usual linear programming reduced cost is defined by

$$c'_{ij} = c_{ij} - y_i - y_j - \sum_{\substack{S \in \delta \text{ with} \\ (i,j) \in \delta(S)}} y_S$$

and the complementarity conditions are

either $$x_{ij} = 0 \quad \text{or} \quad c'_{ij} = 0, \quad \text{for } (i,j) \in E, \tag{10}$$

either $$y_S = 0 \quad \text{or} \quad \sum_{(i,j) \in \delta(S)} x_{ij} = 1, \quad \text{for } S \in \delta. \tag{11}$$

In this section we describe Algorithms I-A and II-A, which are based on characterizations I and II, respectively. Section IV presents Algorithm I-B, a more efficient

implementation of Algorithm I-A, and Section V presents Algorithm II-B, a more efficient implementation of Algorithm II-A.

Both Algorithms I-A and II-A can be viewed as primal-dual linear programming algorithms, with Algorithm I-A working with characterization I and Algorithm II-A working with characterization II. In both cases dual feasibility and complementary slackness are maintained. In addition, $c'_{ij} = 0$ is maintained for all edges in the current alternating tree so that when an augmenting path through the tree is found the complementarity conditions will still be maintained after the augmentation is performed. The algorithm works toward primal feasibility [constraint (1)] by iteratively finding a matching M of larger cardinality until finally $|M| = \frac{1}{2}|V|$, which implies that M is a perfect matching.

We describe each algorithm by specifying the functions and routines required by the general algorithm given in the previous section. Within the algorithms we assign values to y variables associated with both real and pseudovertices. The y values associated with pseudovertices represent y_S for the associated odd-cardinality vertex sets.

For Algorithm I-A we define the following functions:

$$CU(i,j) = c_{ij} - y_i - y_j, \qquad \text{for } (i,j) \in E,$$

$$CS(i,j) = \tfrac{1}{2}(c_{ij} - y_i - y_j), \qquad \text{for } (i,j) \in E,$$

$$CT(k) = \tfrac{1}{2} y_k, \qquad \text{for pseudovertices } k.$$

Note that for $(i,j) \in E$ with $b(i) \neq b(j)$, $CU(i,j) = c'_{ij}$.

We now specify the subprocedures required by MATCH. Throughout this paper we define the minimum over an empty set to be ∞.

MIN-*I*(; *i*, *D*)

set $\alpha_1 = \min\{CU(k,j)$: $(k,j) \in E$ with $l_{b(k)} = 0$ and $l_{b(j)} = +\}$;
set $\alpha_2 = \min\{CS(k,j)$: $(k,j) \in E$ with $l_{b(j)} = l_{b(k)} = +$ and $b(j) \neq b(k)\}$;
set $\alpha_3 = \min\{CT(k)$: k an exterior pseudovertex with $l_k = -\}$;
set $\alpha = \min\{\alpha_1, \alpha_2, \alpha_3\}$;
if $\alpha = \infty$ then set $i = 0$ and return;
if $\alpha = \alpha_1$ or α_2 then let (k^0, j^0) be an edge that achieves the minimum and set $i = b(k^0)$ and $p(i) = j^0$,
otherwise let k^0 be a pseudovertex that achieves the α_3-minimum and set $i = k^0$;
set $D = D + \alpha$;
D-UPDATE(α).

***D*-UPDATE(α)**

For all exterior vertices k with $l_k = +$:
  if k is a pseudovertex then set $y_k = y_k + 2\alpha$,
  for all $i \in \text{REAL}(k)$ set $y_i = y_i + \alpha$;
for all exterior vertices k with $l_k = -$:
  if k is a pseudovertex then set $y_k = y_k - 2\alpha$,
  for all $i \in \text{REAL}(k)$ set $y_i = y_i - \alpha$.

***I*-UPDATE(r; *D*)**

Set $D = 0$.

***S*-UPDATE(CYC, k, *D*)**
Set $y_k = 0$.

G-UPDATE, *E*-UPDATE, and *A*-UPDATE are vacuous in this case.

In Algorithm II–A we explicitly keep track of the reduced costs c'_{ij}. The algorithm maintains a variable $\bar{c}_{ij}$ which equals c'_{ij} for $(i,j) \in E$ with $b(i) \neq b(j)$. Before defining $\bar{c}_{ij}$ completely we define for any vertex k, real or pseudo, $I(k)$ as the set of all vertices that contain k (including k itself). Now, with the y_k defined with respect to DII, we define for all $(i,j) \in E$

$$\bar{c}_{ij} = c_{ij} - \sum_{k \in I(i)} y_k - \sum_{k \in I(j)} y_k.$$

Thus for $(i,j) \in E$ with $b(i) \neq b(j)$, $c'_{ij} = \bar{c}_{ij}$, and for $(i,j) \in E$ with $b(i) = b(j)$, $c'_{ij} = \bar{c}_{ij} + 2\sum_{k \in I(i) \cap I(j)} y_k$.

This definition might seem slightly unnatural, especially with respect to (i,j) with $b(i) = b(j)$. However, when we extend this algorithm to a more efficient version, this definition of $\bar{c}_{ij}$ becomes very useful.

For Algorithm II-A we first redefine the functions required by MIN-*I*:

$$CU(i,j) = \bar{c}_{ij}, \qquad \text{for } (i,j) \in E,$$
$$CS(i,j) = \tfrac{1}{2}\bar{c}_{ij}, \qquad \text{for } (i,j) \in E,$$
$$CT(k) = y_k, \qquad \text{for pseudovertices } k.$$

MIN-*I* is the same as for Algorithm I-A with the following revised version of *D*-UPDATE.

***D*-UPDATE(α)**
For all exterior vertices k with $l_k = +$:
set $y_k = y_k + \alpha$,
for all $i \in \text{REAL}(k)$ and $(i,j) \in E$ set $\bar{c}_{ij} = \bar{c}_{ij} - \alpha$;
for all exterior vertices k with $l_k = -$:
set $y_k = y_k - \alpha$,
for all $i \in \text{REAL}(k)$ and $(i,j) \in E$ set $\bar{c}_{ij} = \bar{c}_{ij} + \alpha$.

The remaining routines required by MATCH for Algorithm II-A are identical to those required by Algorithm I-A.

Throughout the paper we assume that the graph is stored using list structures that enable all edges adjacent to a particular vertex to be scanned in $O(\beta)$ time where β is the degree of that vertex.

Theorem 1 provides a bound on the computation time required by Algorithms I-A and II-A.

Theorem 1. Both Algorithms I-A and II-A require at most $O(|V|^2\,|E|)$ time.

Proof. We refer to an iteration of the algorithm as a cycle in which a tree is grown and an augmenting path is determined. Since after each iteration $|M|$ increases by 1, the number of iterations can be no greater than $\frac{1}{2}|V|$.

It is clear that finding a new root vertex r and each call to AUGMENT and I-UPDATE can be implemented to use at most $O(|V|)$ time, and that each call to MIN-I can be performed in $O(|E|)$ time. Thus all operations outside the innermost *while* loop require at most $O(|V|\,|E|)$ time.

We now show that all operations within the loop use at most $O(|V|\,|E|)$ time per iteration. It can be demonstrated by induction that the number of pseudovertices that can simultaneously exist is bounded by $\frac{1}{2}(|V| - 1)$ (see [16]). Since only vertices with a $-$ label can be expanded and newly formed pseudovertices are given a $+$ label, it follows that a pseudovertex formed during a particular iteration can never be expanded during the same iteration. These two facts imply that each of SHRINK, EXPAND, GROW, and MIN-I can be called at most $O(|V|)$ times during a particular iteration. The basic tree and blossom maintenance operations required in SHRINK, EXPAND, and GROW can be implemented in $O(|V|)$ time. The data structures and procedures required to achieve this bound can be found in [7]. The above facts also imply that MIN-I can be called at most $O(|V|)$ times per iteration so that, since each call uses at most $O(|E|)$ time, in total MIN-I uses at most $O(|V|\,|E|)$ time per iteration. It now follows that the overall complexity of the algorithm is no more than $O(|V|^2\,|E|)$. ∎

IV. AN $O(|V|^3)$ IMPLEMENTATION

In this section we present an $O(|V|^3)$ algorithm. This implementation draws on the ideas of Lawler [13] and of Derigs [6]. In addition to unifying these two approaches, the algorithm presented here includes a new method for keeping track of edges that are potential candidates for blossom formation. This method, which is based on a specialized minimum spanning tree algorithm [2], has linear storage requirements whereas Lawler's methods require the maintenance of matrix whose size is proportional to $|V|^2$.

