Prim-based Support-Graph preconditioners for Min-Cost Flow Problems

A. Frangioni[§] C. Gentile[*]

Abstract

Support-graph preconditioners have been shown to be a valuable tool for the iterative solution, via a Preconditioned Conjugate Gradient method, of the KKT systems that must be solved at each iteration of an Interior Point algorithm for the solution of Min Cost Flow problems. These preconditioners extract a proper triangulated subgraph, with "large" weight, of the original graph: in practice, trees and Brother-Connected Trees (BCTs) of depth two have been shown to be the most computationally efficient families of subgraphs. In the literature, approximate versions of the Kruskal algorithm for maximum-weight spanning trees have most often been used for choosing the subgraphs; Prim-based approaches have been used for trees, but no comparison have ever been reported. We propose Prim-based heuristics for BCTs, which require nontrivial modifications w.r.t. the previously proposed Kruskal-based approaches, and present a computational comparison of the different approaches, which shows that Prim-based heuristics are most often preferable to Kruskal-based ones.

Keywords*: Min Cost Flow problems, Interior Point algorithms, Preconditioned Conjugated Gradient method, Prim algorithm.*

1 Introduction

We present new heuristics, based on the Prim algorithm for the Maximum-weight Spanning Tree (MST) problem, for finding "large-weight" triangulated subgraphs of a given weighted graph. These heuristics have application to the problem of finding efficient support-graph preconditioners for the iterative solution, via a Preconditioned Conjugate Gradient (PCG) method, of the (core part 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. Previous theoretical [9, 10] and/or experimental [14, 7] analyses have shown that support-graph preconditioners, which rely on the idea of extracting a large-weight triangulated subgraph of the original graph, are effective provided that care is taken in properly balancing the effort required for finding the subgraph

and factoring the preconditioner and the corresponding decrease in iterations count of the PCG approach. In the literature, the large-weight subgraph has most often been chosen by heuristics based on (approximate versions of) the Kruskal algorithm for MST; Prim-based heuristics have been proposed only for the case of trees, but no comparison between the two approaches have ever been reported. We present such a comparison, which shows that Prim-based approaches are most often preferable to Kruskal-based ones on large instances, except possibly on dense ones. While Prim- and Kruskal-based approaches are almost identical if the subgraph is restricted to be a tree, nontrivial modifications are needed when Brother-Connected Trees (BCT) of level two [7] are used instead; we present different heuristics based on the Prim algorithm and computationally analyze their performances within the IP application.

The structure of the paper is the following: in Section 2 we briefly review the previous work done in this area and introduce support-graph preconditioners. In Section 3 we discuss the new Prim-based heuristics for finding large-weight BCTs of level two; then, Section 4 presents the results of a computational experience aimed at assessing the effectiveness of the new heuristics within the framework of IP approaches to MCF. Finally, conclusions are drawn in Section 5.

2 Support-graph preconditioners

2.1 Interior Point approaches to Min-Cost Flow problems

Let $G = (N, A)$ be a directed graph, with $m = |A|$ and $n = |N|$; the Min-Cost Flow (MCF) problem is the following linear program

$$\min \{ cx \ : \ Ex = b \ , \ 0 \leq x \leq u \} \ , \tag{1}$$

where E is the node-arc incidence matrix of G, 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 (e.g., [2, 5, 3, 4] among many others).

Specialized IP methods have been shown [17, 14] to be computationally competitive for the solution of large-scale MCF problems. At each iteration of these methods, linear systems of the form

$$(E\Theta E^T)\Delta y = d \tag{2}$$

have to be solved, where Θ and d are respectively a $m \times m$ diagonal matrix with positive entries (that is, a weight $\theta_{ij} > 0$ is assigned to each arc $(i, j) \in A$) and a vector of $\mathbb{R}^n$, which depend on the current solution and on the IP algorithm chosen. Actually, since the system is rank-deficient, at least one of the rows/columns may be eliminated; this is not always the best option, as discussed in Section 2.5.

The solution of (2) typically represents by far the main computational burden of IP algorithms. As $M = E\Theta E^T$ is symmetric and positive (semi)definite, in

general-purpose Linear Programming IP solvers (2) would typically be solved through a Cholesky factorization, preceded by a heuristic re-ordering of the rows of E (nodes of G) in order to diminish as much as possible the *fill-in* of the resulting Cholesky factor. However, for structured problems such as MCF the Cholesky factorization is too slow to make the IP algorithm competitive with the many available efficient "combinatorial" approaches to MCF (e.g., see [8]), and alternatives have to be devised. The Preconditioned Conjugate Gradient (PCG) method offers one such alternative. A proper choice of the preconditioner must balance the cost of forming and factoring it and the corresponding savings due to the decrease of the number of CG iterations required to (approximately) solve the system.

2.2 Support-graph preconditioners

Theoretical analysis [9, 10] and experimental studies [15, 14, 7] have shown that *support-graph* preconditioners are effective in this setting. These are matrices of the form

$$M_S = E_S \Theta_S E_S^T \ , \tag{3}$$

where S is a subgraph 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. This can also be "extended" to

$$M'_S = M_S + \rho \ diag(M - M_S) \tag{4}$$

where $diag(X)$ is the diagonal matrix having as the diagonal elements those of X, and ρ is a parameter that can be chosen according to the structure of the MCF problem at hand; clearly, M'_S is not substantially more costly to invert than M_S, while incorporating information about *all* arcs, rather than only about those in S. Indeed M'_S turns out to be more effective than M_S on some [7]—but not all [10]—classes of instances; since adding the diagonal (or not) changes basically nothing in the theory of support-graph preconditioners, in the following we will always refer to (3), intending that (4) can be used instead if it turns out to be computationally convenient.

