SIAM J. OPTIM.
Vol. 14, No. 3, pp. 894–913

NEW PRECONDITIONERS FOR KKT SYSTEMS OF NETWORK FLOW PROBLEMS*

A. FRANGIONI† AND C. GENTILE‡

Abstract. We propose a new set of preconditioners for the iterative solution, via a preconditioned conjugate gradient (PCG) method, of the KKT systems that must be solved at each iteration of an interior point (IP) algorithm for the solution of linear min cost flow (MCF) problems. These preconditioners are based on the idea of extracting a proper triangulated subgraph of the original graph which strictly contains a spanning tree. We define a new class of triangulated graphs, called *brother-connected trees* (BCTs), and discuss some fast heuristics for finding BCTs of "large" weight. Computational experience shows that the new preconditioners can complement tree preconditioners, outperforming them both in iterations count and in running time on some classes of graphs.

Key words. min cost flow problems, interior point algorithms, preconditioned conjugated gradient method, triangulated graphs

AMS subject classifications. 90C51, 65F10

DOI. 10.1137/S105262340240519X

1. Introduction.

The linear min cost flow (MCF) problem is the following linear program (LP):

$$\min\{cx : Ex = b,\ 0 \le x \le u\}, \tag{1.1}$$

where E is the node-arc incidence matrix of a directed graph $G = (N, A)$, c is the vector of arc costs, u is the vector of arc upper capacities, b is the vector of node deficits, and x is the vector of flows. This problem has a huge set of applications, either in itself or, more often, as a submodel of more complex and demanding problems [1]. This is evidenced by the enormous amount of research that has been devoted to developing efficient solution algorithms for MCF problems [1], either by specializing LP algorithms—such as the simplex method—to the network case, or by developing ad hoc approaches.

Recently, interior point (IP) methods for linear programming have established a reputation as efficient algorithms for large-scale problems; a detailed description of the IP algorithms and their underlying theory can be found in the extensive literature on the subject and in many recent linear programming textbooks, e.g., [19, 24]. At each iteration of these methods, linear systems of the form

$$\left(E\Theta E^T\right)\Delta y = d \tag{1.2}$$

have to be solved, where Θ and d are an $m \times m$ diagonal matrix ($m = |A|$) with positive entries and a vector of $\mathbb{R}^n$ ($n = |N|$), respectively, which depend on the current solution and on the IP algorithm chosen. These systems are often referred to as

*Received by the editors April 8, 2002; accepted for publication (in revised form) August 29, 2003; published electronically March 23, 2004. This work has been partially supported by project CNRG00A4E0 "Interior Point Methods for Structured Linear Programs" of program CNR-Agenzia2000 of the Italian National Research Council (CNR).

http://www.siam.org/journals/siopt/14-3/40519.html

†Dipartimento di Informatica, Università di Pisa, Via Buonarroti 2, 56125 Pisa, Italy (frangio@di.unipi.it).

‡Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti" del CNR, Viale Manzoni 30, 00185 Roma, Italy (gentile@iasi.rm.cnr.it).

KKT systems, because they represent the computational core of a "slackened" KKT system for the problem. Although the form (1.2) is not, strictly speaking, the most general, it has the advantage of being the same for many variants of IP algorithms. Furthermore, in the MCF case, the matrix $M = E\Theta E^T$ has close relationships with several extensively studied objects in both linear algebra and graph theory. When $\Theta = I$, the matrix M is closely related to the *Laplacian* of the *undirected* version of G [2, 7], which has been exploited to explore topological properties of graphs through the spectral properties of some associated matrices [6]. Conversely, the graph G can be thought of as a combinatorial representation of certain algebraic properties of M [20], some of which will be recalled below.

The solution of (1.2) typically represents by far the main computational burden of IP algorithms. Thus, developing a specialized approach for the solution of (1.2) for specially structured matrices E can substantially improve the performance of an IP method. Since the form of the KKT system is independent of the specific variant of IP algorithm used, the same specialized solver for (1.2) can be used to implement all the variants of IP algorithms.

As $M = E\Theta E^T$ is symmetric and positive semidefinite, (1.2) is often solved through a Cholesky factorization, which is computationally effective and numerically stable. That is, a lower triangular *Cholesky factor* L with all diagonal entries equal to 1 and a diagonal matrix D with a positive (nonnegative) diagonal are found such that $M = LDL^T$; this can be done in $O(n^3)$, and, once the factorization has been computed, systems involving M can be solved in $O(n^2)$ with two backsolves on L. However, a well-known drawback of the Cholesky factorization is the *fill-in* phenomenon: a sparse matrix M may have a dense Cholesky factor L. The density of the Cholesky factor may vary by reordering the rows of the matrix E; hence, IP codes usually make an effort at finding a permutation of the rows of E which (approximately) minimizes the fill-in effect. This is only done at the beginning of the algorithm, since the structure of the nonzeros in M (and therefore of its Cholesky factor) does not depend on Θ and therefore does not change with the iterations. The problem of finding the reordering which produces the least fill-in is known to be $\mathcal{NP}$-hard [25]; however, several effective heuristics have been developed for computing a "good" such permutation [19]. Yet, in general the fill-in phenomenon cannot be avoided [4] except in some specific cases, so that alternative methods have been proposed for MCF [17, 15, 13, 14, 23] and other network-structured problems [4]. Most of these methods solve the system using a preconditioned conjugate gradient (PCG) method. The critical choice is therefore that of the preconditioner: it must be inexpensive to compute and invert while delivering a consistent reduction of the number of conjugate gradient iterations required to (approximately) solve (1.2).

The first PCG-based IP algorithm specifically tailored for MCF problems was proposed in [17]. Following suggestions from [12] and [22], the *tree preconditioner* was defined, which is a preconditioner of the form

$$M_S = E_S \Theta_S E_S^T, \tag{1.3}$$

where S is a spanning tree of G, E_S is the node-arc incidence matrix of S, and Θ_S is the restriction of Θ to the arcs in S. In particular, S is chosen as an (approximate) maximum-weight spanning tree, the weight of each arc (i, j) being the corresponding θ_{ij}. Such a tree can be constructed in $O(m)$ with a variant of the classical Kruskal algorithm where arcs are only approximately sorted using a "bucket" data structure

with m buckets. The linear systems involving M_S can then be solved in $O(n)$, at each step of the PCG method, by considering the three linear systems with coefficient matrix E_S, Θ_S, and E_S^T, respectively; it is well known [1] that these systems can be solved by visiting the tree S. The preconditioner M_S can be expected to be spectrally effective, especially in the final iterations of an IP algorithm; in fact, the analysis of IP methods shows that, if the optimal solution of the underlying MCF is unique, the weights θ_{ij} tend to zero on all arcs, except on those corresponding to the basic optimal solution [19] that form a spanning tree; hence $M_S \approx E\Theta E^T$ in the last iterations of the IP method. The analysis in [10] and the experimental results show that the tree preconditioner in fact has good spectral properties in the final iterations of an IP algorithm even in the degenerate case. Finally, a different rationale for the choice of S as a maximum-weight spanning tree has been given in [7].

Unfortunately, tree preconditioners are less effective in the first iterations; this has suggested a hybrid preconditioning technique [17], where the diagonal preconditioner is used in the first iterations, and then some heuristic rules are used to switch to the tree preconditioner in a later stage. The implementation of this approach, refined with better stopping criteria [18] and a custom primal-infeasible/dual-feasible IP algorithm [15], has shown to be competitive with well-known combinatorial MCF codes.

In [13], the tree preconditioner is "extended" by using

$$M'_S = M_S + \rho\,\mathrm{diag}(M - M_S) \tag{1.4}$$

as the preconditioner, where $\mathrm{diag}(X)$ is the diagonal matrix having as the diagonal elements those of X. This has the advantage of incorporating information about all arcs, rather than about only those in S. The parameter ρ can be chosen according to the structure of the MCF problem at hand, with different values proposed in [13] for different classes of MCF problems. The relationships between M'_S and M_S, from the spectral viewpoint, have been analyzed in [10]. Finally, a different preconditioner has been proposed in [14] for the special case of transportation problems, based on an incomplete QR factorization of M, that has been reported as being more effective than the tree preconditioner for this particular class of MCF instances in the early iterations of the algorithm. For a more detailed description of these preconditioners and their relationships the interested reader is referred to [16] and [10].