As mentioned above, Algorithm I-A keeps track of a variable D which is not essential to its execution. We now define certain variables in terms of D and show how these lead to a more efficient implementation, Algorithm I-B. For each vertex k we define a variable d_k; d_k is assigned the current value of D when vertex k first appears in the tree as an exterior vertex. In particular, whenever GROW(i, j, D) is called d_i and d_k are set to D; whenever SHRINK(i, j, D) is called d_k is set equal to D where k is the newly formed pseudovertex; finally, whenever EXPAND(i, D) is called d_k is set equal to D for all $k \in SA \cup TA$. Furthermore, we define a set of "partially updated" dual variables $\bar{y}_k$. Whenever an exterior vertex k is added to the tree $\bar{y}_k$ is set equal to the current value of y_k. For each real vertex i contained in some pseudovertex $b(i)$, $\bar{y}_i$ is set equal to the value of y_i at the time the current $b(i)$ vertex was added to the tree. For all exterior vertices i not in the tree and all interior vertices i, $\bar{y}_i = y_i$. Based on these conditions it is easy to see that whenever MIN-I is called the following relations hold:

$$y_k = \bar{y}_k + 2(D - d_k), \quad \text{for exterior pseudovertices } k \text{ with } l_k = +,$$

$$y_k = \bar{y}_k - 2(D - d_k), \quad \text{for exterior pseudovertices } k \text{ with } l_k = -,$$

$$y_i = \bar{y}_i + D - d_{b(i)}, \quad \text{for real vertices } i \text{ with } l_{b(i)} = +,$$

$y_i = \bar{y}_i - (D - d_{b(i)})$, for real vertices i with $l_{b(i)} = -$,

$y_k = \bar{y}_k$, for interior pseudovertices k and real vertices k with $l_{b(k)} = 0$.

Thus we have that

$$\alpha_1 = \min\{c_{kj} - \bar{y}_k - \bar{y}_j + d_{b(j)}: (k,j) \in E, l_{b(k)} = 0 \text{ and } l_{b(j)} = +\} - D,$$

$$\alpha_2 = \min\{\tfrac{1}{2}(c_{kj} - \bar{y}_k - \bar{y}_j + d_{b(k)} + d_{b(j)}): (k,j) \in E \text{ with } l_{b(k)} = l_{b(j)} = + \text{ and } b(k) \neq b(j)\} - D,$$

$$\alpha_3 = \min\{\tfrac{1}{2}\bar{y}_k + d_k: k \text{ an exterior pseudovertex with } l_k = -\} - D.$$

Note that we need not explicitly compute the α_i to find the edge or vertex that achieves the minimum but rather we can compute $D_i = \alpha_i + D$ using the above formulas with the final $-D$ deleted. The significance of computing the D_i rather than the α_i is that the formulas for the D_i do not explicitly require the dual variables y_i but rather only the $\bar{y}_i$ and d_i. Thus it is not necessary to update all dual variables after each pass through the innermost *while* loop of MATCH. More importantly, we show below that a variety of other efficiencies result from dealing with the D_i, d_i and $\bar{y}_i$, and not with the α_i and y_i. The principal computational efficiencies result from the following simple property.

Proposition 1. For any real vertex i, after $l_{b(i)}$ is set to +, the quantity $c_{ij} - \bar{y}_i + d_{b(i)}$ remains constant throughout the current iteration.

Proof. Once $l_{b(i)}$ is set equal to +, the only condition under which $\bar{y}_i$ or $d_{b(i)}$ can change is if $b(i)$ becomes a member of CYC during a call to SHRINK. If we denote by k' the old value of $b(i)$, by k the new value of $b(i)$, by $\hat{y}_i$ the old value of $\bar{y}_i$, and by y_i^0 the new value of $\bar{y}_i$ we have

$$\begin{aligned} c_{ij} - y_i^0 + d_k &= c_{ij} - (\hat{y}_i + D - d_{k'}) + D \\ &= c_{ij} - \hat{y}_i - d_{k'}. \end{aligned}$$

Thus the old and new values of the relevant expressions are equal. ∎

The importance of this proposition is that it greatly reduces the updating required to find the appropriate edge or vertex in MIN-I. To use this property we first redefine $CU(i,j)$, $CS(i,j)$, and $CT(k)$ as their previously defined values plus D. Now let

$$\text{MI} = \{(i,j) \in E: l_{b(i)} = + \text{ and } l_{b(j)} = 0\},$$

$$\text{MII} = \{(i,j) \in E: l_{b(i)} = + \text{ and } l_{b(j)} = + \text{ with } b(i) \neq b(j)\},$$

$$\text{MIII} = \{k: k \text{ is an exterior pseudovertex with } l_k = -\}.$$

The proposition implies that as long as $(i,j) \in \text{MI}$, the value of $CU(i,j)$ remains constant and as long as (i,j) is a member of MII the value of $CS(i,j)$ remains constant. It is also clear that as long as a vertex h is a member of MIII the value of $CT(h)$ remains

constant. Thus the problem of finding the vertex or edge in MIN-I is not a problem of updating the values of CU, CS, and CT but rather of updating membership and order in MI, MII, and MIII.

We now describe an algorithm that uses this approach. The algorithm maintains subsets of MI and MII, where it is known that edges not in the subsets need not be considered. In the case of MI, it is easy to see that for any exterior vertex k with $l_k = 0$, only one edge adjacent to an exterior vertex i with $l_i = +$ will ever be used. Consequently, we can maintain a subset of MI that includes at most one edge adjacent to each vertex with $l_k = 0$. The edge kept is the one that minimizes $CU(i, j)$.

MII consists of all edges in the subgraph induced by exterior vertices k with $l_k = +$. The important observation here is that it is not possible that edges in a circuit in this subgraph will be used to form pseudovertices. The reason for this is that once all but one edge in a circuit were used to form pseudovertices then the last edge would have both ends within the same pseudovertex (see Fig. 6). Consequently, rather than maintaining all of MII we maintain a minimum-weight forest over MII with the $CS(i, j)$ values used as weights.

We use a common set of labels to maintain the subsets of MI and MII and the set MIII. For each exterior vertex k with $l_k = 0$ let $v_k = \min\{CU(i,j)\colon (i,j) \in E, b(i) = k,$ and $l_{b(j)} = +\}$ and let $p(k) = j$ for some real vertex j which achieves the above minimum. We maintain the minimum spanning forest previously described using a predecessor array defined by the $p(k)$ for exterior vertices k with $l_k = +$. We set $v_k = CS(i, p(k))$ where $(i, p(k))$ is the edge in the forest between vertex k and $b(p(k))$. [Note that $b(i) = k$ and $b(i) \neq b(p(k))$.] Finally, for each pseudovertex k with $l_k = -$, we set $v_k = CT(k)$.

For all real vertices j we define the variables v'_j and $p'(j)$ to aid in updating the values just described. Whenever a pseudovertex is expanded, it is possible that two or more additional exterior vertices k will receive labels $l_k = 0$. To facilitate the computation of their v_k values we define for each real vertex j with $l_{b(j)} = 0$ or $l_{b(j)} = -$

$$v'_j = \min\{c_{ij} - \bar{y}_i + d_{b(i)}\colon (i,j) \in E \text{ and } l_{b(i)} = +\},$$

$$p'(j) = i, \quad \text{for a real vertex } i \text{ which achieves the above minimum.}$$

We should note that the use of the v'_j labels in this manner has been previously described by Lawler [13].