In order for M_S to be inexpensive to invert (or factor), it must have very low fill-in: a way to ensure this is to ask that *no* fill-in is incurred, i.e., that S is a *triangulated* graph [18] (such that every cycle of length at least four has an edge joining two nonconsecutive vertices in the cycle). In particular, trees are obviously triangulated graphs; tree-based preconditioners [15, 14, 13, 10] choose S as a(n approximate) MST of G, the weight of each arc (i, j) being the corresponding θ_{ij}. 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 approximate MST can be constructed in roughly $O(m)$ with a variant of the classical Kruskal algorithm where arcs are only approximately sorted using a "bucket" data structure with m buckets; this is the strategy adopted in all papers except [14], where a Prim algorithm based on a Fibonacci heap data

structure is used instead. However, no rationale for the specific choice is given in any of the papers.

Tree-based preconditioners 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 but those corresponding to the basic optimal solution, that form a spanning tree; hence $M_S \approx E\Theta E^T$ in the last iterations of the IP method. This is true also in the degenerate case [10]. However, these preconditioners are less effective in the first iterations of the IP approach, where the weights are "more evenly" distributed on a larger subset of the arcs of G (in the very first iteration it can even be $\theta_{ij} = 1 \ \forall \ (i,j) \in A$). This has suggested to use, at least in the first IP iterations, support-graph preconditioners with "larger" (strictly containing a spanning tree) triangulated subgraphs.

In [7] we have shown that the latter approach can improve the overall performances provided that due care is taken in seeking the right balance between the increased cost of finding S (and factorizing M_S) and the savings due to the decrease of the PCG iterations, with respect to a tree-based preconditioner. At first, one may think that the immediate extension of the approach would be to use as S the maximum-weight triangulated subgraph of G; however, this turns out to be impossible, since:

- the maximum-weight triangulated subgraph problem is $\mathcal{NP}$-hard [11], and it clearly makes no sense to employ an exact solution approach (such as Branch&Bound) in this application;

- even if it were computationally feasible to exactly (or approximately, with some tight a-priori ratio) find a maximum-weight triangulated subgraph of G, using it as S would not necessarily result in an efficient approach, due to the above-mentioned delicate balance between the extra cost of finding and factoring a "larger" preconditioner M_S and the decrease in PCG iterations [7];

- once S has been determined, some work still has to be done to find the "good" ordering of the nodes, i.e., a $n \times n$ permutation matrix P_n such that the reordered matrix $P_n M_S P_n^T$ has a Cholesky factorization without fill-in. For the case of trees, P_n corresponds to any permutation $\mathcal{P}$ (such as that given by a reverse Breadth-First Search) 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 $\mathcal{P}$, and therefore is already implicitly given by, e.g., the description of the tree in terms of the predecessor function $Pred[\cdot]$; conversely, for the general case P_n has to be explicitly computed [19].

This suggests the use of appropriate sub-families of triangulated graphs where the computation of S and of $\mathcal{P}$ can be organized as to be very efficient in practice. For this purpose, the family of brother-connected trees has been defined in [7].

2.3 Brother-connected trees

A subgraph $S = (N, A_S)$ of G is a *brother-connected tree* (BCT) if either it is 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 brother-connected tree.

This definition 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 anyone 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 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.

The *depth* of a BCT S 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. It is easy to show that BCTs are triangulated graphs; furthermore, one can show [7, Theorem 2.3] that for any brother-connected tree S in G, its representation as a tree $\mathcal{T}$ allows one to compute a "good" ordering $\mathcal{P}$ (such that M_S has a Cholesky factor with no fill-in). For a BCT of depth two, for instance, one just has to "merge" the natural (sons before fathers) ordering of the sub-trees of level 2 with that of the tree of level 1, in which brothers may appear in any order. The result can also be generalized to any positive definite matrix M with a BCT support, thereby allowing modifications to the preconditioner such as that of (4).

All this allows one to define an algorithm that constructs a fill-in-free Cholesky factor of M_S, for S a BCT of depth h, in $O(nh^2)$; all the trees at the same level q can be represented with a unique predecessor function $Pred[q]$ defined on the nodes, such that $Pred[q][u] = v$, if v is father of u at depth q. Using the same data structures, 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—can be constructed; any PCG iteration has then a complexity of $O(nh + m)$. In [7], BCTs of depth two have been shown to yield the best compromise between the extra cost associated with the increase in the cardinality of S (a BCT of depth two can have up to $2n - 3$ arcs) and the improvement in the convergence rate of the PCG.

2.4 Kruskal-based heuristics

A crucial component of the overall approach is then the heuristic that is used to find the large-weight BCT in the first place; it has to be both effective, since

a larger BCT can be assumed to provide a preconditioner with better spectral properties, and efficient, in order not to overbalance the improvement due to the better preconditioner. In [6, 7], several different two-stage heuristics have been proposed based on the following general scheme:

(i) find an initial spanning tree T;

(ii) then, add extra arcs forming trees among brothers in T.

Choosing the initial tree as a(n approximate) MST appears to be the best choice, and is backed by some results that can be proven about the worst-case performances. Since arcs in phase (ii) are added in a greedy fashion, and therefore it is beneficial to try adding more promising arcs first, the initial tree in phase (i) is constructed by a Kruskal algorithm, whose complexity is dominated by that of the initial sorting of the arcs: the costly sorting is then exploited by both phases. Also, as in most of the previous literature, arcs are only approximately sorted in $O(m)$ using a "bucket" data structure with m buckets; as discussed in Section 4.1 we have later questioned the choice of the approximate sorting, but the computational results keep supporting this approach (among the Kruskal-based ones).