Our aim is to improve the effectiveness of IP methods for MCF problems by designing new classes of preconditioners. The basic idea is that of extracting a proper subgraph $S = (N, A_S)$ of G ($A_S \subseteq A$) which contains—possibly strictly—a spanning tree, but such that the corresponding matrix M_S defined as in (1.3) can still be efficiently factored. We will refer to these preconditioners as *subgraph based*, and to S as the *support* of M_S. One way for ensuring efficient factorization is to select S as a *triangulated* (also known as *chordal*) graph [20], so that there exists an ordering of the nodes for which M_S has no fill-in. Other ideas can then be exploited for further improving the effectiveness of these preconditioners, yielding a large variety of preconditioners, some of which provide a better trade-off between the cost of finding S and factoring M_S and the cost of the PCG iterations.

The structure of the paper is the following: in section 2 we introduce and prove the properties of a large family of new preconditioners. In sections 3 and 4 the algorithmic issues related to this new family of preconditioners are discussed. In section 5 the results of a computational experience aimed at assessing the effectiveness of the new preconditioners are presented. Finally, conclusions are drawn in section 6.

2. Subgraph-based preconditioners. We propose to choose S as a *triangulated* graph [20], i.e., such that every cycle of length at least 4 has an edge joining two nonconsecutive vertices in the cycle. Such an edge is called a *chord*, whence the alternative name of *chordal* graphs. Since M_S has the same nonzero structure of the node-node adjacency matrix of S, there exists a "good" ordering of the nodes, i.e., an $n \times n$ permutation matrix P_n, such that the reordered matrix $P_n M_S P_n^T$ has a Cholesky factorization without fill-in. This is in fact a generalization of the result that is exploited when tree preconditioners are used: a P_n exists such that $P_n E_S$ is lower triangular. For the case of trees, P_n corresponds to any permutation $\mathcal{P}$ of the nodes such that if (i, j) is an arc of S with i father of j, then row j precedes row i in $\mathcal{P}$; these permutations include reverse depth-first visit and reverse breadth-first visit. Note that the definition of a father-son relationship implies that a root has been chosen for the spanning tree.

Thus, a natural way to generalize the tree preconditioner would be to choose S as a maximum-weight triangulated subgraph of G. Unfortunately, this does not appear to be an easy problem; although no conclusive evidence is known, the problem is conjectured in [11] to be $\mathcal{NP}$-hard.

However, choosing a maximum-weight triangulated subgraph of G, even if it were computationally feasible, would not necessarily be a good idea in this application. This is due to the fact that, as shown in section 5, for MCF problems the tree preconditioner is already very effective, and only a limited (although sizable) increase of the spectral efficiency of the method can be expected, especially in the last IP iterations. Thus, it is crucial that the extra cost of finding and factoring a "fatter" preconditioner M_S is kept low for the approach to have some chance of improving on the tree preconditioner. Indeed, the most efficient implementations of IP methods for MCF based on the tree preconditioner use an *approximate* algorithm for finding the maximum-weight spanning tree, even though the optimal tree could be found in (low) polynomial time.

Hence, a generalization of the tree preconditioner is sought for finding a large-weight triangulated graph with only slightly more effort than that required for finding an approximate maximum-weight spanning tree. We remark here that, for our application, finding the graph S is not enough; the "good" permutation $\mathcal{P}$ also has to be provided. This can always be done in linear time [21], but in general it is not free.

Not much along these lines has been done in the literature. In [11], a class of triangulated graphs, the k-windmills, is defined in the context of finding the "best" Markov network model of manageable size which "explains" some observed data, a problem that can be recast as that of finding a maximum-weight triangulated subgraph with "small" cliques of a given graph. An approximation algorithm with guaranteed performance is given for the maximum-weight k-windmill problem, but the algorithm requires the solution of an LP and a rounding operation and is therefore not feasible for our application.

A different way to achieve similar results has been proposed in the more general setting of *M-matrices*; the preconditioners are constructed by adding "a few" extra arcs to a spanning tree T, carefully balancing the extra cost of the incurred fill-in with the gain in the number of iterations [22, 3, 5]. This can be done, e.g., by splitting the node set into a small number k of disjoint components of size about n/k, each one spanned by a subtree of T, and then adding to S the arc with largest weight connecting any two of the components. The approach in [9] is similar although more involved and is mostly motivated by the need for finding a preconditioner that parallelizes well: since the graph is recursively subdivided into smaller graphs of roughly equal

size, the workload can be evenly divided among parallel processors. In both cases, a small number of components ensures a "limited" increase in the cost for factoring the preconditioner, given that fill-in is expected.

2.1. Brother-connected trees. We now define a new family of preconditioners of the form (1.3), based on the characterization of a new class of triangulated graphs, strictly containing spanning trees.

DEFINITION 2.1. *A subgraph $S=(N, A_S)$ of G is a* brother-connected tree *(BCT) if it is either a spanning tree $T = (N, A_T)$ of G or it contains a spanning tree T of G such that the subgraph $S' = (N, A_S \setminus A_T)$ obtained by removing all the arcs of T from S is formed of a certain number $k \geq 1$ of node-disjoint connected components $S'_1 = (N_1, A_1), \ldots, S'_k = (N_k, A_k)$ such that all the nodes in N_i are "brothers" (sons of the same node) in T, and each S'_i is a BCT.*

Definition 2.1 is inherently recursive and operational in nature; a BCT can be constructed by iteratively taking a family of BCTs (which may be ordinary trees) and joining all their nodes in a tree, where all the nodes of any one of the original BCTs are sons of the same node. Note that, conversely, it is *not* required that all the sons of the same node in T belong to the same connected component. In particular, the connected components can be composed by only one node; in this case, we consider the empty set of arcs to be a spanning tree (and, therefore, a BCT) for the component. In other words, the arc set A_S of a BCT S is the union of the arc sets of a family $\mathcal{T} = \{T_1, \ldots, T_q\}$ of arc-disjoint subtrees T_i of G. The family $\mathcal{T}$ itself has a tree structure, where a tree T_i is the son of a tree T_j in $\mathcal{T}$ if all the nodes in T_i are brothers in T_j.

Thus, an important characteristic of a BCT S is its *depth*, which is the depth of the associated tree $\mathcal{T}$, i.e., the number of times that the composition operation has to be applied, starting from an empty graph, in order to construct S. A BCT of depth 1 is an ordinary tree, a BCT of depth 2 contains a spanning tree T such that the removal of all the arcs in T leaves a forest, and so on. For example, consider the graph of Figure 2 in section 3: solid arcs define T, dashed arcs are the forest at the second level, and dotted arcs do not belong to the BCT. The BCT depicted on the left side of Figure 2 has a family $\mathcal{T} = \{T_1, T_2, T_3\}$, where $T_1 = T$ are the solid arcs, T_2 is composed of the dashed arcs linking nodes 1, 2, 3, 4, and 5, and T_3 is only the dashed arc $(6, 7)$. The tree structure of $\mathcal{T}$ is simply that T_1 is father of both T_2 and T_3; therefore, the depth of the BCT is 2.

It is easy to show that BCTs are triangulated graphs by induction on the depth. A BCT of depth 1 is a tree, hence a triangulated graph. When building a BCT of depth $k+1$ out of a number of disjoint BCTs of depth at most k, all newly created cycles have length 3. Thus, there exists a permutation $\mathcal{P}$ of the nodes that allows us to factor E_S without fill-in if S is a BCT. Something more, however, can be said: the "good" ordering is "embedded" in the description of the BCT in terms of the associated tree $\mathcal{T}$. Thus, if the description is—as in the case of the heuristics that we propose—available "for free," then the BCT immediately provides all the necessary information for factoring the associated preconditioner without fill-in. This is what we are going to prove in the following.

A well-known property of the Cholesky factorization is that, given a matrix M' with Cholesky factor L', any symmetric positive definite matrix

$$M = \begin{bmatrix} M' & m \\ m^T & \mu \end{bmatrix} \text{ has a Cholesky factor of the form } L = \begin{bmatrix} L' & 0 \\ l^T & 1 \end{bmatrix}.$$

Furthermore, the values in a row of the Cholesky factor L depend only on the values on the same row and on the previous ones. Therefore, if M' is a matrix with no fill-in, then M can have fill-in only in its final row.