We now describe Algorithm I-B, which uses the labels just defined. The algorithm invokes the subprocedure FOREST(AS, k) to update the minimum-weight spanning forest. FOREST, which is described in [2], solves a minimum-weight spanning forest

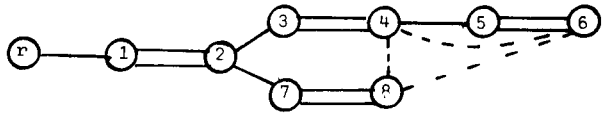


FIG. 6. Once any two members of $\{(4, 6), (6, 8), (8, 4)\}$ were used in a shrink operation the third would become ineligible because it would have both ends in the same pseudovertex.

problem over graphs in which all circuits pass through a single vertex. When FOREST is called by GROW and EXPAND, the set of +-labeled exterior vertices has just been augmented by vertex k. FOREST(AS, k) finds the minimum-weight forest using the original set of forest edges plus the set AS of edges adjacent to k and other +-labeled vertices. When SHRINK calls FOREST(AS, k), k is the newly formed pseudovertex and AS is the set of edges adjacent to k and other +-labeled vertices that were formerly adjacent to −-labeled vertices now interior to k. It is shown in [2] that FOREST (AS, k) can be implemented in $O(|V| + |AS|)$ time.

Whenever an exterior vertex i is given a + label by GROW or EXPAND or whenever an exterior vertex i with $l_i = -$ becomes interior to a +-labeled pseudovertex in SHRINK the real vertices interior to i are scanned. This scanning operation updates the v_k and v'_k values of all adjacent vertices and also finds the set AS used by FOREST. The scanning procedure SCAN is specified below.

SCAN($i, D; AS$)
Set $AS = \phi$;
for each $k \in$ REAL(i) and each $(k, j) \in E$ with $b(j) \neq i$:
if $l_{b(j)} = 0$ then set $v_{b(j)} = \min \{v_{b(j)}, c_{kj} - \bar{y}_k - \bar{y}_i\}$
and set $p(b(j)) = k$ if the value of $v_{b(j)}$ changes,
set $v'_j = \min \{v'_j, c_{kj} - \bar{y}_k\}$ and set $p'(j) = k$
if the value of v'_j changes;
if $l_{b(j)} = +$ then set $AS = AS \cup \{(k, j)\}$;
if $l_{b(j)} = -$ then set $v'_j = \min \{v'_j, c_{kj} - \bar{y}_k\}$ and set $p'(j) = k$
if the value of v'_j changes.

We can now specify MIN-I and the update procedure for Algorithm I-B.

MIN-I(; i, D)
Set $D = \min \{v_j: j$ an exterior vertex$\}$;
if $D = \infty$ then set $i = 0$ and return;
let i equal the j value that achieves this minimum;
set $v_i = \infty$.

I-UPDATE(r, D)
SCAN($r, 0, AS$);
set $v'_j = v_j = \infty$ for all vertices j.

G-UPDATE(i, k, D)
Set $d_i = d_k = D$;
if i is a pseudovertex then set $v_i = d_i + \frac{1}{2}\bar{y}_i$;
SCAN($k, D; AS$);
FOREST(AS, k).

S-UPDATE(CYC, k, D)
For each $h \in$ CYC with $l_h = +$:
if h is a pseudovertex then set $\bar{y}_h = \bar{y}_h + 2(D - d_h)$;
for each $q \in$ REAL(h) set $\bar{y}_q = \bar{y}_q + D - d_h$;
set $SS = \phi$;

for each $h \in \text{CYC}$ with $l_h = -$:
if h is a pseudovertex then set $\bar{y}_b = \bar{y}_h - 2(D - d_h)$,
for each $q \in \text{REAL}(h)$ set $\bar{y}_q = \bar{y}_q - (D - d_h)$,
SCAN$(h, D; AS)$,
set $SS = SS \cup AS$;
FOREST(SS, k);
set $\bar{y}_k = 0$ and $d_k = D$.

***E*-UPDATE(i, SA, TA, UA, D)**
For each $h \in SA$;
for each $u \in \text{REAL}(h)$ set $\bar{y}_u = \bar{y}_u - (D - d_i)$,
set $d_h = D$,
SCAN$(h, D; AS)$,
FOREST(AS, h);
for each $h \in TA$:
for each $u \in \text{REAL}(h)$ set $\bar{y}_u = \bar{y}_u - (D - d_i)$,
set $d_h = D$,
if h is a pseudovertex then set $v_h = d_h + \bar{y}_h$;
for each $h \in UA$:
for each $u \in \text{REAL}(h)$ set $\bar{y}_u = \bar{y}_u - (D - d_i)$,
set $v_h = \min \{v'_u - \bar{y}_u : u \in \text{REAL}(h)\}$ and
set $p(h) = p(u^0)$ where u^0 achieves the above minimum.

***A*-UPDATE(D)**
For each exterior vertex k with $l_k = +$:
if k is a pseudo-vertex then set $\bar{y}_k = \bar{y}_k + 2(D - d_k)$,
for each $i \in \text{REAL}(h)$ set $\bar{y}_i = \bar{y}_i - (D - d_k)$;
for each exterior vertex k with $l_k = -$:
if k is a pseudo-vertex then set $\bar{y}_k = \bar{y}_k - 2(D - d_k)$,
for each $i \in \text{REAL}(k)$ set $\bar{y}_i = \bar{y}_i - (D - d_k)$;

We now show that algorithm I-B has a time bound superior to Algorithm I-A, and II-A.

Theorem 2. Algorithm I-B requires at most $O(|V|^3)$ time.

Proof. As was described in the proof of Theorem 1, the algorithm's running time is dominated by the time required to execute the procedure in the innermost *while* loop. The principle computational advantage of Algorithm I-B is that MIN-*I* can now be executed in $O(|V|)$ time. As was mentioned previously the basic tree and blossom maintenance operations require no more than $O(|V|)$ time per call to GROW, SHRINK, and EXPAND. Consequently, the total time required for these operations and for MIN-*I* is no greater than $O(|V|^3)$.

We must now show that the additional updating required in the calls to *G*-UPDATE, *S*-UPDATE, and *E*-UPDATE has a similar bound. First notice that the dual variable updates in *S*-UPDATE and *E*-UPDATE and the minimizations required to compute v_h for $h \in UA$ in *E*-UPDATE all involve scanning disjoint sets of real vertices and consequently require at most $O(|V|)$ time per call. The remaining operations are dominated by the calls to SCAN and FOREST. For ease in the exposition of this proof we

assume that whenever l_k is set for an exterior vertex k then l_i is set to the same value for $i \in \text{REAL}(k)$. The key property of Algorithm I-B is that whenever the edges adjacent to a real vertex i are scanned the value of l_i has just changed from 0 or − to + and that once l_i is set to + it remains + throughout the current iteration. Thus an edge is scanned at most twice per iteration, once for each of its end vertices. Thus during a particular iteration all scanning operations use at most $O(|E|)$ time.

It also follows from the above arguments that the sets AS of edges passed to FOREST in a particular iteration are disjoint. Since each call to FOREST(AS, k) requires no more than $O(|V| + |AS|)$ time the total time required by FOREST per iteration is no more than $O(|V|^2 + |E|)$ time. We now have that the overall complexity of Algorithm I-B is $O(|V|^3)$. ■

V. AN $O(|V|\,|E| \log |V|)$ IMPLEMENTATION

It was demonstrated in the previous section that when using the d_k a large number of the values over which MIN-I chooses remained constant from one call to the next. This fact suggests the use of priority queues to store the lists scanned by MIN-I. Here we use priority queue to mean any list of elements in which a value is associated with each element, and such that the operations of finding the minimum value element, adding an element and deleting an element can all be performed in $\log k$ time, where k is the size of the list [1]. In this section we present a more efficient version of Algorithm II-A that uses these ideas.

An $O(|V|\,|E| \log |V|)$ algorithm has been previously described by Galil, Micali, and Gabow [12]. The algorithm we present is based on their algorithm; however, we have simplified certain ideas and, in addition, our algorithm is based on characterization II whereas theirs is based on characterization I. The concept that is crucial to our algorithm, which was first described in [12], is a "splittable" priority queue. Such a priority queue has all the priority queue properties described above and, in addition, can be split, in a well-defined way, into two priority queues in $\log k$ time. The split is defined in terms of an ordering of the elements. It should be emphasized that this ordering is different from the ordering based on the element values. A split is performed at a particular element. The result of the split is two priority queues, one containing all elements less than the chosen element and the other containing all elements greater than or equal to the chosen element. Here less and greater are with respect to the ordering and not the element values.

