

Random-Tree Diameter and the Diameter-Constrained MST

Ayman Abdalla, Narsingh Deo, Pankaj Gupta
Department of Computer Science
University of Central Florida

{abdalla, deo, pgupta}@cs.ucf.edu

Abstract

A minimum spanning tree (MST) with a small diameter is required in numerous practical situations. It is needed, for example, in distributed mutual exclusion algorithms in order to minimize the number of messages communicated among processors per critical section. Understanding the behavior of tree diameter is useful, for example, in determining an upper bound on the expected number of links between two arbitrary documents on the World Wide Web. The Diameter-Constrained MST (DCMST) problem can be stated as follows: given an undirected, edge-weighted graph G with n nodes and a positive integer k , find a spanning tree with the smallest weight among all spanning trees of G which contain no path with more than k edges. This problem is known to be NP-complete, for all values of k ; $4 \leq k \leq (n - 2)$. In this paper, we investigate the behavior of the diameter of MST in randomly-weighted complete graphs (in Erdős-Rényi sense) and explore heuristics for the DCMST problem. For the case when the diameter bound k is small—independent of n , we present a one-time-tree-construction (OTTC) algorithm. It constructs a DCMST in a modified greedy fashion, employing a heuristic for selecting an edge to be added to the tree at each stage of the tree construction. This algorithm is fast and easily parallelizable. We also present a second algorithm that outperforms OTTC for larger values of k . It starts by generating an unconstrained MST and iteratively refines it by replacing edges, one by one, in the middle of long paths in the spanning tree until there is no path left with more than k edges. As expected, the performance of this heuristic is determined by the diameter of the unconstrained MST in the given graph. We discuss convergence, relative merits, and implementation of these heuristics on sequential and parallel machines. Our extensive empirical study shows that these two heuristics produce good solutions for a wide variety of inputs.

Keywords: tree diameter, greedy algorithm, iterative refinement, constrained minimum spanning tree.

1 Introduction

The Diameter-Constrained Minimum Spanning Tree (DCMST) problem can be stated as follows: given an undirected, edge-weighted graph G and a positive integer k , find a spanning tree with the smallest weight among all spanning trees of G which contain no path with more than k edges. The length of the longest (unweighted) path in the tree is called the *diameter* of the tree.

Garey and Johnson [10] show the DCMST problem is NP-complete by transformation from the Exact Cover by 3-Sets problem. Let n denote the number of nodes in G . The DCMST problem can be solved in polynomial time for the following four special cases: $k = 2$, $k = 3$, $k = (n - 1)$, or when all edge weights are identical.

The DCMST problem has applications in several areas, such as in distributed mutual exclusion, linear lightwave networks, and bit-compression for information retrieval. In distributed systems, where message passing is used, some algorithms use a DCMST to limit the number of passed messages. For example, Raymond's algorithm [17] imposes a logical spanning tree structure on a network of processors. Messages are passed among processors requesting entrance to a critical section and processors granting the privilege to enter. The maximum number of messages generated per critical-section execution is $2d$, where d is the diameter of the spanning tree. Therefore, a small diameter is essential for the efficiency of the algorithm. Minimizing edge weights reduces the cost of the network.

A DCMST is also useful in information retrieval, where large data structures called *bitmaps* are used in compressing large files. Bookstein and Klein [7] proposed using a spanning tree to cluster bitmap vectors and compress them, where smaller spanning-tree weight results in less storage space. However, to recover a given bitmap vector, all nodes in a path in the spanning tree must be decompressed. Therefore, the compression spanning-tree should have a small diameter for fast retrieval.

The DCMST problem also arises in linear lightwave networks (LLNs), where it is desirable to use a short spanning tree for each multi-destination transmission to minimize interference in the network. An algorithm by Bala *et al* [5] decomposes an LLN into edge-disjoint trees with at least one spanning tree by computing trees whose maximum node-degree is less than a given parameter, assuming the lines of the network are identical. If the LLN has lines of different bandwidths, lines of higher bandwidth should be included in the spanning trees to be used more often and with more traffic. Employing an algorithm that solves the DCMST problem can help find a better tree decomposition for this type of network. The network can be modeled by an edge-weighted graph, where an edge of weight $1/\beta$ is used to represent a line of bandwidth β .

Three exact algorithms for the DCMST problem, developed by Achuthan *et al* [3], used Branch-and-Bound methods to reduce the number of subproblems. The algorithms were implemented on a SUN SPARC II workstation operating at 28.5 MIPS and tested on complete graphs of different orders ($n \leq 100$). The fastest of the three algorithms with diameter bound $k = 4$ produced exact solutions in 113, 366, and 7343 seconds on average for graphs of 40, 50, and 100 nodes, respectively. Clearly, such exact algorithms with exponential time complexity are not suitable for graphs with thousands of nodes.

One special-case approximation algorithm, which computes an approximate DCMST with diameter bound $k = 6$, was proposed by Paddock [16]. The algo-

rithm is not easily generalizable. Bar-Ilan *et al* [6] presented two polynomial-time approximate algorithms for DCMST with the diameter constrained to 4 or 5. The algorithms were specifically designed to provide a logarithmic ratio approximation when the edge-weights in the input graph are the elements of an integral, non-decreasing function. They, too, are not suitable for the general DCMST problem.

There is a significant amount of literature on the height and diameter of a random tree. The expected height of a random labeled rooted tree is $(2 \pi n)^{1/2}$, as derived by Rényi and Szekeres [18]. The problem of tree-enumeration by height and diameter, for labeled and unlabeled trees, was addressed by Riordan [19] among others [12]. Szekeres [20] showed that $3.342171\sqrt{n}$ and $3.20151315\sqrt{n}$ is the expected value of diameter and the diameter of maximum probability, respectively, for a random labeled tree of order n , when n goes to infinity.

For finite values of n , we experimentally study the expected value of MST diameter in complete graphs with uniformly-distributed random edge-weights in Section 2. Then, in Sections 3 and 4, we discuss polynomially solvable exact cases of the DCMST problem and present methods to evaluate approximate solutions for the general cases. Our approximate algorithms for solving the DCMST problem employ two distinct strategies: One-Time Tree Construction (OTTC) and Iterative Refinement (IR). The OTTC algorithm, presented in Section 6, is based on Prim's algorithm and grows a tree within the desired diameter constraint. Each of the two general IR algorithms, presented in Sections 7 and 8, starts with an unconstrained minimum spanning tree, then uses edge replacements to transform it into a spanning tree satisfying the diameter constraint. We analyze the empirical data obtained from implementation of these algorithms, in Section 9.