In graph terms, the above operation corresponds to adding to the graph representing the nonzero structure of the matrix M a new node, possibly connected with all other nodes. Therefore, the following result easily follows.

LEMMA 2.2. *Consider any finite number $k \geq 1$ of node-disjoint triangulated graphs $G_i = (N_i, A_i)$ and their corresponding "good" orderings $\mathcal{P}_i$; the graph $G = (N, A)$ obtained as the union of all the graphs G_i plus a new node u linked by an arc to each node in each of the graphs G_i is triangulated, and the corresponding "good" ordering $\mathcal{P}$ is obtained by composing the permutations $\mathcal{P}_i$ in arbitrary order and placing the new node u as the last node in the ordering.*

Proof. Apply the above observations: M' corresponds to all the G_i, and the new row l in the Cholesky factor of M is dense (completely nonzero), but this corresponds to the fact that S has $n-1$ arcs more than S'; i.e., the row $[m^T \mu]$ is completely nonzero too. □

We can now prove the main result.

THEOREM 2.3. *Given a brother-connected tree S in G, its representation as a tree $\mathcal{T}$ allows us to compute a "good" ordering of the nodes of G (such that M_S has a Cholesky factor L with no fill-in).*

Proof. We will proceed by double nested induction: the first on the depth of S, the second on the number of nonterminal nodes in the tree T contained in the BCT.

The basic step of the (outer) induction corresponds to depth 1; i.e., S is a tree. As we already recalled, any ordering of the nodes such that node j precedes node i if (i, j) is an arc of S and i is the father of j has the desired property. This ordering can be found in linear time.

For the inductive step, we assume that the ordering is available for any BCT with depth at most h and show how to construct it for BCTs of depth $h+1$. Once again we proceed by induction, this time on the number of nonterminal nodes of the spanning tree T included in S.

The basic step of the (inner) induction corresponds to the case where there is only one nonterminal node u; i.e., T is a "star tree," where any other node but u is a leaf. Since S is a BCT, the subgraph S' obtained by removing u (and all its incident arcs) from S is formed of $k \geq 1$ node-disjoint BCTs *of depth at most h*. Therefore, for the (outer) inductive hypothesis we know a good ordering for S', and we can find the one for S as shown in Lemma 2.2.

For the (inner) inductive step, consider a nonterminal node u such that all its sons are leaves of T; call V the set of the sons of u. Let S' be the subgraph of S induced by the nodes $V \cup \{u\}$ and let S'' be the subgraph of S induced by the nodes in $N \setminus V$; note that both subgraphs contain node u. Now we can apply the (inner) inductive hypothesis to S'', as we have reduced the number of nonterminal nodes by one unit; hence, we can find a good ordering $\mathcal{P}''$ of $N \setminus V$. Furthermore, since S is a BCT, then $S' \setminus \{u\}$ is a set of node-disjoint BCTs of depth at most h, and, as in the basic step of the (inner) induction, we can find a good ordering $\mathcal{P}'$ of $V \cup \{u\}$, where u must be the last node. We can then construct an ordering $\mathcal{P}$ of N by simply joining $\mathcal{P}'$ and $\mathcal{P}''$ in a sequence that corresponds to $\mathcal{P}'$ on V and to $\mathcal{P}''$ on $N \setminus V$ such that all the nodes in V precede those in $N \setminus V$. Therefore, the corresponding reordered M_S can be written as

$$M_S = \left[\begin{array}{c|c|c|c} M_{S'\setminus\{u\}} & 0 & m & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline m^T & 0 & \mu & 0 \\ \hline 0 & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{c|c|c|c} 0 & 0 & 0 & 0 \\ \hline 0 & & & \\ \hline 0 & & M_{S''} & \\ \hline 0 & & & \end{array}\right],$$

where $[m^T, \mu]$ is the (completely nonzero) row corresponding to node u in the matrix $M_{S'}$. The two matrices in the right-hand side share only one nonzero position in the row and column associated with u. Hence, the first part of the Cholesky factor L of M_S is equal to that of $M_{S'}$, and so it is the part of the factorization relative to the nondiagonal elements in row u. Thus, the Cholesky factorization of (the reordered) M_S in the first $|V|$ rows/columns has no fill-in. As S' is a graph, the associated matrix $M_{S'}$ is rank deficient and the value d'_u for its LDL^T factorization is zero. Hence, the value d_u for the factorization of M_S is equal to d''_u computed in the factorization of $M_{S''}$. Therefore, the second part of the factorization of M_S is exactly the factorization of $M_{S''}$. For the inductive hypothesis we know that the Cholesky factor of $M_{S''}$ (reordered with $\mathcal{P}''$) has no fill-in, and this finally allows us to conclude that $\mathcal{P}$ is a good ordering for M_S. □

The above result can be easily generalized with the following proposition.

PROPOSITION 2.4. *Let M be a positive definite matrix with a BCT support; then, the ordering of Theorem* 2.3 *is "good" for M.*

Proof. In the general case, the computation of the element d_u in the proof of Theorem 2.3 may depend on the submatrix associated with S'. Let d'_u and d''_u be the values computed in the factorization of $M_{S'}$ and $M_{S''}$, respectively. The first part of the factorization of M_S is equal to the factorization of $M_{S'}$ for rows in $S' \setminus \{u\}$, and the rest can be obtained as the factorization of $M_{S''}$, but starting from the value $d_u = d'_u + d''_u$. Therefore, M can be factored without fill-in. □

To summarize, for a BCT with depth 2, $\mathcal{P}$ must be such that

- The matrix E_T associated with its spanning tree T is lower triangular; i.e., if (i, j) is an arc of T with i father of j, then row j of E_S precedes row i in $\mathcal{P}$;
- For each nonterminal node u of T, each subset of its sons which belong to the same subtree $T_h = S'_h$ (once the arcs of T have been removed) is ordered in the permutation according to the order defined by T_h; i.e., if (i, j) is an arc of T_h with i father of j, then row j precedes row i in $\mathcal{P}$. The roots of the subtrees and the sequence of the trees can be arbitrarily chosen.

In general, $\mathcal{P}$ can be recursively constructed by ordering the nodes of the BCTs of depth h and then composing these orders into orders for the BCTs of depth $h+1$. This can be done with a bottom-up postvisit of the tree $\mathcal{T}$ associated with the BCT, i.e., by visiting the tree $\mathcal{T}$ from the leaves to the root with the constraint that each node of $\mathcal{T}$ can be visited only after all of its sons.

The induction process in Theorem 2.3 suggests an algorithm that performs the Cholesky factorization of M_S without fill-in in $O(nh^2)$, where h is the depth of the BCT. All the trees at the same depth q can be represented with a unique predecessor function $Pred[q]$ defined on the nodes, such that $Pred[q][u] = v$ if v is the father of u at depth q, and $Pred[q][u] = nil$ (null value) if u is a root, i.e., it has no father. For instance, in a BCT of depth 2 the function $Pred[1]$ represents the spanning tree T whose removal leaves a forest F, which is represented by the function $Pred[2]$. The algorithm for computing the LDL^T factorization of a matrix M_S with a BCT support S is shown with the pseudocode in Figure 1. It requires a description of the BCT (of depth h) in terms of the predecessor functions $Pred[q]$, and a description

```
Procedure CholeskyFactorBCT(h, n, M, Pred, Order, L, D)
  begin
    for i = j, ..., n; i = 1, ..., j − 1 do
      L[i, j] = M[i, j];
    for i = 1 ... n do
      D[i] = M[i, i];
      L[i, i] = 1;
    for i = 1 ... n − 1 do
      u = Order[i];
      for q = h ... 1 do
        w = Pred[q][u];
        if w ≠ nil then
          L[w, u] = L[w, u]/D[u];
          D[w] = D[w] − L[w, u]^2 ∗ D[u];
      for q = h ... 1 do
        w = Pred[q][u];
        if w ≠ nil then
          for r = q − 1 ... 1 do
            y = Pred[r][w];
            if y ≠ nil then
              L[y, w] = L[y, w] − L[w, u] ∗ L[y, u] ∗ D[u];
  end.
```

FIG. 1. *Pseudocode for factorization of a matrix with BCT support.*

of a "good" ordering $\mathcal{P}$ in an array $Order[]$. By performing a bottom-up visit of the tree $\mathcal{T}$, it outputs the Cholesky factor L and the diagonal matrix D. The algorithm is similar to the usual procedure for the Cholesky factorization, but it exploits the fact that the fill-in cannot be produced, so nonzero elements of L correspond to pairs (y, w) such that $y = Pred[s][w]$ for some level s. Indeed, the Cholesky factorization using the ordering $\mathcal{P}$ would be

$$L[y,w] = \frac{1}{D[w]}\left[M[y,w] - \sum_{u <_{\mathcal{P}} w} L[w,u]L[y,u]D[u]\right],$$
$$D[w] = M[w,w] - \sum_{u <_{\mathcal{P}} w} L[w,u]^2 D[u],$$

where "$<_{\mathcal{P}}$" is the ordering contained in $Order[]$.