Within our algorithm, a splittable priority queue will be associated with certain pseudovertices i. The elements on this priority queue will be the vertices in REAL(i). The elements will be ordered so that whenever a pseudovertex i is expanded the priority queue associated with i can be split into priority queues associated with each new exterior vertex generated after the expand. Further, the ordering must allow this property to hold after recursive expansions. As Figure 7 illustrates, such an ordering can always easily be obtained.

We now describe the data structures used by Algorithm II-B.

Starting with Algorithm II-A we define d_k and $\bar{y}_k$ in a manner analogous to the way they were defined in terms of Algorithm I-A in the previous section. That is, whenever an exterior vertex k is added to the tree $\bar{y}_k$ is set to y_k and d_k is set to D. As the algorithm proceeds $\bar{y}_k$ is not updated to y_k unless a shrink or expand involving k takes

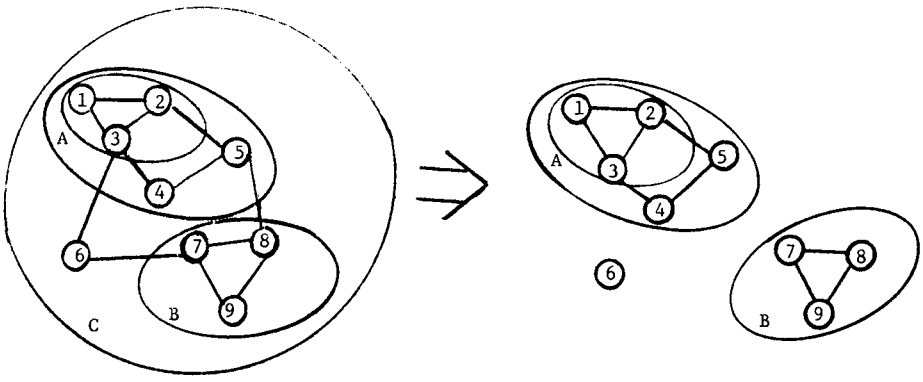


FIG. 7. The vertices are ordered according to the numbering given. The priority queue associated with blossom C contains $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$. When C is expanded this priority queue is split at 6 and at 7 to obtain three priority queues which contain $\{1, 2, 3, 4, 5\}$, $\{6\}$, and $\{7, 8, 9\}$. These correspond to the three (pseudo)vertices uncovered by the expansion.

place. Thus, with these definitions it is easy to see that

$$y_k = \bar{y}_k + D - d_k, \qquad \text{for exterior vertices } k \text{ with } l_k = +,$$

$$y_k = \bar{y}_k - (D - d_k), \qquad \text{for exterior vertices } k \text{ with } l_k = -,$$

$$y_k = \bar{y}_k, \qquad \text{for all interior vertices } k \text{ and exterior vertices } k \text{ with } l_k = 0.$$

Algorithm II-B only updates the $\bar{c}_{ij}$ values at the beginning of an iteration so that we define, for all $(i, j) \in E$, $\tilde{c}_{ij}$ as the value of $\bar{c}_{ij}$ at the beginning of the current iteration. To enable us to compute the $\bar{c}_{ij}$ based on the $\tilde{c}_{ij}$, we define for each vertex k that existed at the beginning of the current iteration,

$$e_k = \sum_{j \in I(k) \setminus \{k\}} y_j$$

where the $I(k)$ used in the above sum is its value at the beginning of the current iteration.

Although the e_k do not change during an iteration we find it most convenient to update them dynamically as expands take place. It is possible to compute required reduced costs from the quantities just defined. For any $(i, j) \in E$ for which $b(i)$ and $b(j)$ existed at the beginning of the current iteration (i.e., neither i nor j was involved in a shrink operation), we have

$$c'_{ij} = \tilde{c}_{ij} + \gamma_i + \gamma_j,$$

where

$$\gamma_k = \begin{cases} e_{b(k)}, & \text{if } l_{b(k)} = 0, \\ e_{b(k)} + (D - d_{b(k)}), & \text{if } l_{b(k)} = -, \\ e_{b(k)} - (D - d_{b(k)}), & \text{if } l_{b(k)} = +. \end{cases}$$

We now define for any $(i,j) \in E$ with $l_{b(i)} = +$,

$$d_{ij} = \begin{cases} d_{b(i)} + c_{ij} - \sum_{k \in I(i)} \bar{y}_k - e_j - \bar{y}_j, & \text{if } l_{b(j)} \neq +, \\ 1/2(d_{b(i)} + d_{b(j)} + \bar{c}_{ij}), & \text{if } l_{b(j)} = +. \end{cases}$$

Note that we are defining a single d_{ij} value for each edge (i,j). The usefulness of the d_{ij} arises from the following:

$$d_{ij} + e_{b(j)} = D + c'_{ij} \quad \text{for } (i,j) \in E \text{ with } l_{b(i)} = + \text{ and } l_{b(j)} = 0,$$
$$d_{ij} = D + 1/2 c'_{ij} \quad \text{for } (i,j) \in E \text{ with } l_{b(i)} = l_{b(j)} = + \text{ with } b(i) \neq b(j).$$

Note that it is also true that

$$d_k + \bar{y}_k = D + y_k, \qquad \text{for exterior pseudovertices } k \text{ with } l_k = -.$$

Thus, by taking the minimum over the above three quantities we obtain the same vertex or edge obtained by MIN-I of Algorithm II-A. Proposition 2 gives a property which facilitates updating required to keep track of the d_{ij}.

Proposition 2. For any real vertex i and $(i,j) \in E$, after $l_{b(i)}$ is set to $+$, $\bar{d} = D + c_{ij} - \sum_{k \in I(i)} y_k$ remains constant throughout the current iteration.

Proof. Let h be the first exterior vertex with $l_h = +$ that contained i. At the time l_h was first set to + we have that

$$\begin{aligned} \bar{d} &= d_h + c_{ij} - \sum_{k \in I(i)} y_k \\ &= d_h + c_{ij} - \sum_{k \in I(i) \setminus \{h\}} y_k - \bar{y}_h. \end{aligned}$$

Now as long as $b(i) = h$, $y_h = \bar{y}_h + (D - d_h)$ so that

$$\bar{d} = D + c_{ij} - \sum_{k \in I(i) \setminus \{h\}} y_k - (\bar{y}_h + D - d_h),$$

which equals $\bar{d}$ at the time l_h is first set to +. We now only need to show that $\bar{d}$ remains constant when $b(i)$ changes; i.e., after a shrink involving i takes place. Now let $h = b(i)$ before the shrink and $h' = b(i)$ after the shrink. Then after the shrink

$$\begin{aligned} \bar{d} &= D + c_{ij} - \sum_{k \in I(i) \setminus \{h'\}} y_k - y_{h'} \\ &= D + c_{ij} - \sum_{k \in I(i) \setminus \{h, h'\}} y_k - [\bar{y}_h + (D - d_h)] - 0. \end{aligned}$$

But it is clear that this last quantity equals $\bar{d}$ at the time h is added to the tree. ∎

The implications of Proposition 2 on updating the d_{ij} are given by the following corollary.

Corollary 1. For any $(i, j) \in E$, as long as $l_{b(i)} = +$ and $l_{b(j)} \neq +$, d_{ij} remains constant and as long as $l_{b(i)} = l_{b(j)} = +$, d_{ij} remains constant.

We now define the values used by the priority queues to determine the appropriate minimums which guide the successive steps of Algorithm II-B. Several priority queues are maintained. PQ-V contains exterior vertices not in the tree and $-$-labeled exterior pseudovertices. PQ-E contains edges adjacent to a pair of $+$-labeled vertices. Finally, a splittable priority queue PQ-P(k) is associated with each exterior pseudovertex k that is either 0 or $-$ labeled. The vertices k in PQ-V are ordered by v_k where

$$v_k = \begin{cases} \min \{d_{ij} + e_{b(j)} : (i,j) \in E, j \in \text{REAL}(k), l_{b(i)} = +\}, \\ \qquad\qquad \text{for exterior vertices } k \text{ with } l_k = 0; \\ d_k + \bar{y}_k, \qquad \text{for exterior pseudovertices } k \text{ with } l_k = -. \end{cases}$$

PQ-E contains edges (i, j) with $l_{b(i)} = l_{b(j)} = +$.