2 An Empirical Study of Expected MST-Diameter

In a randomly-weighted complete graph, every spanning tree is equally likely to be an MST. Thus, there is a one-to-one correspondence between the set of minimum spanning trees of a randomly-weighted complete graph with n nodes and the set of all n^{n-2} labeled trees with n nodes. Therefore, the behavior of MST-diameter in a randomly-weighted complete graph can be studied using unweighted random labeled trees.

When the number of nodes is small, the expected value of labeled-tree diameter can be calculated by computing the number of trees for each possible diameter, and computing the average [19]. However, for trees with hundreds of thousands of nodes, an empirical study of the expected value of diameter is considerably faster.

We compared the average diameter in computer-generated random labeled trees to the expected value computed using Szekeres' formula. The mean diameter was computed for trees up to one million nodes, randomly generated and averaged for 100 different trees of each order. The curve fitting result for the

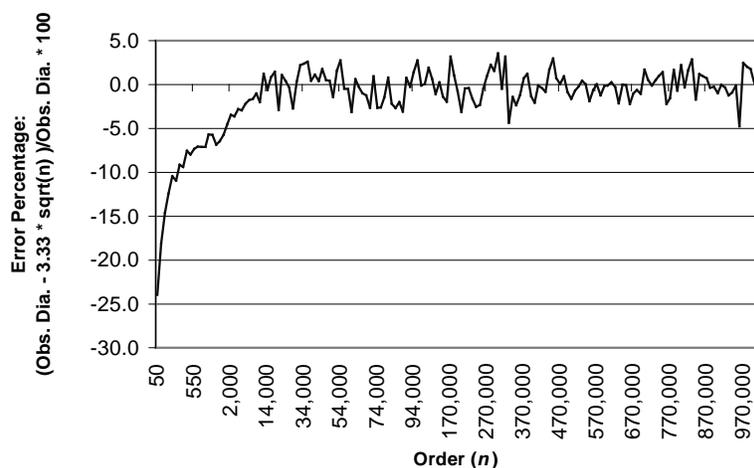


Figure 1. Percentage error in approximate diameter

diameter-means, obtained using a least-squares-fit program, was $3.33125\sqrt{n}$, showing a difference of $0.010921\sqrt{n}$ from Szekeres' formula. The curve fitting error, as illustrated in Figure 1, stabilizes for $n \geq 1100$.

To generate random spanning trees, we used a randomly generated Prüfer code [14]. Then, we tried three different methods for calculating the diameter of the trees. The first algorithm takes a naïve approach by computing the distances between all pairs of leaves in the tree, employing Warshall-Floyd all-pairs-shortest-path algorithm, in $O(n^3)$ time. The second method repeatedly removes all the leaves in the tree, until a single node or a path remains. The diameter is equal to twice the number of deletions, plus the length of the remaining path. Using a queue to keep track of the order in which leaves are deleted, this method requires linear time. In the third method, proposed by Handler [11], we perform a depth-first search (DFS) from an arbitrary node v in the tree to find the farthest node u from v . Node u will always be at one end of a diameter [21]. Then, we perform another DFS to find the farthest node w from u , which gives a diameter of the tree [11]. This method also computes the diameter in linear time. We compared the execution time of the last two linear-time algorithms, on a PC with Pentium III / 500 MHz processor, for the same set of randomly generated trees of up to one million nodes. The running time was averaged over 100 trees for

each order. As seen in Figure 2, for trees of order larger than 5000, the leaf-deletion algorithm is clearly the fastest. The C source code for both algorithms is included in our technical report [2].

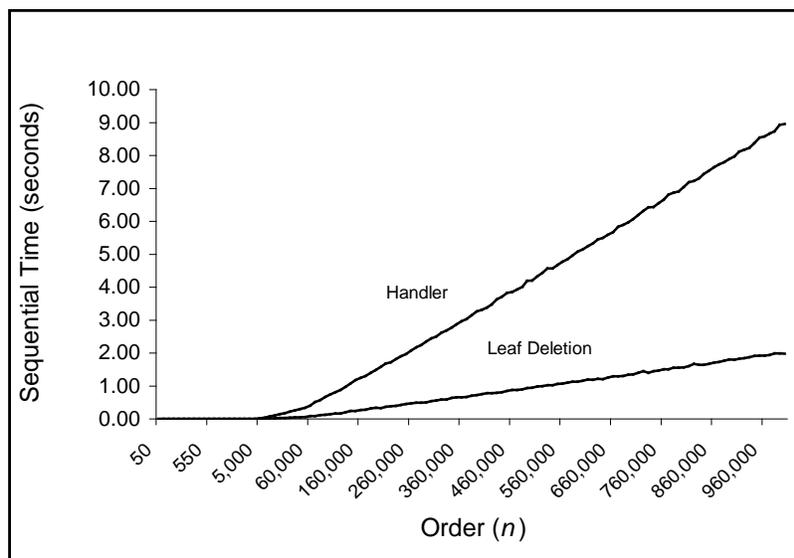


Figure 2. Comparison of speed of two diameter calculation methods

3 Polynomially-Solvable Cases

There are four cases of the DCMST problem that can be exactly solved in polynomial time. When the diameter constraint $k = n - 1$, an MST is the solution. When $k = 2$, the optimal solution is a smallest-weight star, which can be computed in $O(n^2)$ by comparing the weight of every n -node star in G . The optimal DCMST(3) can be computed in $O(n^3)$ by computing all dipolar stars and choosing one with the smallest weight as follows: Clearly, in a DCMST(3) of graph G , every node must be of degree 1 except at most two nodes, call them u and v . To construct a DCMST(3), we select an edge to be the *central edge* (u, v) , then, for every node x in G , $x \notin \{u, v\}$, we include in the spanning tree the smaller of the two edges (x, u) and (x, v) . To get an optimal DCMST(3), we compute all such spanning trees — with every edge in G as its central edge — and select the one with the smallest weight. In a graph of m edges, we have to compute m different spanning trees. Each of these trees requires $(n - 2)$ comparisons to select (x, u) or (x, v) . Therefore, the total number of comparisons required to obtain the optimal DCMST(3) is $(n - 2)m$, which becomes $O(n^3)$ in complete graphs.