Using the same data structures, $Pred$ and $Order$, an $O(nh)$ algorithm that solves systems of the form $M_S r = v$—which is what is actually required if M_S is used as a preconditioner in a PCG algorithm—can be constructed; any iteration of a PCG algorithm which uses a BCT-based preconditioner has a complexity of $O(nh + m)$. In our implementation, we have considered only the case of BCTs of depth 2; this simplifies and streamlines the algorithms, while still leaving room for almost doubling the number of arcs to be put in S with respect to a standard tree preconditioner (a BCT of depth 2 can have up to $2n - 3$ arcs).

Thus, BCT preconditioners can be easily integrated with standard tree preconditioners, and they do not need general-purpose Cholesky factorization routines; in fact, the construction and factorization of the preconditioner are easily and efficiently per-

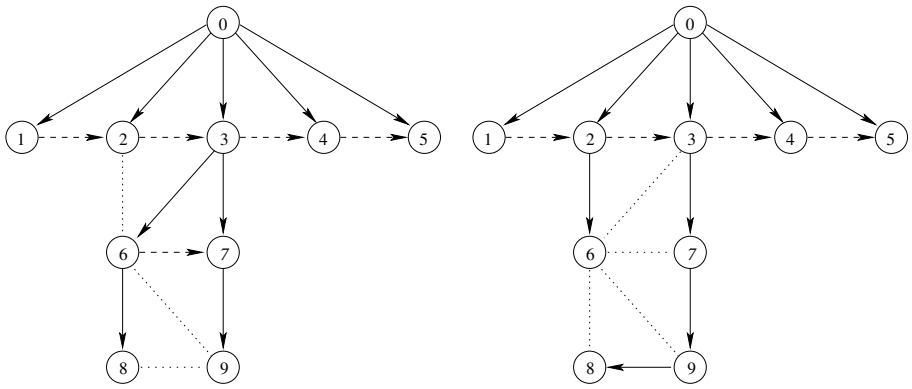

FIG. 2. *Two maximal BCTs on the same graph with different cardinality.*

formed using the data structures naturally produced by the construction of the BCT.

3. Finding brother-connected trees. The complexity of the problem of finding the maximum-weight BCT in a given graph G is not known to us. However, the exact solution of this problem is not crucial in this application; even in the case of tree preconditioners, an approximate solution is usually preferred although the exact solution can be obtained in low polynomial time. It is very unlikely that a maximum-weight BCT can be found with a comparable efficiency, because BCTs are not matroids. This is shown by the two BCTs S_1 and S_2 in Figure 2, where the solid arcs belong to the first level, the dashed ones belong to the second level, and, finally, the dotted ones are in the complements $G \setminus S_1$ and $G \setminus S_2$. It is easy to check that S_1 and S_2 are maximal BCTs (of level two) with different cardinality.

3.1. A class of heuristics for maximum-weight BCT. For all the above reasons, we will resort to heuristics to find the BCT to be applied in the PCG method. A large number of different heuristics can be proposed, by combining different variants of two basic ingredients:

(i) how a spanning tree T is chosen;
(ii) how extra arcs forming trees among brothers in T are chosen.

Some results can be proved about the worst-case performances of this kind of heuristics if T is chosen to be a maximum-weight spanning tree for the graph.

PROPOSITION 3.1. *Let G be a graph; denote by $w(MBCT)$ the weight of the maximum-weight BCT with depth 2 on G and by $w(MST)$ the weight of the maximum weight spanning tree on G. Then $w(MBCT) \leq 2w(MST)$.*

Proof. Consider the following problem: given a graph G, find a connected subgraph $S = (N, A_S)$ of maximum weight with the property that S contains at least a spanning tree $T = (N, A_T)$ of G such that the residual graph $S \setminus T = (N, A_S \setminus A_T)$ is acyclic. Obviously, this problem is a relaxation of the maximum-weight BCT with depth 2. Moreover, its optimal objective function value is less than $2w(MST)$: in fact, $w(MST)$ is an upper bound on both $w(T)$ and $w(S \setminus T)$ as the latter one is acyclic. □

COROLLARY 3.2. *All heuristics for constructing a BCT which augment the maximum-weight spanning tree are 2-approximated.*

Thus, choosing the initial tree as an (approximate) maximum-weight spanning tree appears to be a promising choice. In fact, we have experimented with several

different ways for finding an initial spanning tree, described in [8] and not reported here to save on space, but they were almost invariably outperformed by the "standard" maximum-weight spanning tree.

The above bound is asymptotically tight even if we find the maximum-weight spanning tree T on G and then compute a maximum-weight spanning tree on each connected component induced by the sets of brothers in T, as the following example shows.

Example 3.3. Let us consider the graph with n nodes and the following two types of arcs:

- $(i, i+1)$ for $i = 1, \ldots, n-1$ with weight 1;
- $(1, j)$ for $j = 3, \ldots, n$ with weight $1 - \epsilon$.

Clearly, the maximum spanning tree T is the path from 1 to n composed of the arcs of the first type; hence, there are no brothers in T and the heuristic stops. However, the whole graph is a BCT of depth two, with the arcs connecting 1 with each of the other nodes in the first level and the other arcs in the second level with total weight $2n - 3 - (n-2)\epsilon$.

3.2. Enlarging the tree to a BCT. When a tree T is selected, extra arcs forming trees among brothers in T must be chosen (point (ii)). For this task we propose three different variants:

(ii.a) When the tree is selected, the final ordering of the nodes to be considered in the factorization is also arbitrarily fixed as any "good" ordering for T. Then, the arcs out of T are scanned in (approximated) order of decreasing weight and added to the tree if they are compatible with the fixed ordering and they satisfy the condition that the trees on the second level are paths among brothers.

(ii.b) As in case (ii.a), the arcs are scanned in (approximated) order, and the trees in the second level of the BCT are restricted to being paths; however, the ordering between brothers can be changed. The final ordering is found by considering one of the two possible permutations for each path among brother nodes, and then composing these orders, respecting the tree structure of T.

(ii.c) This variant is analogous to case (ii.b), but the trees in the second level of the BCT are not restricted to being paths.

These three variants require different data structures and amounts of computational time (how many times the list of arcs is scanned), and they find different BCTs. Variant (ii.a) is the cheapest one, but it usually adds fewer new arcs. Variant (ii.c) is the most complex, as it requires a new union-find structure to find trees in the second level and to select the corresponding orders, but it may add more arcs. Variant (ii.b) is something in between.

For variants (ii.b) and (ii.c), it is actually possible to modify the original spanning tree T as the algorithm proceeds, in order to add even more arcs. One way to do that is to apply an operation, which we call *promotion*, whereby a node connected with its grandfather is "promoted" as a brother of its former father. That is, let j be a node, k its father in T, and i the father of k in T. If the arc (i, j) is selected from the (approximated) ordering, it is possible, under some conditions, to modify the tree T in such a way that j becomes a son of i and brother of k. This is done by making (i, j) an arc of T (i.e., in the first level of the BCT), while (k, j) becomes an arc of the second level, as shown in Figure 3. In order to apply the promotion, node j must not have incident arcs (j, l) in the second level of the BCT, as after the promotion j and l are no longer brothers. Moreover, for variant (ii.b) the node k must also have at most one connected brother in the second level of the BCT, for otherwise

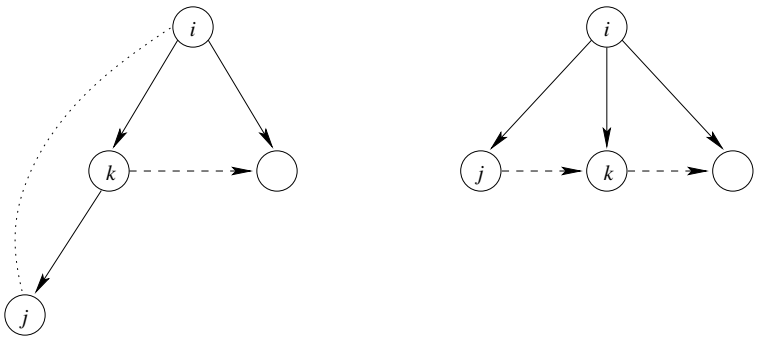

FIG. 3. *The BCT before (left) and after (right) the promotion.*

the tree in the second level would no longer be a path. Note that using the promotion operation in Example 3.3 allows one to discover that the complete graph is indeed a BCT.