Three different variants of the second phase have been proposed in [7], in which the heuristic can change more and more the structure of the original spanning tree: in the first (ii.a) the final ordering of the nodes is arbitrarily fixed as any "good" ordering for T and the arcs out of T are added if they are compatible with the fixed ordering and they form paths among brothers, in the second (ii.b) the restriction that level-two trees must be paths is kept, but the ordering between brothers can be changed, while in the third (ii.c) trees in the second level are not restricted to be paths. The growing freedom enjoyed by heuristics (ii.b) and (ii.c) is paid in terms of more data structures and an increase of cost; however, it also allows *promotion* operations to be performed, whereby a node connected with its grandfather is "promoted" as a brother of its former father if this helps inserting a promising arc in the BCT. Experience shows that the two more complex heuristics perform much better than the simpler one in increasing the size of the BCT, usually resulting in better overall performances of the approach, at least for those classes of instances where BCT preconditioners actually help.

2.5 Further improvements

The procedures recalled above can be further improved by applying some general "tricks" which do not depend from the choice of S:

- using as preconditioner the matrix M'_S of (4), thereby providing the preconditioner with information about the arcs that are left out of S;
- adding to S all arcs (i, j) which are "parallel" to arcs already belonging to it: this does not change the nonzero pattern of M'_S while increasing its weight;

- removing from both M and M_S (at least) one row and solving the equivalent full-rank system rather than applying the PCG directly to the rank-deficient system.

Our previous experience [7] has provided us with guidelines about when each of these (orthogonal) options should be employed; since the focus of this paper is on alternative ways for computing S, in the computational results we use for both the Prim-based and Kruskal-based the particular combination that has been found to be the best for each class of instances. The interested reader is referred to [6, 7] for a through description of these issues.

3 Prim-based heuristics for maximum-weight BCT

The Kruskal-based heuristics for maximum-weight BCT require two separate phases: in the first phase the (approximated) ordered list of arcs is scanned to find the (approximated) MST, in the second phase the same list is scanned to add arcs in the second level of the BCT. This is necessary because the BCT definition requires a root to be fixed, otherwise the concept of "brother" is undefined, but the Kruskal algorithm iteratively constructs a maximum-weight forest of growing cardinality, that only at the very last step becomes a tree. On the contrary, the Prim algorithm constructs the maximum-weight spanning tree starting from an arbitrary node chosen as the root of the tree, and therefore brotherhood is well defined at each point of the algorithm: this allows us to obtain Prim-based heuristics for the maximum-weight BCT with some slight modifications of the Prim-algorithm, thereby scanning the node-set of the graph only once. Furthermore, the Prim algorithm (hence our heuristics) does not require ordering of the whole arc set.

We recall the Prim algorithm in the following box, where $G = (N, A, w)$ is a weighted graph, $Pred$ is the predecessor function of the tree that is going to be constructed, d is the vector of labels, and Q is a priority queue (with operations "find-max(Q)= $\arg\max\{d(u)|u \in Q\}$" and "increase-key(v,Q)", that either add node v to Q if not present or update Q according to an improved label for v):

for each $u \in N \setminus \{r\}$ **do** $d(u) = -\infty$;
$d(r) = 0$; $Pred(r) = nil$; $Q = \{r\}$;
do
  $u =$ find-max(Q); $Q = Q \setminus \{u\}$; $d(u) = +\infty$;
  for each $(u, v) \in A$ **do**
    if $d(v) < w_{uv}$ **then** ($\star$)
      begin
        $d(v) = w_{uv}$; $Pred(v) = u$; increase-key(v,Q);
      end
while $Q \neq \emptyset$

In the main step ($\star$) of the Prim algorithm, the arcs connecting the last extracted node u with previously extracted nodes v (easily recognized by their label $d(v)$,

which has been set to $+\infty$ immediately extraction from Q) are discarded; the basic idea of Prim-based heuristics for the maximum-weight BCT is to consider these arcs and check if they can be added to the BCT. This simply amounts to adding the following "else" branch to ($\star$):

else
  if $d(v) == +\infty$ **and**
    v and u are brothers according to $Pred$ **and**
    (u, v) does not form a cycle with selected second level arcs
  then add (u, v) to the second level of the BCT

Checking if a candidate arc for the second level does not form cycles is performed by using a Union-Find structure, as in the Kruskal algorithm. Candidate arcs are accepted in their scanning order, thus the ordering in which arcs outgoing from a node are scanned has an impact upon the obtained BCT.

The experience reported with Kruskal-based preconditioners [7] shows that two points are important:

- the predecessor function (i.e., the ordering) for the second level of the BCT must be computed only when *all* arcs in the BCT have been selected, for otherwise arcs may be refused only for incompatibility with the already decided ordering (see remarks on the Kruskal-based heuristic (ii.a) in Section 2.4);
- a *promotion* operation modifying the structure of the first level tree T is often useful to add new arcs.