Finally, for the case when all edge-weights in G are equal, a minimum diameter spanning tree can easily be constructed using breadth first search in $O(mn)$ time [4, 13].

4 Quality of a DCMST

Since the exact DCMST for large graphs cannot be determined in a reasonable time, we use the ratio of the weight of the approximate DCMST to that of the unconstrained MST as a rough measure of the quality of the solution.

To obtain a crude upper bound on the approximate DCMST(k) weight (where k is the diameter constraint), observe that DCMST(2) and DCMST(3) are feasible (but often grossly suboptimal) solutions of DCMST(k) for all $k > 3$. Since there are polynomial-time exact algorithms for DCMST(2) and DCMST(3), these solutions can be used as upper bounds for the weight of an approximate DCMST(k). In the next section, we develop a special approximate-heuristic for DCMST(4) and compare its weight that of DCMST(3) to verify that the heuristic provides a tighter upper bound and produces a lower-weight spanning tree for $k = 4$. Let $W(T)$ denote the weight of tree T . Then, clearly for any $k \geq 4$, $W(\text{MST}) \leq W(\text{DCMST}(k)) \leq W(\text{DCMST}(3)) \leq W(\text{DCMST}(2))$.

5 Special Iterative Refinement Heuristic for DCMST(4)

The special-case algorithm to compute an approximate DCMST(4) starts with an exact DCMST(3), then replaces higher-weight edges with smaller-weight edges, allowing the diameter to increase to 4. The refinement process starts by arbitrarily selecting one end node of the central edge (u, v) of DCMST(3), say u , to be the center of DCMST(4). Let $W(a, b)$ denote the weight of an edge (a, b) . For every node x adjacent to v , we attempt to obtain another tree of smaller-weight by replacing edge (v, x) with edge (x, y) , where y is adjacent to u , and $W(x, y) < W(x, v)$. Furthermore, the replacement (x, y) is an edge such that for all nodes z adjacent to u and $z \neq v$, $W(x, y) \leq W(x, z)$. If no such edge exists, we keep edge (v, x) in the tree. We use the same method to compute a second approximate DCMST(4), with v as its center. Finally, we accept the DCMST(4) with the smaller weight as the approximate solution.

Suppose there are ρ leaves adjacent to u in DCMST(3). Then, there are $(n - \rho - 2)$ leaves adjacent to v . Therefore, we make $2\rho(n - \rho - 2)$ comparisons to get an approximate DCMST(4). It can be shown that employing this procedure for a complete graph, the expected number of comparisons required to obtain an approximate DCMST(4) from an exact DCMST(3) is $(n^2 - 8n - 12)/2$.

6 One-Time Tree Construction

In the One-Time Tree Construction (OTTC) strategy, a modification of Prim's algorithm is used to compute an approximate DCMST in one pass. Prim's algo-

rithm is chosen since it has been experimentally shown to be the fastest algorithm for computing an MST for large dense graphs [15].

Our OTTC algorithm starts with one node, and grows a spanning tree by

```

procedure ModifiedPrim
INPUT: Graph  $G$ , Diameter bound  $k$ , Start node  $z_0$ 
OUTPUT: Spanning Tree  $T = (V_T, E_T)$ 
initialize  $V_T := \{z_0\}$  and  $E_T := \emptyset$ 
initialize  $near(u) := z_0$  and  $wnear(u) := w_{uz_0}$ , for every  $u \notin V_T$ 
compute a next-nearest-node  $z$  such that:
     $wnear(z) = \text{MIN}_{u \notin V_T} \{wnear(u)\}$ 

while ( $|E_T| < (n - 1)$ )
    select the node  $z$  with the smallest value of  $wnear(z)$ 
    set  $V_T := V_T \cup \{z\}$  and  $E_T := E_T \cup \{(z, near(z))\}$ 

    // 1. set  $dist(z, u)$  and  $ecc(z)$  //
    for  $u = 1$  to  $n$ 
        if  $dist(near(z), u) > -1$  then
             $dist(z, u) := 1 + dist(near(z), u)$ 
         $dist(z, z) := 0$ 
         $ecc(z) := 1 + ecc(near(z))$ 

    // 2. update  $dist(near(z), u)$  and  $ecc(near(z))$  //
     $dist(near(z), z) = 1$ 
    if  $ecc(near(z)) < 1$  then
         $ecc(near(z)) = 1$ 

    // 3. update other nodes' values of  $dist$  and  $ecc$  //
    for each tree node  $u$  other than  $z$  or  $near(z)$ 
         $dist(u, z) = 1 + dist(u, near(z))$ 
         $ecc(u) = \text{MAX}\{ecc(u), dist(u, z)\}$ 

    // 4. update the  $near$  and  $wnear$  values for other nodes in  $G$  //
    for each node  $u$  not in the tree
        if  $1 + ecc(near(u)) > k$  then
            examine all nodes in  $T$  to determine  $near(u)$  and  $wnear(u)$ 
        else
            compare  $wnear(u)$  to the weight of  $(u, z)$ .

```

Figure 3. OTTC Modified Prim algorithm

connecting the nearest neighbor that does not violate the diameter constraint. Since such an approach keeps the tree connected in every iteration, it is easy to keep track of the increase in tree-diameter. This Modified Prim algorithm is formally described in Figure 3, where we maintain the following information for each node u :

$near(u)$ is a tree node, such that for all tree nodes x adjacent to node u , $W(u, near(u)) \leq W(u, x)$.
 $wnear(u)$ is the weight of edge $(u, near(u))$.
 $dist(u, v)$ is the (unweighted) distance from u to v if u and v are in the tree, and is set to -1 if u or v is not yet in the tree.
 $ecc(u)$ is the eccentricity of node u , (the distance in the tree from u to the farthest node) if u is in the tree, and is set to -1 if u is not yet in the tree.

To update $near(u)$ and $wnear(u)$, we determine the edges that connect u to partially-formed tree T without increasing the diameter (as the first criterion) and among all such edges we want the one with minimum weight.