In all the above heuristics, an initial ordering of the nodes is assumed that is "good" for the initial tree T; this is done by selecting a root node and performing a visit of the tree. Since this order impacts the heuristics (especially (ii.a), where it is fixed), the selection of the root node is potentially critical. We considered two possible strategies for selecting the root node: choosing the node with the largest adjacency list ("static") or choosing the node with the largest total weight of the set of incident arcs ("dynamic").

Let us remark here that the matrix $M = E\Theta E^T$ has rank equal to n minus the number of connected components in G, i.e., at most $n-1$. It is always possible to assume that G is connected, as otherwise the original MCF problem can be partitioned into a set of smaller subproblems, one for each of the connected components; hence, we can assume that the rank of M is $n-1$. When solving the KKT system, it is therefore possible to work with full-rank matrices by just deleting one row of E; alternatively, it is possible to work with the rank-deficient KKT system, although in this case M_S is rank deficient, too. The choice of the row (node) to be eliminated is arbitrary, yet it may have some consequences; when a node (row of the matrix E) is deleted, we choose it as the one associated with the root node of the tree T, although in principle different choices would be possible.

4. Further improvements. All the preconditioners that we have proposed so far can be further improved by applying two kinds of operations that attempt to incorporate in M_S information regarding arcs which have been left out of the support S.

The first operation amounts to adding to S all arcs (i, j) which are "parallel" to arcs already belonging to S, i.e., every other arc (i, j) or (j, i) belonging to G; we will denote these as "tree/BCT+parallel," or "T/BCT+P" for short, preconditioners. Clearly, this cannot generate fill-in other than that already present in the original M_S, as the support of the two matrices is the same. Note that "parallel" arcs, i.e., multiple copies of the same arc (or of its reverse arc) with different costs and capacities, are often present in MCF problems, e.g., to model piecewise linear convex separable flow cost functions [1]. This kind of operation has not been explicitly described before in the literature of IP approaches for MCF problems, while it is taken for granted when *M-matrices* are approached; our computational experience shows that the option has to be kept open. In fact, when "many" parallel arcs are present, it is useful to set the

weight of each edge $\{i, j\}$ in the MST computation equal to the sum of the weights of all parallel arcs (i, j) or (j, i), in order to correctly estimate the importance of adding any of those parallel arcs (and, therefore, all the others) to the support. However, when "few" parallel arcs are present, the extra computational burden required for computing the weights of the edges is not worth the corresponding improvement in the PCG convergence.

The second operation, proposed in [13] for the tree preconditioner, consists of using as preconditioner the matrix

$$M'_S = M_S + \rho \operatorname{diag}(M - M_S).$$

This cannot have more fill-in than M_S, and it contains at least some information about all arcs. We will denote these as "tree/BCT+diagonal," or "T/BCT+D" for short, preconditioners. Of course, combining the two ideas gives "T/BCT+P+D" preconditioners.

Adding the diagonal can be very useful for some classes of instances, but, as reported in [10] and essentially confirmed by our experience, it is not always convenient, so the option has to be kept open. Note that this operation adds some complexity to the Cholesky factorization of the preconditioner. This is more clearly seen in the case of T+D preconditioners; while the pure tree preconditioner basically does not need any factorization (it can be factored by just permuting the rows), the T+D preconditioner does need a true—although simple—factorization phase. Analogously, the factorization of BCT+D preconditioners requires the modification described by Proposition 2.4. It may also be worth remarking that the factorization routine can be somewhat simplified if $\rho = 1$, which is significant in light of the results reported in the next section.

5. A computational comparison of preconditioners. In this section, we present the results of a large-scale computational test aimed at assessing the effectiveness of our new family of preconditioners.

For our tests, we selected three well-known random generators of MCF problems: `goto` (GridOnTOrus), `gridgen`, and `netgen`.[1] For each generator, we generated a total of 12 classes of instances, with $n = 2^k$ for $k = 8, 10, 12, 14$, and 16 and up to three different densities. In particular, for $k = 8$ we generated instances with density 8, 16, and 32, for $k = 10$ we generated instances with density 8, 32, and 64, for $k = 12$ we generated instances with density 8, 64, and 256, for $k = 14$ we generated instances with density 8 and 64, and for $k = 16$ we generated only instances with density 8. In the following, we will use the form `genX.Y` to refer to the class of instances generated by the generator `gen` (`goto`, `grid`, or `net`), with $k =$ `X` and density equal to `Y`. In each class, five different instances were generated by simply changing the seed of the pseudorandom number generator.

For all the above instances, we ran an implementation of a primal-dual IP method, using a standard tree preconditioner, in order to collect the data for reproducing the matrices M at the IP iterations. Then, the different preconditioners were tested on these matrices, and an estimate of the total time that would be spent by an IP method if using each preconditioner is computed. This way, we ensure that for every preconditioner we solve exactly the same sequence of linear systems; since the systems are

[1] Source code for these generators can be downloaded, e.g., at http://www.di.unipi.it/di/groups/optimize/Data/MMCF.html; parameters for reproducing the instances are available upon request from the authors.

only approximately solved, the sequence of systems solved by different preconditioners within an IP approach would in general be different, so that directly comparing the total time spent in the solution of the linear systems during an IP method using each given preconditioner would have been unfair. The impact of the choice of the preconditioner on the overall optimization process will be analyzed in depth in a forthcoming paper, also taking into account many important details such as the different choices of IP algorithm (primal, dual, primal-dual) with several variants each (affine, barrier, predictor-corrector, etc.), and the required precision in the solution of the systems.

We remark that each one of the preconditioner procedures has been carefully implemented. In particular, during the factorization phase we have exploited as much as possible the structure of the preconditioner in order to speed up operations. For instance, T/T+P preconditioners have a Cholesky factor with entries in $\{1, -1, 0\}$ and where the entries in matrix D depend only on the predecessor arcs of each node; these matrices need not be directly constructed as such, so that both the factorization phase and the solution of the linear systems at each PCG iteration are faster in this case. Analogously, for BCT/BCT+P preconditioners the entries in the Cholesky factor depend only on the brothers and on the predecessor, but they do not depend on the sons in the first level of the BCT, which leads to some simplification in the factorization routine. Since the efficiency of these procedures is crucial, all efforts have been made to obtain the best possible implementation for all the tested preconditioners.

We also remark that we have used an adaptive stopping rule for the PCG: the algorithm for solving (1.2) is stopped when a vector Δy is found such that

$$|d_i - M_i \Delta y| \leq \varepsilon_1 \max(|b_i - E_i \bar{x}|, \varepsilon_2 \max(|b_i|, 1))$$

for all components i, where $\bar{x}$ is the current primal solution of the IP algorithm. It is easy to check that this stopping rule allows early termination in the initial IP iterations, thereby improving the overall efficiency of the IP approach, by ensuring that the PCG is stopped as soon as the system is solved with enough precision to decrease the infeasibility of the primal solution, if it is not feasible yet, or that the violation of the primal constraints is not worsened too much, if the primal solution is already feasible (usually, because of cancellation of errors this is enough to keep the primal solution feasible until termination). The tolerance ε_1 is set to 0.1, while ε_2 is the relative feasibility precision required to the constraints satisfaction, typically set to `1e-6`. We have also tested the alternative "cosine" stopping rule proposed in the literature [18], but we have found it to be less reliable from the IP viewpoint; this is probably due to the fact that we have used a standard primal-dual IP algorithm rather than a primal-infeasible/dual-feasible one [15].