The promotion is performed when an arc joins the extracted node u with its grandfather v, as depicted in Figure 1. In (a), solid arrows represent the first level arcs of the BCT, dotted arrows represent second level arcs, while dashed arcs are under examination. Ordinarily, arc (u, v) could not be accepted because u already has a father, q, and v is not a brother of u; however, one can declare u to be a son of v instead while keeping arc (q, u) in the BCT, just by moving it from the first level to the second one (since u becomes brother of q). We call this a *simple promotion* (SP). However, at each step of the algorithm, the promotion operation is incompatible with the brother-connection operation, as the former changes the father of the current node u from q to v, while the brother-connected operations searches for nodes connected to u that are also sons of q (i.e., brothers of u before the promotion). Therefore, the choice between the two operations is done by computing the weight contribution of each, and applying the one giving the best result. However, when promotion is applied, the resulting BCT can be improved by searching for brothers of node u after the promotion, i.e., for sons of v connected to u (cf. Figure 1(c)). We call this improved operation *wide promotion* (WP); in practice, this consists of applying first the simple promotion and then the brother-connection on the modified BCT, hence trying to obtain the advantages of both.

An interesting property of our Prim-based heuristics is the following:

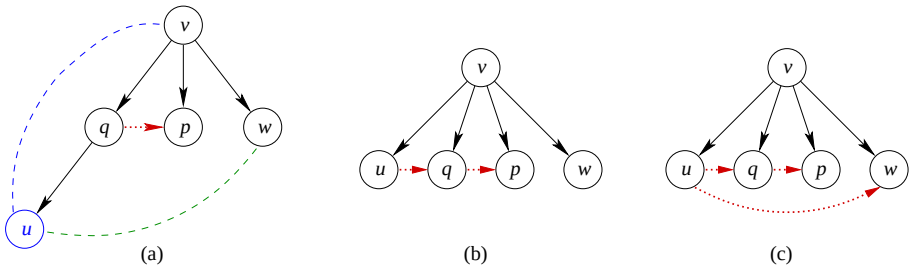

Figure 1: (a) is the initial situation, (b) is the BCT after SP, (c) is the BCT after WP

Proposition 3.1 *Heuristics Prim-SP and Prim-WP determine a BCT containing the optimal MST computed by the Prim algorithm.*

Proof: The Prim label $d(\cdot)$ is not modified by brother-connections or simple/wide promotions, as they are performed only between the node u just extracted from Q and nodes already extracted. In particular, brother-connections do not change any first-level arc, while promotions only moves the predecessor arc of u from the first level to the second level of the BCT. Hence, the sequences of node insertions and extractions from Q is not modified, and therefore the optimal MST is contained in the final BCT. □

By [7, Corollary 3.1], all heuristics for costructing BCTs which augment the maximum-weight spanning tree are 2-approximated; hence, using Proposition 3.1, we can conclude that both heuristics Prim-SP and Prim-WP are 2-approximated algorithms for the maximum-weight BCT problem.

Two elements are important in the practical implementation of the above approaches:

- how Q is implemented, which may even impact on the worst-case performances of the algorithm: for instance, a binary heap results in $O(m \log n)$ complexity while an unordered list results in $O(n^2)$ complexity, hence the former is favored in sparse graphs while the latter is favored in dense graphs;
- in which order the arcs leaving the current node u are scanned: basically, one can either accept the ordering of the arcs that is implicit in the description of the instance (that is, one with presumably no specific property) or order the arcs by nonincreasing weight, so that "more interesting" arcs are examined first.

The impact of those implementation issues is discussed in the next section.

4 A computational comparison of preconditioners

In this section, we present the results of a large-scale computational test aimed at assessing the effectiveness of Prim-based preconditioners with respect to Kruskal-based ones. The tests have been performed on a PC with an Athlon MP 2400+ and 1Gb RAM, running Linux. The code was compiled using the GNU g++ compiler version 3.3, using optimization option "-O3".

For our tests, we selected three well-known random generators of MCF problems: `goto` (GridOnTOrus), `gridgen` and `netgen`. For each generator, we generated a total of 8 classes of instances named `gen`k_d, where `gen` is the specific generator, $n = 2^k$ (for k comprised between 12 and 16) is the number of nodes and d (comprised between 8 and 256) is the average density of the graph; we could not generate `net`16_64 instances because of limitations of the generator. The data set is similar to that of [7], except that some smaller-size instances were dropped in favor of larger-size—and therefore more significant—ones; hence, for these instances the "optimal" performances of Kruskal-based BCT preconditioners have already been extensively studied. As usual, source code for the generators can be downloaded, e.g., at `http://www.di.unipi.it/optimize/Data/`; parameters for reproducing the instances are also available upon request from the authors.

For all the 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. We remark that our testing methodology completely disregards the possible effects that the different iterative solvers, by providing slightly different solutions to the same system, may have on the overall IP approach; in actuality, the sequences of KKT systems solved by the IP approach using each solver would be different. However, since the focus of the paper is on the efficiency of the KKT system solution, we believe that the chosen testing methodology is the most appropriate one: comparing the actual solution time of the IP algorithm using each solver may incur serious risks of distorting the results, if only because the number of IP iterations may vary. Furthermore, the KKT solution time is a very significant part of the total IP time: it is never less than 65%, most often much higher, and the percentage grows as the size of the instances does, overcoming 95% on the largest ones.

As in [7], a typical adaptive stopping rule is employed for the PCG which terminates the procedure as soon as a vector Δy is found such that

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

for all $i \in N$, where $\bar{x}$ is the current primal solution of the IP algorithm; in our experiments, $\gamma = 0.1$ and $\varepsilon =$`1e-4` is also both the relative feasibility tolerance

and the relative optimality tolerance of the IP approach. This stopping rule allows early termination in the initial IP iterations, while ensuring that the accuracy is always "enough" to guarantee convergence of the IP algorithm; in our experiments, the $|b_i - E_i\bar{x}|$ values are also saved along with Θ and d to ensure that the accuracy requirements are exactly the same for each different preconditioner. Since the focus of the paper is on the solution of the KKT system, we have elected not to employ any optimal face detection procedure [16] to terminate the IP approach early on.