In Code Segment 1 of the OTTC algorithm, we set the $dist(z)$ and $ecc(z)$ values for node z by copying from its parent node $near(z)$. In Code Segment 2, we update the values of $dist(\cdot)$ and $ecc(\cdot)$ for the parent node in n steps. In Code Segment 3, we update the values of $dist(\cdot)$ and $ecc(\cdot)$ for other nodes. We make use of the $dist(\cdot)$ and $ecc(\cdot)$ arrays, as described above, to simplify the OTTC computation.

In Code Segment 4, we update the $near(\cdot)$ and $wnear(\cdot)$ values for every node not yet in the tree. If adding node z to the tree would increase the diameter beyond the constraint, we must reexamine all nodes of the tree to find a new value for $near(z)$. This can be achieved by examining $ecc(x)$ for nodes x in the tree; *i.e.*, we need not recompute the tree diameter. This computation includes adding a new node, z , to the tree, where $wnear(z)$ is minimum, and the addition does not increase the tree diameter beyond the constraint.

The complexity of Code Segment 4 is $O(n^2)$ when the diameter constraint, k , is small, since it requires looking at each node in the tree once for every node not in the tree. The *while* loop requires $(n - 1)$ iterations. Each iteration requires at most $O(n^2)$ steps, which makes the worst-case time complexity of the algorithm $O(n^3)$.

This algorithm does not *always* find a DCMST. Furthermore, the algorithm is sensitive to the node chosen for starting the spanning tree. In both the sequential and parallel implementations, we compute n such trees, one for each starting node. Then, we output the spanning tree with the largest weight.

To reduce the time needed to compute a DCMST further, we develop a heuristic that selects a small set of starting nodes for OTTC as follows. Select the q nodes (q is independent of n) with the smallest sum of weights of the edges emanating from each node. Since this is the defining criterion for trees with diameter 2, it is polynomially computable. Now, producing q spanning trees instead of n reduces the overall time complexity by a factor $O(n)$ when q is a

constant. For incomplete graphs, we can choose the q nodes with the highest degrees, and break a tie by choosing the node with the smaller sum of weights of edges emanating from it.

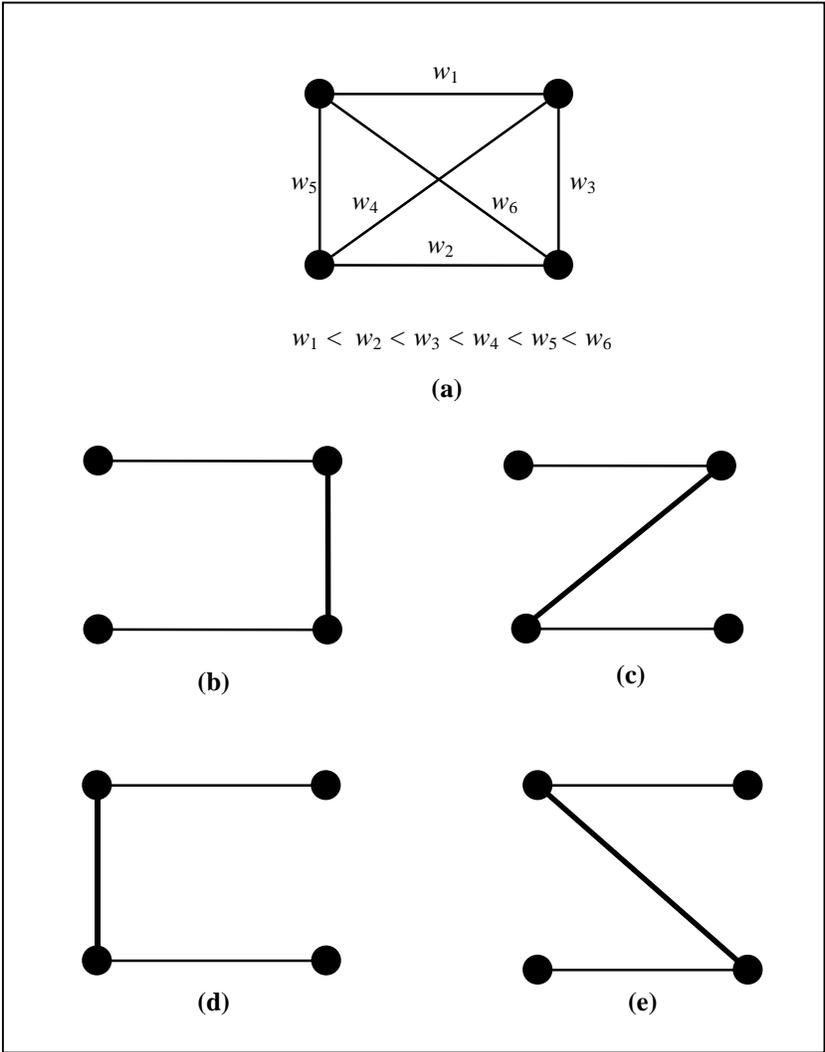


Figure 4. An example of cycling in Iterative Refinement

7 The IR1 Iterative Refinement Algorithm

The IR1 general Iterative Refinement (IR) algorithm first computes an unconstrained MST, then iteratively refines this MST by edge-replacement until the diameter constraint is satisfied.