These edges are ordered by d_{ij}. It follows directly from the definitions that the minimum of the v_k and the d_{ij} with $b(i) \neq b(j)$ over these two lists yields the appropriate vertex or edge for the next step of the alrogithm.

The splittable priority queues PQ-P(k) are necessary to allow for easy determination of the v_k values after expands. The real vertices on these priority queues are ordered by v_j' where

$$v_j' = \min \{d_{ij} : l_{b(i)} = + \text{ and } (i,j) \in E\}.$$

The pointer $p'(j)$ is set to the i value that achieves the above minimum. The usefulness of the v_j' arises from the following property:

$$v_k = \min \{v_j' : j \in \text{REAL}(k)\} + e_k.$$

We can now define the subprocedures for Algorithm II-B.

***I*-UPDATE(r; D)**
Set $d_{ij} = \infty$ for all $(i, j) \in E$;
set $v_k = \infty$, $e_k = 0$, and $l_k = 0$ for all exterior vertices k;
set $d_r = D = 0$;
SCAN(r, D).

SCAN(i, D)
For each $k \in \text{REAL}(i)$ and each $(k, j) \in E$:
if $l_{b(j)} \neq +$ or $l_{b(j)} = +$ and $d_{kj} = \infty$ then
set $d_{kj} = D + \tilde{c}_{kj} + e_i + D - d_i$,
otherwise set $d_{kj} = 1/2(d_{kj} + D + e_i + D - d_i)$,
if $l_{b(j)} = 0$ then set $v_{b(j)} = \min \{v_{b(j)}, d_{kj} + e_{b(j)}\}$ and
if the value of $v_{b(j)}$ changes in the above minimum then
set $p(b(j)) = k$ and adjust the position of $b(j)$ on PQ-V,

if $l_{b(j)} = +$ and $b(k) \neq b(j)$ then place (k, j) on PQ-E,
if $l_{b(j)} \neq +$ then set $v'_j = \min \{v'_j, d_{kj}\}$
if the value of v'_j changes in the above minimum then
set $p'(j) = k$ and adjust the position of j on PQ-P($b(j)$).

MIN-I(; i, D)
Set $D_1 = \min \{v_j : j \text{ on PQ-V}\}$;
set $D_2 = \infty$;
if PQ-E is not empty and for the top edge (k, j), $b(k) = b(j)$,
then take edges off of PQ-E until PQ-E is empty or
for the top edge (k, j), $b(k) \neq b(j)$;
if PQ-E is not empty then set $D_2 = d_{kj}$;
set $D = \min \{D_1, D_2\}$;
if $D = \infty$ then set $i = 0$ and return;
if $D = D_1$ then set i equal to the vertex on top of PQ-V and remove i from PQ-V;
otherwise $(D = D_2)$, let (k, j) be the top edge on PQ-E, set $i = b(k)$, $p(i) = j$, and
remove (k, j) from PQ-E.

G-UPDATE(i, k, D)
Set $d_i = d_k = D$;
if i is a pseudovertex then set $v_i = d_i + \bar{y}_i$ and place i on PQ-V;
SCAN(k, D).

S-UPDATE(CYC, k, D)
Set $\bar{y}_k = 0$;
for each $h \in$ CYC with $l_h = +$ set $\bar{y}_h = \bar{y}_h + (D - d_h)$;
for each $h \in$ CYC with $l_h = -$:
set $\bar{y}_h = \bar{y}_h - (D - d_h)$,
SCAN(h, D).

E-UPDATE(i, SA, TA, UA, D)
Split PQ-P(i) into $\{\text{PQ-P}(h)\}_{h \in SA \cup TA \cup UA}$;
for all $h \in SA \cup TA \cup UA$ set $e_h = e_i + \bar{y}_i$;
for all $h \in SA \cup TA$ set $d_h = D$;
for all $h \in SA$ SCAN(h, D);
for all $h \in TA$ if h is a pseudovertex then
set $v_h = d_h + \bar{y}_h$ and put h on PQ-V;
for all $h \in UA$:
if PQ(h) is not empty then:
set $v_h = v'_k + e_h$ and $p(h) = p'(k)$ where k is the top element on PQ-P(h), place h
on PQ-V if $v_h < \infty$.

A-UPDATE(D)
For each exterior vertex k with $l_k = +$ set $\bar{y}_k = \bar{y}_k + (D - d_k)$;
for each exterior vertex k with $l_k = -$ set $\bar{y}_k = \bar{y}_k - (D - d_k)$;
for each edge $(i, j) \in E$:
if $l_{b(i)} = +$ and $l_{b(j)} = 0$ then set $\tilde{c}_{ij} = d_{ij} - D + e_{b(j)}$,
if $l_{b(i)} = +$ and $l_{b(j)} = -$ then set $\tilde{c}_{ij} = d_{ij} - d_{b(j)} + e_{b(j)}$,
if $l_{b(i)} = l_{b(j)} = +$ then set $\tilde{c}_{ij} = 2d_{ij} - 2D$,

if $l_{b(i)} \neq +$ and $l_{b(j)} \neq +$ then:
set $\tilde{c}_{ij} = \tilde{c}_{ij} + e_{b(i)} + e_{b(j)}$,
if $l_{b(i)} = -$ then set $\tilde{c}_{ij} = \tilde{c}_{ij} + (D - d_{b(i)})$,
if $l_{b(j)} = -$ then set $\tilde{c}_{ij} = \tilde{c}_{ij} - (D - d_{b(j)})$.

We will not formally prove that Algorithm II-B correctly solves the matching problem. However, we should note that this result follows from the fact that MIN-I of Algorithm II-B computes the minimum over the same set of values considered by Algorithm II-A offset by the value of D.

We now give a bound on the running time of Algorithm II-B.

Theorem 3. Algorithm II-B requires at most $O(|V|\,|E| \log |V|)$ time.

To demonstrate this result, it suffices to show that no more than $O(|E| \log |V|)$ time is spent per iteration, since there can be no more than $\frac{1}{2}|V|$ iterations.

To obtain the required bound we need a better bound than $O(|V|)$ on the basic SHRINK operations. If the operations could be performed in $O(|\text{CYC}|)$ time then since a vertex can only be a member of CYC once per iteration and since there are at most $O(|V|)$ vertices and pseudovertices "handled" per iteration all shrink operations could be performed in $O(|V|)$ time. It is fairly easy to achieve this time bound on all operations except for updating the $b(i)$ values. However, if the new pseudovertex is given the "same name" as the larger of its interior pseudovertices then the $b(i)$ values need not be changed for real vertices in the largest pseudovertex in CYC. In this case whenever $b(i)$ changes the number of real vertices in the outermost pseudovertex containing i at least doubles. Consequently, for any real vertex i, $b(i)$ can change at most $\log |V|$ times and then the total time required per iteration for updating $b(i)$ values is $O(|V| \log |V|)$. For more details on the data structures required for the basic GROW, SHRINK, EXPAND, and AUGMENT operations see [3, 7, 12, 13].

We now consider the scanning and other update operations performed by SCAN, G-UPDATE, S-UPDATE and E-UPDATE. First note that a vertex is only scanned when it is first + labeled. Thus each edge can be considered at most twice by SCAN. When each edge is scanned, at most $O(\log |E|) = O(\log |V|)$ time is expended. Thus, in total, for all scan operations $O(|E| \log |V|)$ time is used per iteration. Each of G-UPDATE, S-UPDATE, and E-UPDATE never performs a particular operation on a particular vertex more than once. All operations either require constant time or, in the case of E-UPDATE, $\log |V|$ time per vertex. Thus in total $|V| \log |V|$ time is expended per iteration.

Now note that a vertex can be placed in PQ-V at most twice, once when it is labeled 0 and once when it is labeled $-$, and an edge can be placed on PQ-E at most once. Consequently, throughout an iteration MIN-I can remove a vertex from PQ-V at most twice and can remove an edge from PQ-E at most once. Thus in total MIN-I requires at most $O(|E| \log |V|)$ time per iteration.