The computational experiments were performed in two phases. In the preliminary phase, a significant subset of the instances were tested with some variants of the Prim-based preconditioners (and also of the Kruskal-based ones) to determine which of the possible implementations of the algorithms described in Section 3 were more promising. The results of these tests are described in Section 4.1 without actually reporting the tables, in order to improve the clarity and save on space. Then, in the second phase the most promising Prim-based preconditioners were tested against the Kruskal-based ones on the full set of instances; the results are reported and discussed in detail in Section 4.2.

4.1 Preliminary experiments

In the preliminary phase, we tested the influence of some of the main implementation choices on the efficiency of the preconditioners.

- **Choice of the priority queue.** We implemented three different priority queues for being used by the Prim-based approaches: a binary heap, an unordered list and a bucket list. We also tested the `priority_queue` data structure of the Standard Template Library. We find out that our binary heap is almost always the best choice.

- **Choice of the arc ordering.** We found out that ordering the arcs in the star of each node does not provide significantly better preconditioners than the unsorted case, while being more costly due to the extra ordering time.

- **Choice of the BCT heuristic.** The two heuristics Prim-SP and Prim-WP showed very similar results in terms of cardinality and weight of the obtained BCT. The more complex one was not significantly more costly than the simpler one, while sometimes showing slightly better improvement of the convergence rate, especially in the first iterations; however, the converse was also occasionally true.

- **Choice of the ordering method.** For Kruskal-based approaches we tested the influence of the sorting algorithms both on time and on quality of the weight of the initial MST. The results clearly showed that the approximate bucket sort is the best choice compared against exact sorting algorithms (quick-sort in STL or an exact bucket sort): the weight of the obtained spanning tree always matched the optimal weight at least

in the first six significant digits, while requiring considerably less time to compute the ordering.

Hence, at the end of the first phase we decided to stick with the binary heap. We did not order the star of the nodes, and we used the approximated bucket-sort for the Kruskal-based approaches; however, since none of Prim-SP and Prim-WP was clearly dominating the other, we had to keep testing both even in the second phase of the computational experiments.

4.2 The second phase

The aim of the second phase was to directly compare Prim-based and Kruskal-based preconditioners. A first set of results is reported in Table 1: columns "TP", "SP", and "WP" report data about the Prim-based approaches (respectively, the Tree preconditioner and the two variants of BCT preconditioner), while column "TK" reports data about the Kruskal-Tree, preconditioner. For each approach, columns "time"/"iter" report respectively the ratio between the computing time/number of PCG iterations of the specific approach and those of the "best" Kruskal-BCT preconditioner, according to [7], i.e., the variant (ii.b); thus, a number less than one indicates that the corresponding approach is more effective (for iterations) or efficient (for time) than the Kruskal-BCT preconditioner. Finally, Column "Chol" reports data (in particular, time ratio measured as for the other approaches) about solving the KKT system with a direct method; in particular, we used the efficient Cholesky factorization of the Ng-Peyton package [12]. For that approach, a number in parenthesis reports the number of nonzeroes in the Cholesky factor, which was too large to allow solution of the system; the reported sizes correspond to memory requirements of well over 1Gb for the core data of the numerical factorization only. A "*" means instead that the package even failed to compute the symbolic factorization (hence, not even the number of nonzeroes is known).

Table 1 draws a quite complex picture. On the "difficult" (cf. [7, 8]) `goto` instances, the Prim-BCT approach is competitive with both the *-Tree ones, and also with the Kruskal-BCT one on all instances except the densest ones; furthermore, there looks to be a trend whereby the Prim-BCT approach is more and more competitive (for fixed density) as the number of nodes increases. On the "easy" `net` and `grid` instances, the new approach is always better (or only very slightly worse) than Kruskal-BCT except on the smallest instances; it is *not* better than Kruskal-Tree, but, as expected from the results in [7], it is then Prim-Tree that is then competitive with Kruskal-Tree. Also, a general positive trend with respect to the size of the instance also shows up; for these "easy" classes, the trend is actually clearer when density grows than when the number of nodes increases.

As for the direct approach, it is obviously not competitive with the iterative ones, except in a few cases for the `goto` instances, even disregarding the fact that the largest graphs simply cannot be solved.