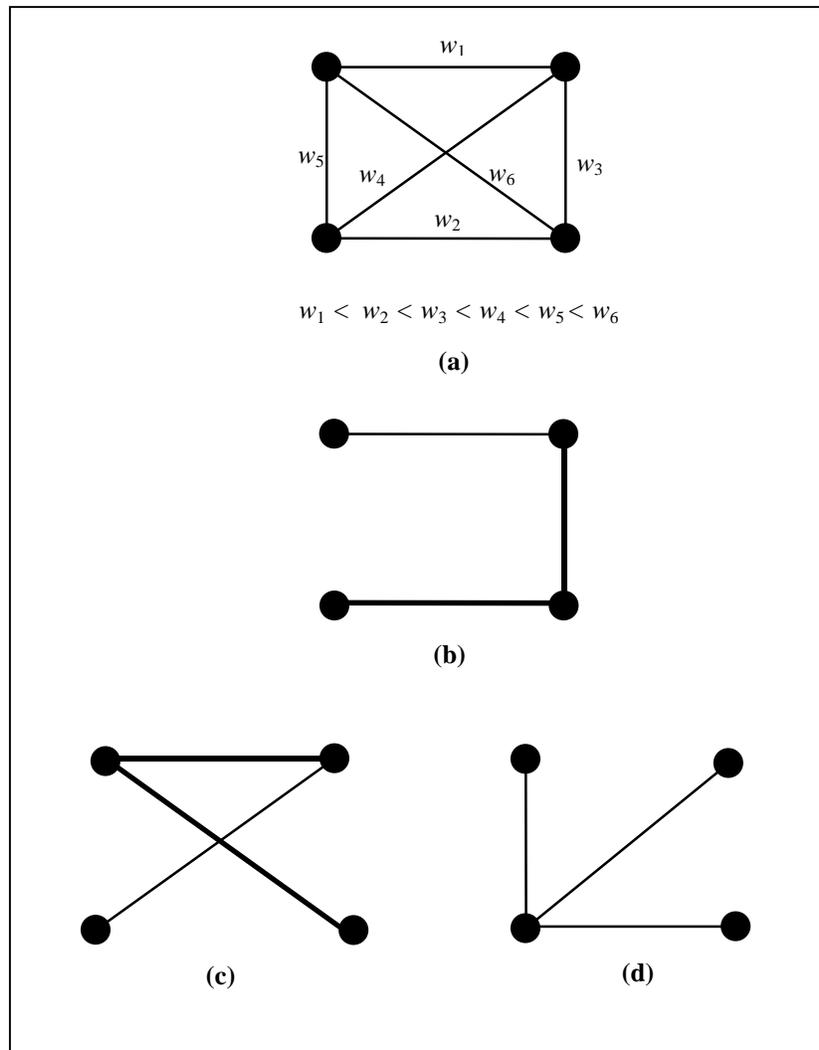


Figure 5. Finding a DCMST(2) by penalizing 2 edges per iteration

The heart of IR1 is a problem-specific *penalty function*. A penalty function succinctly encodes how many edges to penalize, which edges to penalize, and what must the penalty amount be, where the penalty is an increase in edge-weight. In each iteration of IR1, an MST of the graph with the current weights is computed, and then a subset of tree edges are penalized (using the penalty-function), so that they are discouraged from appearing in the MST in the next iteration. To reduce the diameter, this subset must contain edges belonging to long paths in the current tree, so that these long paths will be broken down into short subpaths. Obviously, an edge at the center of a long path is a good candidate to be penalized, since it would split the path into two subpaths of equal length. However, penalizing only one edge per iteration may not be sufficient, as illustrated by the example of Figure 4.

For this complete graph, shown in Figure 4(a), and a specified diameter bound of 2, the MST is the path (w_1, w_3, w_2) , shown in Figure 4(b). After penalizing the center edge, w_3 , and recomputing the MST, we get the path (w_1, w_4, w_2) , shown in Figure 4(c). The center edge w_4 in this path is penalized next, producing the path in Figure 4(d). The algorithm fails to reduce the diameter of this tree as well, producing the tree in Figure 4(e), which, in the next iteration, reproduces the MST we started with. The iterative refinement cycles among these paths of length 3, and never finds any of the four spanning trees of diameter 2.

However, if two edges are penalized in every iteration, there is no cycling for this example. The solution is found in three iterations, as shown in Figure 5. Such is the case for every edge-weighted graph with $n = 4$. However, for $n = 5$, penalizing two edges per iteration may not be sufficient.

To reduce the possibility of cycling, the number of edges to be penalized per iteration should increase with n . However, penalizing too many edges would cause the iterative refinement to jump (in a single iteration) from one tree to another, which is many edges different, thereby missing a number of solutions with perhaps smaller weight. Therefore, the number of edges penalized must be a slow-growing function of n , say $\log_2 n$. We penalize the edges incident to the center. If there are less than $\log_2 n$ edges incident to the center, the edges at distance two from the center are chosen, and so on. A tie can be broken by choosing the higher-weight edge to penalize.

To be effective without causing overflow, the penalty value must relate to the range of the weights in the spanning tree. Let $w(l)$ denote the current weight of an edge l being penalized, and w_{\max} and w_{\min} denote the largest and the smallest edge-weight, respectively, in the current MST. Also, let $distc(l)$ denote the distance of an edge l from the center node, plus one. When the center is an edge l_c , it has $distc(l_c) = 1$, an edge l incident to only one end-point of the center edge has $distc(l) = 2$, and so on. Therefore, the penalty amount imposed on a tree edge l is given by:

$$MAX \left\{ \frac{(w(l) - w_{\min}) w_{\max}}{distc(l) (w_{\max} - w_{\min})}, \epsilon \right\},$$

where $\varepsilon > 0$ is a minimum penalty that ensures the iterative refinement will not stay at the same spanning tree by imposing zero penalties on all edges.

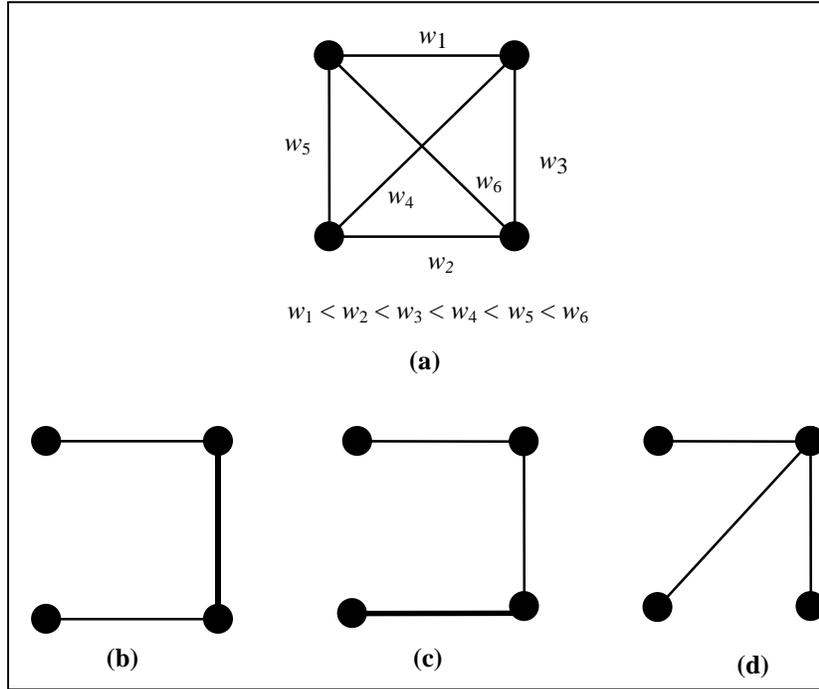


Figure 6. Example of IR2

The penalty amount decreases as the penalized edge becomes farther away from the center of the current MST, so that a long path will be broken into two significantly shorter subpaths, rather than a short subpath and a long one. When two edges have the same value of $distc(\cdot)$, the edge with larger weight is penalized by a larger amount to discourage it from appearing in the next MST.