The computational experiments were performed in three phases. In the preliminary phase, a significant subset of the instances were tested with *all* the over 200 possible variants of preconditioners obtained by implementing the ideas presented in sections 3 and 4 and in [8]. This allowed us to discover that certain choices were consistently outperformed, thus reducing the set of promising preconditioners to only eight. In the second phase these preconditioners were tested on the full set of instances, in order to develop automatic rules for choosing the right preconditioner for each instance. Finally, in the third phase we compared the performances of the code having the automatic preconditioner selection rule with that of the corresponding T/T+D (whichever of the two was better) preconditioner. We will report the results of the three phases separately.

5.1. Preliminary experiments. In the preliminary phase, we were able to establish with a high degree of confidence the following facts:

- As already mentioned, using approximated maximum-weight spanning trees as the basis for the heuristics is consistently the best choice.
- The T+D and BCT+D preconditioners were found to be preferable to their "pure" counterparts for the `grid` and `net` classes, while the converse happens for the `goto` problems (except in the very first iteration when $\Theta = I_m$); this basically confirms the results reported in [10].
- When a "+D" preconditioner is used, $\rho = 1$ seems to be the best option in general, at least for the classes of instances at hand.
- Working with the full rank-deficient system M is consistently better than eliminating one row when a "+D" preconditioner is used (this is reasonable, since then the preconditioner is nonsingular even if the whole system is not), while eliminating the row and working with a nonsingular system is preferable if the diagonal is not added.
- When one row (node) has to be removed from the system, the best choice appears to be the one with the largest total weight of the set of incident arcs.
- When working with the rank-deficient system, the choice of the root node—which impacts the heuristics for the maximum-weight BCT computation—has little effect.

We are not reporting the tables relative to the experiments in the preliminary phase in order to save space.

At the end of the preliminary phase, we were therefore able to decide that all preconditioners should find the initial tree with an approximated maximum-weight spanning tree computation. Furthermore, for `goto` problems we did not use the "+D" preconditioners, and therefore we eliminated one row and worked with the full-rank subsystem, while for `grid` and `net` problems we did use the "+D" preconditioners, therefore working with the rank-deficient system M. The remaining choices were about which heuristic was used for finding the BCT ((ii.a), (ii.b), (ii.c), or none, i.e., the tree preconditioner) and whether or not "+P" preconditioners are used, for a grand total of eight different variants. For those we ran the code on all the instances, obtaining the results reported in the next section.

5.2. The second phase. The complete results of the second phase are shown in Table 5.1. There are seven groups of two columns. The first three, labeled *B-a*, *B-b*, and *B-c*, report the results relative to BCT preconditioners where the BCT is found with heuristic (ii.a), (ii.b), and (ii.c), respectively. The fourth group, labeled *TP*, reports the results relative to the T+P preconditioner. Finally, the last three groups, labeled *BP-a*, *BP-b*, and *BP-c*, report the results relative to BCT+P preconditioners. For `grid` and `net` problems only, these preconditioners have to be intended as "+D" also. All the results in the tables are normalized with respect to those obtained by the tree preconditioner (without "+P", and with or without "+D" according to the problem class); that is, the numbers in the columns *Iter* and *Time* are, respectively,

$$Iter = \frac{\text{number of iterations of the corresponding preconditioner}}{\text{number of iterations of the tree preconditioner}}$$

and

$$Time = \frac{\text{running time of the corresponding preconditioner}}{\text{running time of the tree preconditioner}}$$

(averaged among the five instances of each class). This makes it easier to spot where

TABLE 5.1
Comparison of the most promising preconditioners.

	$B-a$		$B-b$		$B-c$		TP		$BP-a$		$BP-b$		$BP-c$	
goto	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time
8.8	0.97	*	0.87	*	0.87	*	0.90	*	0.88	*	0.80	*	0.79	*
8.16	0.97	*	0.84	*	0.84	*	0.82	*	0.78	*	0.69	*	0.68	*
8.32	0.96	*	0.82	*	0.82	*	0.78	*	0.75	*	0.64	*	0.63	*
10.8	0.99	*	0.86	*	0.85	*	0.77	*	0.76	*	0.70	*	0.70	*
10.32	0.97	0.98	0.81	0.85	0.80	0.86	0.77	0.78	0.74	0.77	0.62	0.68	0.62	0.67
10.64	0.98	1.00	0.86	0.90	0.84	0.88	0.74	0.75	0.69	0.71	0.62	0.66	0.62	0.65
12.8	0.99	1.01	0.86	0.93	0.86	0.92	0.84	0.84	0.82	0.86	0.79	0.85	0.78	0.84
12.64	0.98	0.99	0.84	0.88	0.84	0.87	0.73	0.67	0.71	0.67	0.64	0.62	0.64	0.61
12.256	0.97	0.97	0.80	0.84	0.79	0.83	0.72	0.71	0.68	0.71	0.50	0.56	0.50	0.56
14.8	0.99	1.00	0.76	0.83	0.76	0.83	0.33	0.37	0.33	0.37	0.30	0.39	0.30	0.38
14.64	0.98	1.00	0.78	0.83	0.78	0.83	0.63	0.64	0.62	0.65	0.53	0.62	0.53	0.59
16.8	1.01	1.01	0.72	0.74	0.71	0.74	0.22	0.24	0.22	0.24	0.19	0.22	0.19	0.22
grid	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time
8.8	1.00	*	0.99	*	0.99	*	0.87	*	0.87	*	0.87	*	0.87	*
8.16	0.99	*	0.99	*	0.98	*	0.97	*	0.97	*	0.95	*	0.95	*
8.32	0.99	*	1.00	*	1.00	*	0.97	*	0.96	*	0.97	*	0.97	*
10.8	1.00	*	1.00	*	1.00	*	0.82	*	0.82	*	0.82	*	0.82	*
10.32	0.99	1.07	0.98	1.12	0.98	1.11	0.94	0.96	0.94	1.02	0.94	1.12	0.94	1.13
10.64	1.00	1.12	0.98	1.28	0.98	1.31	0.99	0.99	0.98	1.09	0.98	1.24	1.00	1.29
12.8	1.00	1.05	1.00	1.08	1.00	1.09	0.63	0.70	0.63	0.73	0.63	0.77	0.63	0.76
12.64	1.00	1.12	1.00	1.29	1.00	1.34	1.00	0.94	1.00	1.05	0.99	1.20	0.99	1.29
12.256	1.00	1.08	0.99	1.26	0.99	1.39	0.97	0.94	0.97	1.06	0.76	1.07	0.96	1.35
14.8	1.00	0.94	1.00	0.98	1.00	1.00	0.38	0.40	0.38	0.41	0.38	0.44	0.38	0.44
14.64	1.00	1.08	1.00	1.20	1.00	1.25	0.91	0.97	0.91	1.03	0.91	1.14	0.92	1.20
16.8	1.00	1.00	1.00	1.03	1.00	1.02	0.31	0.33	0.31	0.34	0.31	0.35	0.31	0.36
net	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time
8.8	0.99	*	1.00	*	1.00	*	1.00	*	1.00	*	1.00	*	1.00	*
8.16	1.00	*	0.99	*	0.98	*	1.00	*	1.00	*	0.99	*	0.99	*
8.32	1.00	*	1.00	*	1.01	*	1.00	*	1.00	*	1.00	*	1.00	*
10.8	0.99	*	0.99	*	0.99	*	1.00	*	0.99	*	0.99	*	0.99	*
10.32	1.00	1.08	1.00	1.15	1.00	1.17	1.00	1.09	1.00	1.14	1.00	1.19	0.99	1.19
10.64	0.99	1.04	1.00	1.15	1.01	1.21	1.00	0.94	0.99	1.02	1.00	1.12	1.00	1.11
12.8	1.05	1.05	1.00	1.09	1.00	1.09	0.99	1.00	1.03	1.06	1.02	1.14	1.02	1.13
12.64	1.00	1.11	1.00	1.19	1.00	1.25	1.00	1.02	0.99	1.16	0.99	1.21	0.99	1.28
12.256	0.99	1.13	0.99	1.27	0.99	1.35	1.00	1.01	0.99	1.18	0.99	1.33	0.99	1.36
14.8	1.00	0.94	1.00	1.02	1.00	1.11	1.00	1.11	1.00	1.06	1.00	1.13	1.00	1.17
14.64	1.00	1.05	1.00	1.15	1.00	1.19	1.00	0.97	1.00	1.06	1.00	1.16	1.00	1.22
16.8	1.00	1.06	1.00	1.13	1.00	1.15	1.00	1.01	1.00	1.06	1.00	1.13	1.00	1.16

the new preconditioners improve upon the known ones (entries < 1), and it highlights some interesting trends, as we will see later on. However, for the smaller instances we elected not to report running times, as each system was timed separately, and the time required to solve one system was too near to the precision of the timing routines, and therefore too affected by errors, to be significant.