Finally, note that A-UPDATE and I-UPDATE use no more than $O(|E|)$ time per iteration. We now have overall bound of $O(|V|\,|E| \log |V|)$. ∎

VI. SHORTEST AUGMENTING PATH INTERPRETATION

In Section III the blossom algorithms were introduced as primal-dual algorithms. Primal-dual methods have been shown to be key techniques for solving many graph

related problems of linear programming. A general discussion of primal-dual methods can be found in [15]. In Sections IV and V we showed how dual adjustments can be condensed to improve the performance of the algorithm.

In this section we now look at the blossom algorithms and the improved versions from a more combinatorial point of view. Our approach buids on the one given in [6], where such a combinatorially motivated algorithm based on characterization II is discussed.

An iteration of the blossom algorithm can be described as follows. At the beginning a matching M and a dual solution y are at hand, fulfilling the complementary slackness conditions with the additional property that all $S \in \mathcal{S}$ with nonzero dual value y_S are hypomatchable and $y_i = 0$ for all vertices $i \notin V(M)$. Moreover, all these sets $S \in \mathcal{S}$ with $y_S > 0$ are shrunk to pseudovertices. Starting from an unmatched vertex $r \notin V(M)$ an alternating tree is grown until an augmenting path P is detected. Then the matching M is changed to M' by reversing the role of matching and nonmatching edges of P and the dual solution is altered such that complementary slackness and the additional conditions from the beginning of the iteration are fulfilled again.

It is easy to show that for both matchings M and M' the following is true.

Lemma 1. Let M be the matching at the beginning of a blossom iteration. Then M is an optimal solution to the following matching problem:

MCM:

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}$$

such that

$$\sum_{j:(i,j)\in E} x_{ij} \begin{cases} \leqq 1, & \text{for } i \notin V(M), \\ = 1, & \text{for } i \in V(M), \end{cases}$$

$$x_{ij} \geqq 0, \quad \text{for integer } (i,j) \in E.$$

Proof. Add the set of blossm constraints I (II) and formulate the associate dual linear programs DI′ (DII′). Then the dual solution y which is at hand is also feasible for DI′ (DII′) and complementary slackness with M is fulfilled. ∎

Now let P be the augmenting path which is used to augment M to M' and let us define the *length* of P by

$$c(P) = \sum_{(i,j)\in P\setminus M} c_{ij} - \sum_{(i,j)\in M\cap P} c_{ij}. \tag{12}$$

Then the following relation holds:

$$\sum_{(i,j)\in M'} c_{ij} = \sum_{(i,j)\in M} c_{ij} + c(P). \tag{13}$$

Thus the cost of the matching is enlarged by the length of the path P.

Now the following is true.

Lemma 2. Let P' be an augmenting path with respect to M starting at vertex $r \notin V(M)$, then

$$c(P) \leqq c(P');$$

i.e., P is the shortest augmenting path with respect to M which starts at r.

Proof. Extend the argument given in the proof of the previous lemma. ■

Thus the blossom algorithm can also be viewed as a method which successively looks for *shortest augmenting paths* with respect to a given matching and a predetermined start vertex. Of course this is not a standard shortest path problem that can be solved by direct application of the well-known shortest path algorithms. Rather two additional conditions are imposed: (i) the arcs in the path must alternately be in M and not in M; (ii) the length of the path is not defined as the sum of the lengths of each of its arcs but rather by (12). Many of the difficulties associated with (ii) are overcome by working with a transformed set of costs. At the start of any iteration for each arc (i, j) we have the reduced cost c'_{ij} associated with the current dual solution, i.e.,

$$c'_{ij} = c_{ij} - y_i - y_j + \sum_{\substack{(i,j) \in \gamma(S) \\ S \in \mathcal{S}}} y_S$$

if characterization I is used and

$$c'_{ij} = c_{ij} - y_i - y_j - \sum_{\substack{(i,j) \in \delta(S) \\ S \in \mathcal{S}}} y_S$$

if characterization II is used.

Now let $G' = (V', E')$ be the graph which is at hand at the beginning of an iteration where all hypomatchable sets S with $y_S > 0$ are shrunk into pseudovertices. It can be seen easily that any augmenting path P' in G' uniquely determines an augmenting path P in G. With respect to the costs of these paths several useful relations hold.

Lemma 3. Let P be an augmenting path with respect to M in G which is induced by an augmenting path in G'. Then

$$c(P) = c'(P).$$

Proof. See [6].

This lemma motivates the use of the reduced cost c'_{ij} instead of the actual cost c_{ij} since because of the fact that $c'_{ij} = 0$ if $(i, j) \in M$, the cost definition (12) becomes a simple sum, i.e.,

$$c(P) = \sum_{(i,j) \in P} c'_{ij}. \tag{14}$$

The next lemma enables us to work in G' rather than G.

Lemma 4. Let P be an augmenting path in G which is induced by an augmenting path in G' and let $S \in \mathcal{S}$ with $y_S > 0$. By P_S we denote the partial path of P which is contained in S. Then $c'(P_S) = 0$.

Proof. The result follows immediately from the fact that the property

$$c'_{ij} = 0 \qquad \text{for all } (i,j) \in P_S \text{ with } y_S > 0$$

is maintained throughout MATCH. ∎

Thus far we have presented the foundation for working in G' and looking for a shortest augmenting path with respect to c'_{ij}. Yet the shortest augmenting path in G with respect to c_{ij} need not be induced by any augmenting path in G'. When this is not the case the approach developed thus far cannot produce the solution. Thus the matching algorithm must be able to detect whether this is the case and it must overcome this situation. The algorithm maintains throughout the path finding steps in G' control variables which indicate the possibility of the existence of shorter augmenting paths which cannot be generated from augmenting paths in G'. In such cases certain pseudovertices must be expanded so that these paths can be detected.

The following lemma is used to determine when such paths may exist.

Lemma 5. Let P be an augmenting path with respect to M in G which is not induced by an augmenting path in G', but is by one in G'', where G'' is obtained from G' by expanding the pseudovertex representing odd set S, then

$$c(P) = \begin{cases} c'(P) + y_S, & \text{for characterization I,} \\ c'(P) + 2y_S, & \text{for characterization II.} \end{cases}$$

Proof. The result follows by an immediate extension of the proof of Lemma 3. ∎

During each iteration Algorithm II-B can be interpreted as a combination of a modified Dijkstra algorithm which finds a shortest augmenting path in G' and a procedure which modifies G' when there is an indication that the shortest augmenting path in G is not induced by an augmenting path in G'. Let us at first assume that the shortest augmenting path in G is induced by an augmenting path in G'. Then we interpret the steps of Algorithm II-B as path finding techniques. Note that in this case Algorithm II-B would never call EXPAND (assuming no ties are encountered by MIN-I) and thus all e_k values encountered would be 0 so that these variables can be ignored. Note that the $\tilde{c}_{ij}$ equal the c'_{ij} at the beginning of the iteration, so these are the edge costs for the shortest path problem.

It can be shown that throughout the iteration D is a lower bound on the length of a shortest augmenting path and that d_k is the length of a shortest alternating path from r to k ending with a nonmatching edge for exterior vertices k with $l_k = -$, respectively, ending with a matching edge for exterior vertices k with $l_k = +$. Furthermore, d_{ij} is the length of the shortest alternating path from r to $b(j)$ ending with edge (i,j). We now show that the updates to D, d_k, and d_{ij} made by Algorithm II-B by GROW and SHRINK are in fact setting path lengths.

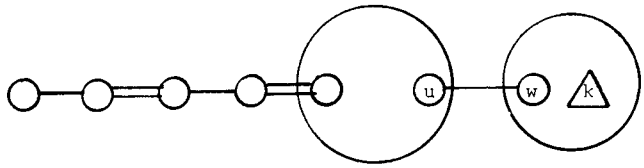


FIG. 8. Illustration for Case 1: $D = v_k = d_{b(u)} + \tilde{c}_{uw}$ is the length of the alternating path from r over $b(u)$ to k.

Case 1. Let $D = D_1$ with $v_k = \min\{v_j | j \text{ on PQ-V}\}$ and $l_k = 0$. Then $v_k = d_{uw}$ with $w \in \text{REAL}(k)$ and $l_{b(u)} = +$. (See Fig. 8.)