One problem with this approach is that it recomputes the MST in every iteration, which sometimes reproduces spanning trees that were already examined, even when the replacement increases the diameter. When implemented, this algorithm succeeded in avoiding cycling when the diameter constraint $k \geq 0.1n$, but failed to find an approximate DCMST(k) when k is a small constant. We present a different IR algorithm in the next section that avoids the cycling problem, and produces solutions with smaller values of k .

8 The IR2 Iterative Refinement Algorithm

The next iterative refinement algorithm, IR2, does not recompute the MST in every iteration; rather, a new spanning tree is computed by modifying the

```

procedure IR2
INPUT: Graph  $G = (V, E)$ , diameter bound  $k$ 
OUTPUT: Spanning tree  $T$  with diameter  $\leq k$ 
compute MST and  $\text{ecc}_T(z)$  for all  $z$  in  $V$ 
 $C := \emptyset$ 
 $\text{move} := \text{false}$ 
repeat
   $\text{diameter} := \text{MAX}_{z \in V} \{\text{ecc}_T(z)\}$ 
  if  $C = \emptyset$  then
    if  $\text{move} = \text{true}$  then
       $\text{move} := \text{false}$ 
       $C :=$  edges  $(u, z)$  that are one edge farther from
        the center of  $T$  than in the previous iteration
    else
       $C :=$  edges  $(u, z)$  at the center of  $T$ 
  repeat
     $(x, y) :=$  highest weight edge in  $C$ 
    // This splits  $T$  into two trees:  $\text{subtree1}$  and  $\text{subtree2}$  //
  until  $C = \emptyset$  OR  $\text{MAX}_{u \in \text{subtree1}} \{\text{ecc}_T(u)\} = \text{MAX}_{z \in \text{subtree2}} \{\text{ecc}_T(z)\}$ 
  if  $C = \emptyset$  then // no good edge to remove was found //
     $\text{move} := \text{true}$ 
  else
    remove  $(x, y)$  from  $T$ 
    get a replacement edge and add it to  $T$ 
    recompute  $\text{ecc}_T$  values
  until diameter  $\leq k$  OR we are removing edges farthest from the
    center of  $T$ 

```

Figure 7. The IR2 Iterative Refinement algorithm

previously computed one. The modification performed does not regenerate previously-generated trees and it guarantees the algorithm will terminate. Unlike IR1, this algorithm removes one edge at a time and prevents cycling by moving away from the center of the tree whenever cycling becomes imminent. Figure 6 illustrates how this technique prevents cycling for the original graph of Figure 4. After computing the MST, the algorithm considers the middle edge (shown in bold) as the candidate for removal, as in Figure 6(b). But this edge does not have a replacement that can reduce the diameter, so we consider edges a little farther away from the center of the tree. The edge shown in bold in Figure 6(c) is the highest-weight such edge. As seen in Figure 6(d), we are able to replace it by another edge, and that reduces the diameter.

Algorithm IR2 starts by computing the unconstrained MST for the input graph, $G = (V, E)$ using Prim's algorithm. The initial eccentricity values for all

nodes in the MST can be computed using a preorder tree traversal where each node visit consists of computing the distances from that node to all other nodes in the spanning tree. This requires a total of $O(n^2)$ computations. As the spanning tree changes, we only recompute the eccentricity values that change. After computing the MST and the initial eccentricity values, the algorithm iteratively identifies one edge to remove from the spanning tree and replaces it by another edge from G until the diameter constraint is met or the algorithm fails. When implemented and executed on a variety of inputs, we found that this process required no more than $(n + 20)$ iterations. Each iteration consists of two parts. In the first part, described in Subsection 8.1, we find an edge whose removal can contribute to reducing the diameter, and in the second part, described in Subsection 8.2, we find a good replacement edge. The IR algorithm is shown in Figure 7, and its edge-replacement subprocedure is shown in Figure 8. We use $ecc_T(u)$ to denote the eccentricity of node u with respect to tree T ; the maximum distance from u to any other node in T . The diameter of a tree T is given by $MAX\{ecc_T(u)\}$ over all nodes u in T .

8.1 Selecting Edges for Removal

To reduce the diameter, the edge removed must break a longest path in the tree and should be near the center of the tree. The center of tree T can be found by identifying the nodes u in T with $ecc_T(u) = \lceil \text{diameter}/2 \rceil$, the node (or two nodes) with minimum eccentricity. The edges candidate for removal are kept in a sorted list, call it C , implemented as a max-heap and sorted according to edge weights, so that the highest-weight candidate edge is at the root.

Removing an edge from a tree does not guarantee breaking all longest paths in the tree. Therefore, we must verify that removing an edge splits the spanning tree, T , into two subtrees, $subtree1$ and $subtree2$, such that each of the two subtrees contains a node v with $ecc_T(v)$ equal to the diameter of T , *i.e.*, an end point of a longest path in T . If the highest-weight edge from list C does not satisfy this condition, we remove it from list C and consider the next highest. This process continues until we either find an edge that breaks a longest path in T or the list C becomes empty.

If list C does not contain an edge good for removal, we must consider edges farther from the center. This is done by identifying the nodes u with $ecc_T(u) = \lceil \text{diameter}/2 \rceil + \text{bias}$, where bias is initialized to zero, and incremented by 1 when list C contains no edges good for removal. Then, list C is recomputed as all the edges incident to set of nodes u . Every time the algorithm succeeds in finding an edge to remove, the bias is reset to zero.