We will now comment on the results for the three classes of problems separately.

`goto` *instances.* For these instances, the new preconditioners are quite competitive with the tree one, obtaining, when parallel arcs are added, improvements of up

TABLE 5.2
More detailed results for 10.32 *instances.*

	goto 10.32				grid 10.32				net 10.32			
	T	B-b	TP	BP-b	T	B-b	TP	BP-b	T	B-b	TP	BP-b
0	729	0.78	0.37	0.26	12	0.95	0.95	0.95	10	1.00	0.98	1.00
1	95	0.78	0.85	0.69	10	0.98	0.96	0.96	11	0.98	1.00	0.98
$k/4$	77	0.81	0.77	0.62	11	1.00	0.98	0.98	11	1.00	0.98	0.98
$k/2$	47	0.82	0.86	0.68	16	0.98	0.94	0.93	15	1.00	1.00	1.00
$3k/4$	27	0.87	0.92	0.80	14	0.99	0.94	0.93	15	1.00	1.00	1.00
k-1	16	0.95	1.00	0.92	7	1.00	0.85	0.85	7	1.00	1.00	1.00
k	16	0.95	1.00	0.92	3	1.00	0.87	0.87	3	1.00	1.00	1.00

to a factor of five in iterations count, and only slightly less so in time. Among BCT preconditioners, the more complex heuristics (ii.b) and (ii.c) clearly outperform the simpler (ii.a), with the most complex one, (ii.c), oftentimes slightly outperforming (ii.b). There does not seem to be a clear trend regarding graph density, with denser graphs sometimes benefiting more and other times benefiting less from BCT preconditioners than sparse ones; however, there is a clear positive trend with graph size, in that larger problems benefit most from BCT preconditioners.

`grid` *instances.* Even for these instances, enriching the support graph by adding more arcs turns out to be in general a good strategy; this time, however, it is the addition of parallel arcs that makes up the largest part of the improvement. In fact, although improvements of up to a factor of three are still obtained, the T+P preconditioner is the most competitive one. BCT preconditioners often obtain smaller iteration counts than the corresponding tree one, but only slightly so, and this does not pay for the extra cost of finding the preconditioner. Among BCT preconditioners, the more complex heuristics (ii.b) and (ii.c) fail, on this class of instances, to obtain more than minor improvements with respect to the simpler (ii.a), so that the most complex one, (ii.c), is usually the slowest one. The same positive trend with graph size as in the `goto` case shows up; this time, however, there appears to be something of a more defined trend with density, too, as improvements tend to be more consistent for problems on sparser graphs.

`net` *instances.* For this class of instances, the new preconditioners are not competitive with the tree one. Although enriching the support graph fairly often decreases the iterations count, the decrease is always minimal, and adding parallel arcs does not help; for these instances, all the mechanisms for enriching the support graph actually increase the total running time required for solving the systems.

In order to better understand the behavior of the preconditioners, it is worthwhile to examine some of the results in greater detail. In Table 5.2 we report some data about the number of iterations required to solve problems of the same size (the class `10.32`) generated by the three different generators. For each generator, we report seven rows corresponding to the systems solved at IP iterations 0, 1, $k/4$, $k/2$, $3k/4$, $k-1$, and k, where k is the index of the last iteration; this is a significant sample of the matrices generated during the IP algorithm. In particular, the systems of iteration 0 are those solved to find an initial interior solution, for which $\Theta = I_m$ (i.e., $M = EE^T$ [7]). For each generator, the column T reports the number of PCG iterations required for solving the system using the tree preconditioner, while the columns *TP*, *B-b*, and *BP-b* have the same meaning as the columns *Iter* in the corresponding sections of Table 5.1.

TABLE 5.3
Number of arcs added to the support graph.

	T	$B-a$		$B-b$		$B-c$	
`goto`	#TP	#B	#BP	#B	#BP	#B	#BP
0	927	2	0	410	8	410	8
1	927	2	0	384	8	384	8
$k/4$	843	54	14	385	33	385	33
$k/2$	724	86	27	370	72	371	73
$3k/4$	722	88	26	368	72	369	73
$k-1$	721	87	26	367	70	367	70
k	721	84	26	354	69	354	69
`grid`	#TP	#B	#BP	#B	#BP	#B	#BP
0	15	5	2	13	6	13	6
1	401	96	35	145	53	188	67
$k/4$	569	5	1	14	3	14	3
$k/2$	546	5	1	12	3	12	3
$3k/4$	544	5	2	13	3	13	3
$k-1$	528	2	1	6	2	6	2
k	498	0	0	1	0	1	0
`net`	#TP	#B	#BP	#B	#BP	#B	#BP
0	5	29	0	41	0	41	0
1	4	8	0	24	0	24	0
$k/4$	4	9	0	23	0	23	0
$k/2$	6	9	0	20	0	20	0
$3k/4$	6	11	0	20	0	20	0
$k-1$	6	7	0	13	0	13	0
k	5	4	0	8	0	8	0

The results show that the systems corresponding to `goto` instances are considerably more difficult to solve than those corresponding to either `grid` or `net` instances. The effect of the BCT preconditioner on `goto` instances is larger in the first iterations, where the tree preconditioner is less effective, and diminishes as the IP algorithm proceeds; for `grid` and `net` instances the effect is very limited across the board, and no clear trend emerges. The effect of the "+P" variant is less easy to characterize, with a decreasing trend showing up for `goto` instances and no clear trend emerging for `grid` instances. It is, however, interesting to note that, for the `goto` instances, in the very final iterations of the IP algorithm the "+P" variant alone does not seem to produce any improvement to the tree preconditioner, while it is capable of helping out, albeit slightly, the BCT one.

The above results can be better understood by looking at Table 5.3, where the number of arcs added to the spanning tree in the different variants of preconditioners is reported. In the table, the three groups of two columns labeled $B-a$, $B-b$, and $B-c$ correspond to the heuristics (ii.a), (ii.b), and (ii.c), respectively, for the maximum-weight BCT computation. In each group, the column #B reports the number of arcs in the second level of the BCT found by the heuristic, and the column #BP reports the number of arcs "parallel" to those in the second level of the BCT. Finally, the column #TP reports the number of arcs "parallel" to those of the original spanning tree. The table shows the (averaged) results for the `10.8` instances for the three different generators; these results can be considered typical. For each generator, we report seven rows corresponding to the systems at the same seven "snapshots" of the optimization process as in Table 5.2.

These results show that the effectiveness of the new preconditioners—at least, relative to that of the tree one—is directly related to the number of arcs that are added to the support graph. In particular, for `goto` instances the heuristic (ii.a) adds considerably fewer arcs than (ii.b) or (ii.c), and in fact it is less effective; furthermore, a large number of "parallel" arcs are added to the support graph, and in fact the corresponding preconditioners improve upon those where this is not done. For `grid` instances, the BCT heuristics are not capable of adding many arcs to the support graph (except in the second iteration), while a large number of "parallel" arcs are added; indeed, adding parallel arcs is what makes the difference for these instances. Finally, for `net` instances very few arcs are added to the support graph by both methods, and this directly translates into the inferior performances of the new preconditioners.

These results lead us to the following conclusions:

- Of all heuristics for finding the BCT, (ii.b) is the one that obtains the best performances, being far more efficient than (ii.a) in adding arcs to the support graph and only slightly less so than (ii.c), but is, however, much more costly; this confirms that balancing the effort for finding/factoring the preconditioner with the improvement in the convergence rate of the PCG is crucial.
- Enriching the support graph turns out to be a good strategy for those problems that are not easily solved by the tree preconditioner, whereas it is less useful for systems that are already very efficiently solved by the tree preconditioner.
- The relative efficiency of the new subgraph-based preconditioners with respect to the tree one is well predicted by the number of arcs added to the spanning tree; this has been confirmed by the analysis of data for all the instances, which we do not report here to save space.