Now if vertex k is unmatched we have found an augmenting path of length D, i.e., a shortest augmenting path. If vertex k is matched with, say q, the algorithm sets

$$d_k = d_q = D$$

since the matching edge incident with k has (reduced) cost zero. When SCAN(q, D) is called it sets the d_{ij} for edges adjacent to q to the appropriate path lengths, i.e., $d_q + \tilde{c}_{ij}$ (note that in this case $D - d_q = 0$).

Case 2. $D = D_2$ with $D = d_{ij}$. (See Fig. 9.)

Adding edge (i, j) to the tree creates an odd cycle CYC. Let $q \in \text{CYC}$ with $l_q = -$. After adding edge (i, j) an even alternating path from q to r exists [which uses only edges in the tree and edge (i, j)]. The length L of this path is given by

$$L = d_{b(j)} + d_{b(i)} + \tilde{c}_{ij} - d_q = 2D - d_q.$$

Now let t be a 0-labeled vertex adjacent to a vertex s with $b(s) = q$. Then the alternating path from r to t over s has length

$$L + \tilde{c}_{st} = d_{st} = 2D - d_{b(s)} + \tilde{c}_{st}.$$

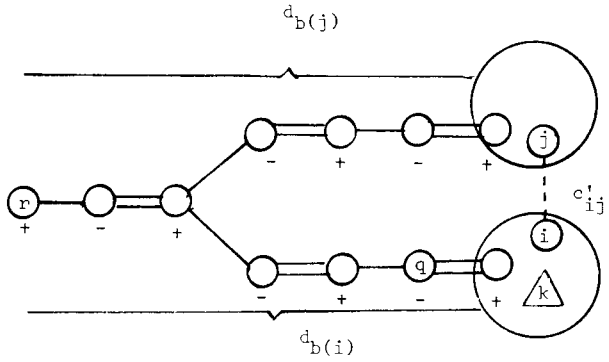


FIG. 9. Illustration for Case 2: $d_{b(j)}$ is the length of the alternating path from j to r, $d_{b(i)}$ is the length of the alternating path from i to r.

During a shrink SCAN(q, D) is called for all vertices q in CYC with $l_q = -$. When the edges (s, t) with $s \in \text{REAL}(q)$ are scanned d_{st} is set to the value just derived. Note that if before the shrink $b(s) = q$ and $l_q = +$ the algorithm need not update d_{st} since these path lengths do not change.

Thus far we have discussed the operation of the algorithm assuming the shortest augmenting path in G is induced by an augmenting path in G'. If MIN-I sets D to v_k for an exterior pseudovertex k with $l_k = -$ then this indicates that the shortest augmenting path in G may be induced by an augmenting path in the graph obtained by expanding k. We now interpret the updates performed by the algorithm in this case. These updates involve the e_k variables. The e_k can be interpreted as penalties for traversing paths that are interior to blossoms at the start of the iteration. These penalties are justified by Lemma 5.

Case 3. $D = D_1 = v_k$ with $l_k = -$. (See Fig. 10).

In EXPAND the pseudovertex k is expanded and in E-UPDATE some of the d_{ij} are redefined and some new d_i and d_{ij} are set. Recall that $D = d_k + \bar{y}_k$.

Denote the hypomatchable set represented by pseudovertex k by R. For any $i \in R$ for which $l_{b(i)}$ is set to + and any j adjacent to i with $l_{b(j)} = 0$, let P_t be the alternating path through the tree from r to j. By Lemma 5 we have

$$c(P_t) = c'(P_t) + 2\bar{y}_k.$$

This equation is equivalent to the relation

$$c(P_t) = d_k + \tilde{c}_{st} + 2\bar{y}_k.$$

Note that when E-UPDATE calls SCAN(s, D), the value of d_{ij} is set as follows:

$$\begin{aligned} d_{ij} &= D + \tilde{c}_{ij} + e_s + D - d_s \\ &= d_k + \bar{y}_k + \tilde{c}_{ij} + \bar{y}_k + 0, \end{aligned}$$

which equals the value derived above.

From this formula it becomes clear that even if the reduced cost $\tilde{c}_{st}$ equals zero this edge will not necessarily be a candidate for the next GROW operation. There may still be some other edges (u, v) incident to +-labeled vertices having $d_{uv} \in [D, d_{st})$ and

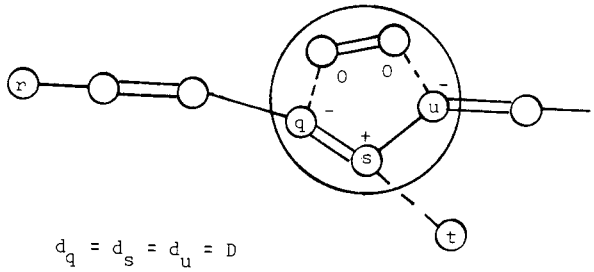


FIG. 10. Updating during expansion: $d_q = d_s = d_u = D$.

such edges could have $j \notin V(M)$, i.e., induce an augmenting path with length shorter than d_{ij}.

Thus if one is interested only in finding the shortest augmenting path this expansion may not be imperative at that particular stage. Yet, the algorithm which is motivated by a search for shortest *alternating* paths performs an expansion at this time because it is able to fix d_h values for h interior to k at the lower bound D (vertices q, s, and u in Fig. 10). However, as we have demonstrated above none of these paths can lead to *augmenting* paths of length less than $D + \bar{y}_k$.

Considering the fact that an expansion is very costly in all implementations one may wish to investigate the computational advantage of postponing expansions.

Based on the interpretation given in this section it directly follows that :

Theorem 4. Given a graph $G = (V, E)$, a matching $M \subseteq E$ with $|M| \leqq \frac{1}{2}|V| - 1$ and non-negative weights c_{ij} for $(i, j) \in E$, Algorithm II-B can easily be modified to find the shortest augmenting path starting at a vertex $r \in V \backslash V(M)$ in $O(|E| \log |V|)$ time, where the cost of a path P is defined by $\Sigma_{(i,j)\in P} c_{ij}$.

VII. POSTOPTIMALITY PROCEDURES

Postoptimality procedures are of practical importance since matching problems most naturally occur within a more complex system which requires that the problem be resolved on the same graph with a few changes (new edges/new edge weights). Then it can be advantageous to start from the old solution rather than to solve every modified problem from scratch.

The computerized system for mass transit crew scheduling [4] handles matching problems with a small number of side constraints. This problem is tackled via a Lagrangian Relaxation approach. The optimization routine requires the solution of a series of matching problems on the same graph with only a few number of edge weights changed and thus asks for an efficient postoptimality procedure.

Nemhauser and Weber [14] use matchings to solve a Lagrangian Relaxation for the set-partitioning problem. They use the blossom algorithm based on characterization I. For the sensitivity analysis Weber [17] gives a set of updating formulas distinguishing the different cases that may occur (weight is increased/decreased for a matching/nonmatching edge which is shrunk/not shrunk in the optimal solution). Cunningham and Marsh [5] also describe a procedure that can be used in the context of their primal matching algorithm.

The need for efficient postoptimality procedures in connection with the crew scheduling application motivated our further research. Below we demonstrate that sensitivity analysis can be performed very simply and efficiently within our general framework. No special updating formulas/subroutines are necessary. Rather, after some minor changes the general procedure MATCH can be reentered. Again this efficient and simple approach was developed within the framework of "shortest augmenting path" using combinatorial arguments (see [8]). Here we present the adaption of these ideas to our general matching algorithm.

At the end of all the matching implementations that we have described there is always a perfect matching M and a dual feasible solution y at hand which together fulfill the complementary slackness conditions. Some of the original vertices may be shrunk

into pseudovertices which represent those hypomatchable sets with nonzero dual values. Now let $r \in V$ be such that for one or more edges $(r, j) \in \delta(r)$ the edge weight c_{rj} has been modified to $\hat{c}_{rj}$.