In order to gather a better understanding of the results, in Table 2 we report

		TP		SP		WP		TK		Chol
		iter	time	iter	time	iter	time	iter	time	time
goto	12_8	1.15	1.00	0.99	0.95	0.99	0.96	1.07	0.94	76.8
	12_64	1.11	0.94	0.99	0.89	0.99	0.90	1.14	1.01	30.3
	12_256	1.25	1.08	1.16	1.13	1.16	1.15	1.25	1.08	43.7
	14_8	1.08	0.81	0.97	0.84	0.94	0.83	1.12	0.92	1.0
	14_64	1.20	0.99	0.98	0.85	0.98	0.87	1.18	1.02	74.5
	14_256	1.28	1.16	1.12	1.08	1.13	1.09	1.20	1.10	(9.5e7)
	16_8	0.88	0.78	0.79	0.77	0.78	0.77	1.18	1.07	2.0
	16_64	1.15	1.01	0.84	0.78	0.84	0.78	1.26	1.14	(2.5e8)
grid	12_8	1.05	1.13	1.05	1.15	1.05	1.19	1.00	0.89	360.9
	12_64	1.00	0.66	1.06	0.95	1.07	0.97	1.00	0.67	115.8
	12_256	1.02	0.60	1.00	0.91	1.00	0.92	1.02	0.61	36.6
	14_8	0.89	0.83	0.82	0.95	0.86	1.01	1.00	0.85	2186.5
	14_64	0.86	0.66	0.92	0.93	0.86	0.89	1.00	0.78	(9.2e7)
	14_256	1.00	0.61	1.00	0.89	1.00	0.92	1.00	0.58	*
	16_8	0.98	0.90	0.93	1.01	0.93	1.02	1.00	0.91	*
	16_64	0.93	0.74	0.97	0.95	0.97	0.96	1.00	0.83	*
net	12_8	0.97	0.94	1.04	1.04	1.04	1.04	1.00	0.94	162.6
	12_64	0.99	0.55	0.99	0.74	0.99	0.75	1.00	0.68	87.9
	12_256	1.00	0.48	1.00	0.67	1.00	0.68	1.00	0.65	24.6
	14_8	0.81	0.81	0.83	0.87	0.83	0.85	1.01	1.00	147.4
	14_64	0.96	0.78	0.96	0.85	1.03	0.91	0.94	0.83	(9.5e7)
	14_256	1.00	0.46	1.00	0.62	1.00	0.65	1.00	0.59	*
	16_8	0.91	0.91	0.93	0.93	0.93	0.94	1.00	0.99	*

Table 1: Comparison of Prim-based and Kruskal-based preconditioners

some detailed data about the behavior of the different preconditioners (averaged) on (the five instances of) class 12_64. For each generator we report seven rows corresponding to the systems solved at IP iterations 1, 2, $k/4$, $k/2$, $3k/4$, $k-1$ 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 1 are those solved to find an initial interior solution, for which $\Theta = I_m$, hence $M = EE^T$. Columns "TP" and "SP" report respectively data about the Prim-based Tree and BCT preconditioner (in particular, the SP variant), while columns "TK" and "BK" report respectively data about the Kruskal-based Tree and BCT preconditioner (in particular, the variant (ii.b)). For each approach, columns "time" and "iter" reports respectively the computing time (in seconds) and the number of PCG iterations required for solving the system. For the BCT approaches, column "cR" contains the "cardinality ratio" $|S|/(n-1)-1$, i.e., the percentage of arcs (with respect to those of the spanning tree) added to the support graph, while column "wR" contains the "weight ratio" $w(S)/w(T)-1$, where $w(S)$ is the weight of the support graph while $w(T)$ is the weight of the

MST.

		TP		SP				TK		BK			
gen	IP it.	iter	time	cR	wR	iter	time	iter	time	cR	wR	iter	time
goto	1	910	8.33	.93	2e-1	662	6.25	1052	9.65	.72	1e-1	982	9.11
	2	94	0.90	.95	9e-1	86	0.87	94	0.96	.51	5e-1	94	1.03
	$k/4$	82	0.80	.85	6e-1	73	0.76	82	0.86	.56	3e-1	69	0.85
	$k/2$	64	0.65	.82	5e-1	57	0.61	64	0.69	.55	2e-1	52	0.66
	$3k/4$	46	0.48	.83	5e-1	45	0.50	46	0.51	.56	2e-1	40	0.50
	$k-1$	25	0.29	.84	5e-1	24	0.31	25	0.31	.56	2e-1	23	0.32
	k	25	0.29	.84	5e-1	24	0.31	25	0.31	.56	2e-1	23	0.32
grid	1	10	0.36	.06	5e-2	10	0.55	10	0.31	.03	3e-2	10	0.42
	2	10	0.36	.23	2e-2	10	0.53	10	0.33	.21	4e-3	10	0.49
	$k/4$	10	0.36	.23	4e-2	10	0.54	10	0.35	.22	7e-2	10	0.58
	$k/2$	15	0.41	.05	3e-2	15	0.58	15	0.45	.05	2e-2	15	0.69
	$3k/4$	16	0.42	.04	3e-2	16	0.59	16	0.48	.04	3e-2	16	0.72
	$k-1$	17	0.43	.05	3e-2	17	0.60	17	0.36	.00	1e-3	17	0.47
	k	101	1.32	.05	3e-2	136	1.87	101	1.25	.00	1e-4	101	1.36
net	1	10	0.27	.68	7e-1	10	0.39	10	0.24	.43	4e-1	10	0.32
	2	10	0.26	.05	4e-2	10	0.37	10	0.34	.05	4e-2	10	0.53
	$k/4$	11	0.28	.06	4e-2	11	0.38	11	0.35	.06	3e-2	11	0.55
	$k/2$	16	0.33	.05	3e-2	16	0.43	16	0.46	.05	2e-2	16	0.66
	$3k/4$	15	0.32	.05	3e-2	15	0.42	15	0.37	.05	3e-2	15	0.56
	$k-1$	10	0.26	.05	3e-2	10	0.36	10	0.25	.00	2e-3	10	0.30
	k	17	0.34	.05	3e-2	17	0.44	22	0.37	.00	2e-4	22	0.43

Table 2: Detailed results for 12_64 instances (time in seconds)