This method of examining edges helps prevent cycling since it considers a different edge every time until an edge that can be removed is found. But to guarantee the prevention of cycling, always select a replacement edge that reduces the length of a path in spanning tree T . This will guarantee that the refinement process will terminate, since we will either reduce the diameter below

the bound k , or bias will become so large that we try to remove the edges incident to the end-points of the longest paths in T .

In the worst case, computing list C requires examining many edges in T , requiring $O(n)$ comparisons. In addition, sorting C will take $O(n \log n)$ time. A replacement edge that helps reduce the diameter is found in $O(n^2)$ time since it requires recomputing eccentricity values for all nodes. Therefore, the iterative process, which removes and replaces edges for n iterations, will take $O(n^3)$ time in the worst case. Since list C has to be sorted every time it is computed, the execution time can be reduced by a constant factor if we prevent C from becoming too large. This is achieved by an edge-replacement method that keeps the spanning tree, T , fairly uniform so that it has a small number of edges near the center, as we will show in the next subsection. Since list C is constructed from edges near the center of T , this will keep C small.

```

procedure ERM
  recompute  $ecc_{subtree1}$  and  $ecc_{subtree2}$  for all nodes in each subtree
   $m_1 := ecc_{subtree1}(x)$ 
   $m_2 := ecc_{subtree2}(y)$ 
   $(a,b) :=$  minimum-weight edge in  $G$  that has:
     $a \in subtree1$  AND  $b \in subtree2$  AND  $ecc_{subtree1}(a) \leq m_1$ 
    AND  $ecc_{subtree2}(b) \leq m_2$  AND
     $(ecc_{subtree1}(a) < m_1$  OR  $ecc_{subtree2}(b) < m_2)$ 
  if such an edge  $(a, b)$  is found then
    add edge  $(a, b)$  to  $T$ 
  else
    add the removed edge  $(x, y)$  back to  $T$ 
   $move := true$ 

```

Figure 8. The edge-replacement method, ERM

8.2 Selecting a Replacement Edge

When we remove an edge from a tree T , we split T into two subtrees: $subtree1$ and $subtree2$. Then, we select a non-tree edge to connect the two subtrees in a way that reduces the length of at least one longest path in T without increasing the diameter. The diameter of T will be reduced when all longest paths in T have been so broken. We develop method ERM to find such replacement edges. This method, shown in Figure 8, computes $ecc_{subtree1}(\cdot)$ and $ecc_{subtree2}(\cdot)$ values for each subtree individually. Then, the two subtrees are joined as follows. Let the removed edge (x, y) have $x \in subtree1$ and $y \in subtree2$. The replacement edge will be the smallest-weight edge (a, b) which (1) guarantees that the new edge does not increase the diameter of the current spanning tree, T , and (2) guar-

antees reducing the length of a longest path in T at least by one. We enforce condition (1) by:

$$ecc_{\text{subtree1}}(a) \leq ecc_{\text{subtree1}}(x) \text{ AND } ecc_{\text{subtree2}}(b) \leq ecc_{\text{subtree2}}(y),$$

and condition (2) by:

$$ecc_{\text{subtree1}}(a) < ecc_{\text{subtree1}}(x) \text{ OR } ecc_{\text{subtree2}}(b) < ecc_{\text{subtree2}}(y).$$

If no such edge (a, b) is found, we must remove an edge farther from the center of the tree, instead. Since the replacement is selected from a large set of edges, ERM produces DCMSTs with reasonably small weights without creating nodes of high degree near the center.

9 Implementation

In this section, we present empirical results obtained by implementing the OTTC and IR algorithms on the MasPar MP-1, a massively-parallel SIMD machine of 8192 processors. The processors are arranged in a mesh where each processor is connected to its eight neighbors. The source code for all algorithms implemented, written in MPL-C, is included in the technical report [2].

Complete graphs, K_n , represented by their $(n \times n)$ weight matrices, were used as input. An incomplete graph can be viewed as a complete graph in which the missing edges have infinite weight. Since the MST of a randomly generated graph has a small diameter, $O(\sqrt{n})$, it is not suited for studying the performance of DCMST algorithms. Therefore, we generated graphs in which the minimum spanning trees are forced to have diameter $(n - 1)$. The edge-weights were non-negative numbers, randomly chosen with equal probability, following the Erdős-Rényi model [9].

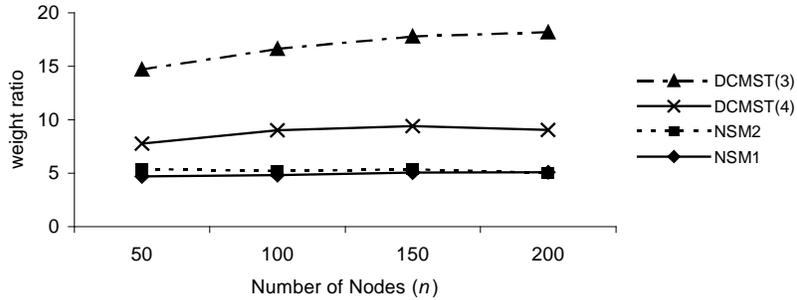


Figure 9. Weight of DCMST(5), obtained using two different node-search heuristics, as a multiple of MST weight. Initial diameter = $n - 1$

9.1 One-Time Tree Construction

We parallelized the OTTC algorithm and implemented it on the MasPar MP-1 for graphs of up to 1000 nodes. The approximate DCMST generated from one start node for a graph of 1000 nodes took roughly 71 seconds, which means it would take it about 20 hours to run with n start nodes. We address this issue by running the algorithm for a carefully selected small set of start nodes.

We used two different methods to choose the set of start nodes. One node selection method, NSM1, selects the center nodes of the q smallest stars in G as start nodes. The other method, NSM2, selects q nodes from G at random. As seen in Figure 9, the weight quality of DCMST obtained from these two heuristics, where we chose $q = 5$, is similar. The execution times of these two heuristics were also almost identical.