If vertex r is shrunk into a pseudovertex we modify in a first step the optimal dual solution so that all hypomatchable sets containing r get zero dual value. After deleting the matching edge incident with vertex r from M we then adjust the edge weights for all edges in $\delta(r)$ and update the dual variable y_r so that again a dual feasible solution is at hand. Then we reenter our basic procedure for one further iteration, i.e., determination of a shortest augmenting path. At the end of this procedure $\hat{c}'_{ij} \geqq 0$ for $(r, j) \in \delta(r)$ and a perfect matching is again at hand. We now give a more detailed description of this routine.

The first step is necessary only if vertex r is shrunk into a pseudovertex, say v. Let $t \in V$ be the vertex matched with r. If vertex r is not shrunk we simply delete the matching edge (r, t) from the matching and start immediately with the second step. To accomplish the first step, we introduce two artificial vertices $a, b \notin V$ and two artificial edges (a, r) and (b, t) as illustrated in Figure 11.

Depending on what linear characterization has been used to solve the original matching problem we have to introduce appropriate edge weights and dual variables for the artificial vertices and edges. For characterization I we set

$$c_{ar} = y_r, \tag{15}$$

$$c_{bt} > \max \left\{ y_t, y_t + \tfrac{1}{2}\left(\sum_{\substack{r \in S, \\ t \notin S}} y_S - \sum_{r,t \in S} y_S \right) \right\}. \tag{16}$$

For characterization II we set

$$c_{ar} = y_r + \sum_{r \in S} y_S, \tag{17}$$

$$c_{bt} > y_j + \sum_{\substack{t \in S, \\ r \notin S}} y_S + \sum_{\substack{t \in S, \\ r \notin S}} y_S. \tag{18}$$

For both characterizations the additional dual variables can be set to zero, i.e., $y_a = y_b = 0$ and implicitly $y_S = 0$ for all $S \subseteq V \cup \{a, b\}$ odd with $S \cap \{a, b\} \neq \phi$.

It is easy to see that with this definition we obtain in both cases a feasible dual solu-

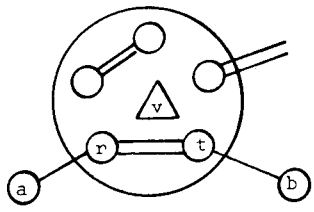


FIG. 11. Graph with artificial vertices and edges.

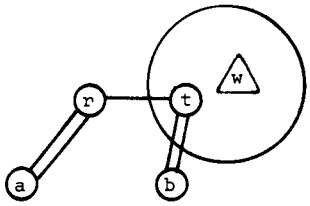


FIG. 12. Modified graph after application of MODDUAL(a).

tion which with the (nonperfect) matching still fulfills the complementarity conditions. After making this modifications we execute the following procedure which modifies the dual solution and makes vertex r an exterior vertex.

MODDUAL(a)
Set $l_a = +$ and $l_k = 0$ for all exterior vertices $k \neq r$;
I-UPDATE(r, D);
MIN-I(; i, D);
while i is matched
  if $l_i = 0$ then GROW($i, b(p(i)), D$),
  if $l_i = +$ then SHRINK($i, b(p(i)), D$),
  if $l_i = -$ then EXPAND(i, D),
  MIN-I(; i, D),
AUGMENT($i, b(p(i)), D$).

To see that r has become an exterior vertex in the sense of Section VI we view MODDUAL as a shortest augmenting path finding routine which is started from vertex a. The only existing augmenting path has to use edge (t, b). By (16) [(18)] we have made this edge expensive enough so that it can become part of the shortest path tree only after all pseudovertices containing r are expanded.

After executing MODDUAL delete the artificial vertices a and b and the artificial edges (a, r) and (t, b) and start with the second step. With the new edge weights $\hat{c}_{ij}$ we calculate

$$\Delta = \begin{cases} \min\limits_{j \in \delta(r)} \{\hat{c}_{rj} - y_r - y_j\}, & \text{for characterization I,} \\ \min\limits_{j \in \delta(r)} \left\{\hat{c}_{rj} - y_r - y_j - \sum\limits_{r,j \in S} y_S\right\}, & \text{for characterization II.} \end{cases}$$

If we now set $y_r = y_r + \Delta$ all edges (r, j) in $\delta(r)$ will have $\hat{c}'_{ij} \geqq 0$.

The following subroutine REOPT will then reoptimize, i.e., give the optimal matching with respect to the altered edge weights, by augmenting along the shortest augmenting path starting at vertex r.

REOPT(r)
Set $l_r = +$ and $l_k = 0$ for all exterior vertices $k \neq r$;
I-UPDATE(r, D);
MIN-I(; i, D);

while i is matched:
  if $l_i = 0$ then GROW$(i, b(p(i)), D)$,
  if $l_i = +$ then SHRINK$(i, b(p(i)), D)$,
  if $l_i = -$ then EXPAND(i, D),
  MIN-$I(\ ; i, D)$;
AUGMENT$(i, b(p(i)), D)$.

A comparison of the two subroutines MODDUAL(a) and REOPT(r) with the basic procedure MATCH(V, E, c) shows that the reoptimizing technique is identical to two iterations of the basic procedure without the first initialization of the dual variables and the graph structure. Thus in a computer implementation one would program MATCH in such a way that it can be entered with an appropriate graph structure and appropriate dual variables.

Now let $U \subseteq V$ be such that the edge weights are only altered for edges $(i, j) \in \delta(U) \cup \gamma(U)$, then the complexity to reoptimize equals $2|U|$ times the complexity of one iteration of the basic procedure. If the cardinality of U is small this compares favorably to solving the altered problem from scratch. We note that this is the same property achieved by the algorithm of Cunningham and Marsh [5] in the context of their primal algorithm.

This work was supported by Sonderforschungsbereich 21 (DFG), Institut für Operations Research, Universität Bonn, West Germany and by Contract MD-06-0041 from the Urban Mass Transportation Administration of the U.S. Department of Transportation. We wish to thank Bezalel Gavisch for bringing Ref. [12] to our attention.

References

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA (1974).
[2] M. O. Ball, A linear-time algorithm for finding a minimum-weight spanning tree in graphs in which all cycles pass through a single vertex. Unpublished.
[3] M. O. Ball and L. D. Bodin, A computational study comparing the efficiency of various list structures in matching algorithm. Unpublished.
[4] M. O. Ball, L. D. Bodin, and R. Dial, A matching based heuristic for scheduling mass transit crews and vehicles. *Transportation Sci.* 17 (1983) 4–31.
[5] W. H. Cunningham and A. Marsh, A primal algorithm for optimal matching. *Math. Programming Study* 8 (1976) 50–72.
[6] U. Derigs, A shortest augmenting path method for solving minimal perfect matching problems. *Networks* 11 (1981) 379–390.
[7] U. Derigs, Matching code theory part I: Combinatorial structures and the cardinality matching problem. Working Paper MS/S No. 81-041, College of Business and Management, University of Maryland at College Park (1981).
[8] U. Derigs, Shortest augmenting paths and sensitivity analysis for optimal matchings. Report 82222-OR, Institut für Ökonometrie und Operations Research, Universität Bonn (1982).
[9] U. Derigs and G. Kazakidis, On two methods for solving minimal perfect matching problems. In *Proceedings of the Second Danish/Polish Mathematical Programming Seminar,* J. Krarup and S. Walukiewicz, Eds. (1980), pp. 85–100.
[10] J. Edmonds, Maximum matching and a polyhedron with 0,1 vertices. *J. Res. Natl. Bur. Standards*, 69B (1965) 125–130.

[11] J. Edmonds and W. R. Pulleyblank, Facets of 1-matching polyhedra. In *Hypergraph Seminar*, Lecture Notes in Mathematics 411 (1974) 214–242.
[12] Z. Galil, S. Micali, and H. Gabow, Priority queries with variable priority and an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. Unpublished manuscript (1982).
[13] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York (1976), pp. 217–263.
[14] G. Nemhauser and G. Weber, Optimal set partitioning, matchings and Lagrangian duality. *Naval Res. Logist. Quart*. 26 (1979) 553–563.
[15] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ (1976).
[16] W. H. Pulleyblank, Faces of matching polyhedra. Thesis, University of Waterloo, 1973.
[17] G. Weber, Sensitivity analysis of optimal matchings. *Networks* 11 (1981) 41–56.

Received February 4, 1982
Accepted May 4, 1983