Table 2 clearly show that the two approaches build quite different support graphs; also, the behavior varies significantly with the class of instances. For all instances, the Prim-based BCT usually contains more arcs than the Kruskal-based one; the difference is relevant for `goto` instances, and marginal for the others, except possibly at the first and last iterations. Also, the cardinality ratio is much more uniform in the `goto`, while it varies significantly in the others. Analogously, except in one case the weight ratio of SP is better than that of BK; the difference is relevant for `goto` instances, and marginal for the others, except at the first and especially at the last iterations, where SP shows a pretty stable ratio while the ratio of BK sharply decreases. Yet, a "larger" BCT (both in weight and cardinality) is not always associated with fewer PCG iterations; for `goto` instances, it actually is on the first IP iterations but not in the subsequent ones (albeit, with an automatic switching rule it is the first iterations that are actually important, since, later on, the Tree is used anyway). So, SP being faster on `goto` instances has to be due to the combined effect of less PCG iterations (sometimes) and more efficient finding of S; this is confirmed by the fact that—barring the first iteration—TP requires exactly the same number of PCG iterations than BK, while being faster. The situation is analogous for `grid` and `net` instances, although there (as expected from the results in [7]) the BCT preconditioners almost never improve on the iterations count, and therefore end up being slower than Tree ones. Among the latters, TP is clearly

better than TK on `net` instances, while the situation is far less clear on `grid` ones.

Further elements confirming the complexity of the analysis are reported in Table 3, which shows the time spent in finding the support graph S (either a tree or a BCT) as a fraction of the total time required for solving the system. We remark that the fraction comprises both the time for finding S and that for factoring the preconditioner; the latter is however negligibile, being most often less than 1% of the total time, and never above 3%. The table shows that the time for finding the preconditioner is, in general, much less relevant for `goto` instances, because they require much more PCG iterations to be solved, while it takes a very significant fraction of the total time for the other "simpler" classes, especially as density of the graph increases. Also, BCTs require significantly more time than Trees. However, there seems to be a general trend where the fraction tends to decrease as the number of nodes increase (if the density remains constant).

	goto					grid					net				
	TP	SP	WP	TK	BK	TP	SP	WP	TK	BK	TP	SP	WP	TK	BK
12_8	.05	.10	.10	.09	.13	.39	.42	.43	.23	.31	.23	.25	.28	.19	.26
12_64	.06	.10	.10	.10	.19	.54	.67	.67	.55	.69	.44	.58	.59	.55	.68
12_256	.14	.23	.24	.13	.24	.59	.73	.73	.59	.75	.45	.61	.62	.60	.74
14_8	.08	.12	.13	.15	.23	.35	.45	.46	.28	.40	.04	.07	.07	.04	.06
14_64	.06	.09	.09	.11	.21	.38	.52	.54	.35	.52	.26	.38	.38	.34	.47
14_256	.05	.10	.11	.07	.14	.54	.68	.68	.44	.71	.43	.58	.59	.57	.74
16_8	.05	.08	.08	.07	.12	.21	.30	.31	.16	.24	.04	.07	.07	.04	.06
16_64	.03	.05	.06	.06	.15	.22	.31	.32	.18	.32					

Table 3: Fraction of total time spent in finding S

The final element to be taken into account is the effect the automatic switching rules. In fact, from [7] (and the previous results) we know that in general BCT preconditioners are seldom always better than the corresponding Tree ones across different instance classes, different sizes within the same class, and even different IP iterations for the same instance. However, it turns out that a reasonably simple rule can be used to construct a "hybrid" preconditioner which works better than either the pure BCT and the pure Tree ones; the rule just computes the "cardinality ratio" (cf. Table 2) and compares it with a fixed threshold. If the threshold is exceeded then the preconditioner actually includes those arcs, otherwise the operation is disabled in that and all the following IP iterations; this is justified 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 helping at a given iteration, it is somewhat unlikely that is it going to help later. Permanently disabling the rule is simple to implement, and has the advantage of avoiding the cost for finding a BCT (which may be very significant,

cf. Table 3) that is not going to be used.

In [7], it was reported that a threshold of 0.45 was effective for Kruskal-based preconditioners; however, from the data in Table 2 it can be expected that such a value does not carry over to Prim-based ones, since they usually find rather larger subgraphs. In order to avoid any distortion in the comparison due to the choice of the threshold, we decided to experimentally find, independently for all classes of instances, and independently for Prim-based and Kruskal-based approaches, the "best" possible value of the threshold (in increments of 5%). We also tested whether the "weight ratio" (cf. again Table 2) could be a more dependable indicator upon which basing the switching rule; again, independently for each class of instances and each approach we tested all reasonable values of the threshold (in increments of half an order of magnitude) and collected the overall best results. We found out that the weight-based switching rule may indeed provide slightly better results for Prim-based approaches than the cardinality-based one; however, since the difference was not particularly relevant, we finally decided to report results only about the latter.

This is done in Table 4; in columns "TP", "SP" and "TK" we report the ratio between the best running time obtained by the corresponding preconditioner—using the switching rule in case of SP—and that of Kruskal-BCT with "optimal" switching rule.

	`goto`			`grid`			`net`		
	TP	SP	TK	TP	SP	TK	TP	SP	TK
12_8	1.06	1.01	1.00	1.27	1.27	1.00	1.00	1.00	1.00
12_64	0.94	0.89	1.01	0.99	0.99	1.00	0.81	0.81	1.00
12_256	1.08	1.08	1.08	0.98	0.98	1.00	0.73	0.73	1.00
14_8	0.88	0.88	1.00	0.97	0.97	1.00	0.81	0.81	1.00
14_64	0.99	0.85	1.02	0.85	0.85	1.00	0.94	0.94	1.00
14_256	1.16	1.07	1.10	1.05	1.05	1.00	0.78	0.78	1.00
16_8	0.78	0.76	1.07	0.99	0.99	1.00	0.92	0.92	1.00
16_64	1.01	0.78	1.14	0.90	0.90	1.00			

Table 4: Kruskal- and Prim-based preconditioners with "optimal" threshold

The results show that even class-specific tuning of the threshold does not change the general picture. On the "easy" `grid` and `net` instances, the optimal preconditioner is almost indistinguishable from the Tree one, both in the Prim and Kruskal cases. For these instances, Prim-based preconditioners can improve upon Kruskal-based ones only if the Prim-based MST is more efficient than the Kruskal-based MST; this actually happens for all `net` instances and in the vast majority of `grid` ones. For `goto` instances, where the BCT approach instead consistently improve the quality of the preconditioner, then the Prim-based approach also provides an extra boost to performances due to the different characteristics of the support graph (larger, and with larger weight at the initial iterations); this happens for instance in the 14_64 case, where TP, TK and BK

are basically equivalent, while BK consistently improves on them. Overall, the Prim-BCT approach is competitive on almost all the `goto` instances, except the densest ones.