5.3. Final results. Given the results of the previous section, we have tested the effect of an automatic rule for choosing the preconditioner. Sticking to heuristic (ii.b) for finding the BCT, we initially start by using both BCT and "+P" preconditioners. The number of arcs added to the support graph S by both operations are counted; if this number is larger than a fixed threshold, the preconditioner actually includes those arcs; otherwise the operation is disabled in that and all the following IP iterations. This choice is motivated by the fact that the tree preconditioner becomes more and more efficient as the IP algorithm proceeds; hence if adding arcs to the support graph is not likely to help at a given iteration, it is somewhat unlikely that is it going to help later. Permanently disabling the rule is simple and has the advantage of avoiding the cost for finding a BCT and/or parallel arcs that are not going to be used (the cost for factoring M_S is not paid anyway because the decision is taken before the factorization).

The analysis of the obtained results has shown that reasonable thresholds are 45% for the BCT and 10% for "+P"; that is, using the BCT is disabled if it does not add at least as many as $0.45(n-1)$ arcs, and using parallel arcs is disabled if it does not add at least as many as $0.10(n-1)$ arcs. These thresholds appear to work well for all three classes of instances.

The results of using these rules are shown in Table 5.4; as for the previous tables, the results are relative to those obtained by the tree preconditioner ("+ D" or not, according to the class of instances).

The table shows that the rules are, at least in these instances, capable of choosing the right preconditioner at the right time. Most often the chosen preconditioner is always the same for all the IP iterations, but in some cases a switch happens during

TABLE 5.4
Results with the automatic selection rule.

	goto		grid		net	
	Iter	Time	Iter	Time	Iter	Time
8.8	0.80	*	0.87	*	1.00	*
8.16	0.69	*	0.97	*	1.00	*
8.32	0.64	*	0.97	*	1.00	*
10.8	0.70	0.84	0.82	0.90	1.00	1.00
10.32	0.62	0.68	0.94	0.96	1.00	1.00
10.64	0.62	0.66	0.99	0.99	1.00	0.99
12.8	0.79	0.85	0.63	0.70	1.00	1.00
12.64	0.64	0.62	1.00	0.97	1.00	1.00
12.256	0.50	0.56	0.97	0.94	1.00	1.00
14.8	0.30	0.39	0.38	0.40	1.00	1.00
14.64	0.53	0.59	0.91	0.97	1.00	1.00
16.8	0.19	0.22	0.31	0.33	1.00	1.00

the optimization process which may modify the running time w.r.t. the case where the same preconditioner is used throughout the IP algorithm, either decreasing it (as for `goto` 14.64 and `net` 10.64) or increasing it (as for `grid` 12.64) but always by a relatively small amount. More sophisticated selection rules may further improve the results, but the obtained ones already show that BCT preconditioners, if carefully implemented and paired with appropriate automatic selection rules, can effectively complement tree preconditioners as a solution tool for the linear systems arising in IP methods for MCF problems.

6. Conclusion and directions for future work. We have proposed a new family of subgraph-based preconditioners for the solution of the KKT systems arising in the solution of MCF problems through IP methods. For some families of instances, these preconditioners improve on those known in the literature both in iterations count and total time. Also, the new family of preconditioners offers some flexibility in the way to select the subgraph, thereby allowing us to tune the trade-off between the cost of computing and using the preconditioner and the corresponding reduction in the number of PCG iterations. Therefore, we believe that our new preconditioners can be a valuable tool for constructing efficient IP algorithms for MCF problems. Furthermore, they may find broader application for the solution of linear systems with *M-matrices* [3].

Further work along this line of research will involve perfecting our implementation of an IP method for MCF problems and testing it against efficient MCF codes from the literature; the results will be presented in a forthcoming paper, where all the issues relative to the effectiveness of the different variants of preconditioners for different IP algorithms will be discussed. Also, other fast heuristics for the maximum-weight BCT problem will be tested, trying to find an optimal compromise between the quality of the BCT found and the extra cost involved in finding it; that is a critical parameter for the overall efficiency of the approach. Finally, theoretical investigations on the class of BCT graphs may pay off in terms of better heuristics, characterization of some classes of graphs where "large" BCTs can be easily found, and a better understanding of the complexity class of the maximum-weight BCT computation.

REFERENCES

[1] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] W. Anderson and T. Morley, *Eigenvalues of the Laplacian of a graph*, Linear and Multilinear Algebra, 18 (1985), pp. 141–145.

[3] M. Bern, J. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo, *Support-graph preconditioners*, SIAM J. Matrix Anal. Appl., submitted.

[4] J. Castro, *A specialized interior-point algorithm for multicommodity network flows*, SIAM J. Optim., 10 (2000), pp. 852–877.

[5] D. Chen and S. Toledo, *Vaidya's preconditioners: Implementation and experimental study*, Electron. Trans. Numer. Anal., 16 (2003), pp. 30–49.

[6] D. Cvetkovic, M. Doob, and H. Sachs, *Spectra of Graphs*, Academic Press, New York, 1979.

[7] A. Frangioni and S. S. Capizzano, *Spectral analysis of (sequences of) graph matrices*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 339–348.

[8] A. Frangioni and C. Gentile, *Interior Point Methods for Network Problems*, Tech. report 539, IASI-CNR, Rome, 2000.

[9] K. Gremban, G. Miller, and M. Zagha, *Performance evaluation of a parallel preconditioner*, in Proceedings of the 9th IEEE International Parallel Processing Symposium, Santa Barbara, CA, 1995, pp. 65–69.

[10] J. Júdice, J. Patricio, L. Portugal, M. Resende, and G. Veiga, *A study of preconditioners for network interior point methods*, Comput. Optim. Appl., 24 (2003), pp. 5–35.

[11] D. Karger and N. Srebro, *Learning Markov networks: Maximum bounded tree-width graphs*, in Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM Press, New York, SIAM, Philadelphia, 2001, pp. 392–401.

[12] N. K. Karmarkar and K. G. Ramakrishnan, *private comunication*, 1988.

[13] S. Mehrotra and J. Wang, *Conjugate gradient based implementation of interior point methods for network flow problems*, in Linear and Nonlinear Conjugate Gradient-Related Methods, L. Adams and J. Nazareth, eds., SIAM, Philadelphia, 1996, pp. 124–142.

[14] L. Portugal, F. Bastos, J. Júdice, J. Paixao, and T. Terlaky, *An investigation of interior-point algorithms for the linear transportation problem*, SIAM J. Sci. Comput., 17 (1996), pp. 1202–1223.

[15] L. Portugal, M. Resende, G. Veiga, and J. Júdice, *A truncated primal-infeasible dual-feasible network interior point method*, Networks, 35 (2000), pp. 91–108.

[16] M. Resende and P. Pardalos, *Interior point algorithms for network flow problems*, in Advances in Linear and Integer Programming, J. Beasley, ed., Oxford University Press, Oxford, UK, 1996, pp. 147–187

[17] M. Resende and G. Veiga, *An implementation of the dual affine scaling algorithm for minumum-cost flow on bipartite uncapacited networks*, SIAM J. Optim., 3 (1993), pp. 516–537.

[18] M. Resende and G. Veiga, *An efficient implementation of a network interior point method*, in Network Flows and Matching: First DIMACS Implementation Challenge, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 12, D. Johnson and C. McGeoch, eds., AMS, Providence, RI, 1993, pp. 299–348.

[19] T. Roos, T. Terlaky, and J.-P. Vial, *Theory and Algorithms for Linear Optimization: An Interior Point Approach*, John Wiley, Chichester, UK, 1997.

[20] D. Rose, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.

[21] R. Tarjan and M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Comput., 13 (1984), pp. 566–579.

[22] P. Vaidya, *Solving Linear Equations with Symmetric Diagonally Dominant Matrices by Constructing Good Preconditioners*, Tech. report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1990.

[23] C. Wallacher and U. Zimmermann, *A combinatorial interior point method for network flow problems*, Math. Programming, 56 (1992), pp. 321–335.

[24] S. Wright, *Primal-Dual Interior-Point Methods*, SIAM, Philadelphia, 1997.

[25] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Disc. Meth., 2 (1981), pp. 77–79.