5 Conclusion

We have proposed and experimentally tested a new family of heuristics, based on modifications to the Prim approach, for finding large-weight subgraph based preconditioners for the solution of the KKT systems arising in the solution of Min Cost Flow problems through IP methods. For most test instances, these heuristics improve on those known in the literature; this is due to a more efficient computation of the Minimal Spanning Tree, and possibly to the different characteristics of the obtained triangulated subgraph.

Thus, our new heuristics seem to provide a valuable tool for improving the efficiency of IP algorithms for MCF problems. The obtained results further underline the importance of finding the right balance between the cost of finding and factoring a better preconditioner and the savings resulting from the decrease in PCG iterations; only when all the factors are duly taken into account an overall efficient procedure is obtained.

Acknowledgements. We thank Claudia Papa for having contributed to the formulation of a first version of the Prim heuristics. We also thank three anonymous referees for enabling us to improve the quality of the paper with their remarks.

References

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

[2] A. Borghetti, A. Frangioni, F. Lacalandra, and C.A. Nucci. Lagrangian Heuristics Based on Disaggregated Bundle Methods for Hydrothermal Unit Commitment. *IEEE Transactions on Power Systems*, 18:313–323, 2003.

[3] J. Castro. A Specialized Interior-Point Algorithm for Multicommodity Network Flows. *SIAM Journal on Optimization*, 10:852–877, 2000.

[4] T.G. Crainic, A. Frangioni, and B. Gendron. Bundle-based Relaxation Methods for Multicommodity Capacitated Fixed Charge Network Design Problems. *Discrete Applied Mathematics*, 112:73–99, 2001.

[5] A. Frangioni and G. Gallo. A Bundle Type Dual-Ascent Approach to Linear Multicommodity Min Cost Flow Problems. *INFORMS Journal on Computing*, 11(4):370–393, 1999.

[6] A. Frangioni and C. Gentile. Interior Point Methods for Network Problems. Technical Report 539, IASI - CNR, Roma, 2000.

[7] A. Frangioni and C. Gentile. New Preconditioners for KKT Systems of Network Flow Problems. *SIAM Journal on Optimization*, 14(3):894–913, 2004.

[8] A. Frangioni and A. Manca. A Computational Study of Cost Reoptimization for Min Cost Flow Problems. *INFORMS Journal on Computing*, 18(1):61–70, 2006.

[9] A. Frangioni and S. Serra Capizzano. Spectral Analysis of (Sequences of) Graph Matrices. *SIAM Journal on Matrix Analysis and Applications*, 23(2):339–348, 2001.

[10] J.J. Jùdice, J. Patricio, L.F. Portugal, M.G.C. Resende, and G. Veiga. A study of preconditioners for network interior point methods. *Computational Optimization and Applications*, 24:5–35, 2003.

[11] A. Natazon, R. Shamir, and R. Sharan. Complexity classiffication of some edge modification problems. *Discrete Applied Mathematics*, 113:109–128, 2001.

[12] E. Ng and B.W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14:1034–1056, 1993.

[13] P.M. Pardalos and M.G.C. Resende, editors. *Handbook of Applied Optimization*. Oxford University Press, 2002.

[14] L.F. Portugal, M.G.C. Resende, G. Veiga, and J.J. Jùdice. A Truncated Primal-infeasible Dual-feasible Network Interior Point Method. *Networks*, 35:91–108, 2000.

[15] M.G.C. Resende and P.M. Pardalos. Interior point algorithms for network flow problems. In J.E. Beasley, editor, *Advances in linear and integer programming*, pages 147–187. Oxford University Press, 1996.

[16] M.G.C. Resende, T. Tsuchiya, and G. Veiga. Identifying the optimal face of a network linear program with a globally convergent interior-point method. In W.W. Hager, D.W. Hearn, and P.M. Pardalos, editors, *Large-Scale Optimization: State of the Art*, pages 362–387. Kluwer Academic Publishers, 1994.

[17] M.G.C. Resende and G. Veiga. An Implementation of the dual affine scaling algorithm for minumum cost flow on bipartite uncapacited networks. *SIAM Journal on Optimization*, 3/3:516–537, 1993.

[18] D.J. Rose. Triangulated Graphs and the Elimination Process. *Journal of Mathematical Analysis and Applications*, 32:597–609, 1970.

[19] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):567–579, 1984.

Footnote List Page

[§]Department of Computer Science, University of Pisa, Corso Italia 40, 56125 Pisa (ITALY), e-mail: frangio@di.unipi.it

[*]Istituto di Analisi dei Sistemi ed Informatica del C.N.R., Viale Manzoni 30, 00185 Rome (ITALY), e-mail: gentile@iasi.cnr.it; this author has been partially supported by the UE Marie Curie Research Training Network no. 504438 ADONET