The results from running the OTTC algorithm with n start nodes were obtained for graphs of 50 nodes and compared with the results obtained with 5 start nodes for the same graphs; for $k = 4, 5$, and 10. The results compare the average value of the smallest weight found from NSM1 and NSM2 to the average weight found from the OTTC algorithm that runs for n iterations. The quotient of these values is reported. For $k = 4$, the approximate DCMST obtained using NSM1 had weight of 1.077 times the weight from the n -iteration OTTC algorithm. The weight of NSM2-tree was 1.2 times that of the n -iteration tree. For $k = 5$, NSM1 weight-ratio was 1.081 while NSM2 weight-ratio was 1.15. For $k = 10$, NSM1 weight-ratio was 1.053 while NSM2 weight-ratio was 1.085. In these cases, NSM1 outperforms NSM2 in terms of the weight of solutions, in some cases by as much as 12%. The results obtained confirm the theoretical analysis that predicted an improvement of $O(n)$ in execution time over the n -iteration algorithm, as described in Section 6.

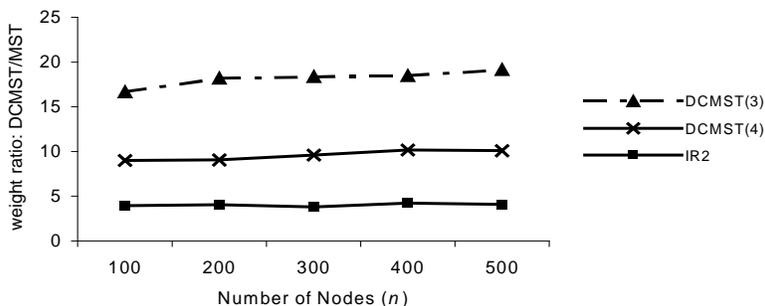


Figure 10. Weight of DCMST(10), obtained using IR2, as a multiple of MST weight. Initial diameter = $n - 1$

9.2 The Iterative Refinement Algorithms

The special IR heuristic for DCMST(4) was parallelized and implemented on the MasPar MP-1. It produced approximate DCMST(4) with weight approximately half that of exact DCMST(3), as we see in Figures 9, 10, and 12. The time to refine DCMST(3) into approximate DCMST(4) was about 1% of the time to calculate DCMST(3).

The general IR algorithms were also parallelized and implemented on the MasPar MP-1. We ran the code for IR1 on random graphs with 1000 nodes, whose minimum spanning trees are forced to be Hamiltonian paths, for $k = 500, 100, \text{ and } 60$. The upper bounds and tree weights resulting from IR1 are reported as factors of the Minimum Spanning Tree weight. The results are as follows: the range of upper bounds is $[16.9, 19.0]$ with a mean of 18.4 and a standard deviation of 8.84 over all runs, while the range of tree weights resulting from IR1 is $[1.001, 1.008]$ with a mean of 1.004 and a standard deviation of 0.0024. This indicates remarkable performance of IR1 when the diameter constraint is a large fraction of the number of nodes. The algorithm was also fast, as it reduced the diameter of a 3000-node complete graph from 2999 to 103 in about 15 minutes. Nonetheless, this IR algorithm was unable to obtain approximate DCMST(k) when k was a small fraction ($< 5\%$) of the number of nodes. It should be used only for large values of k .

As expected, IR2 did not enter an infinite loop, and it always terminated within $(n + 20)$ iterations. The algorithm was unable to find a spanning tree with diameter less than 12 in some cases for graphs with more than 300 nodes. In graphs of 400, 500, 1000, and 1500 nodes, our empirical results show a failure rate of less than 20%. The algorithm was 100% successful in finding an approximate DCMST with $k = 15$ for graphs of up to 1500 nodes. This indicates that the failure rate of the algorithm does not depend on what fraction of n the value of k is. Rather, it depends on how small the constant k is.

To see this, note that the algorithm will fail only when we try to remove edges incident to the end-points of the longest paths in the spanning tree. Also note that IR2 moves away from the center of the spanning tree every time it goes through the entire list C without finding a good replacement edge, and returns to the center of the spanning tree every time it succeeds. Thus, the only way this algorithm fails is when it is unable to find a good replacement edge in $\lceil \text{diameter}/2 \rceil$ consecutive attempts, each of which includes going through a different list C . Empirical results show that it is unlikely that the algorithm will fail for 8 consecutive times, which makes it suitable for finding approximate DCMST(k), for constant $k \geq 15$. The algorithm still performs fairly well with $k = 10$, and we did use that data in our analysis, where we excluded the few cases in which the algorithm did not achieve diameter 10. This exclusion should not affect the analysis, since the excluded cases all achieved diameter less than 15 with approximately the same speed as the successful attempts.

The weight quality of approximate DCMST(10) obtained by IR2 is shown in Figure 10. The diagram shows the weight of the computed approximate

DCMST(10) as a multiple of the weight of the unconstrained MST. The time taken by IR2 to obtain approximate DCMST(10) is shown in Figure 11. It can be seen from these figures that IR2 can achieve feasible solutions with diameter 1% of MST diameter. This is significantly smaller than the diameters of solutions obtained with IR1.

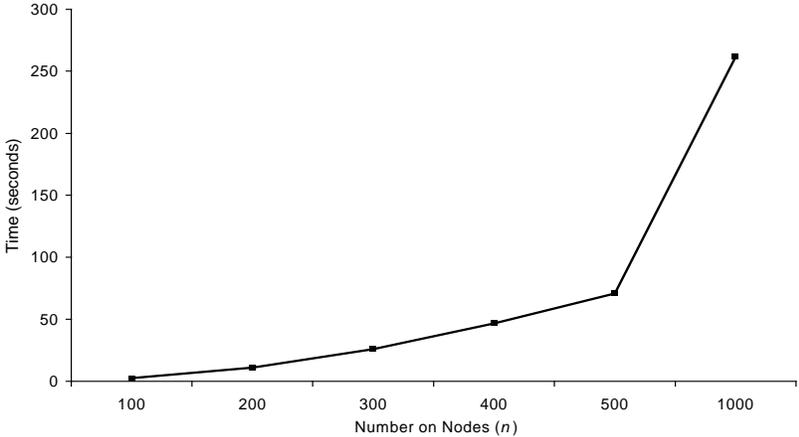


Figure 11. Time to obtain DCMST(10) using IR2. Initial diameter = $n - 1$

We tested IR2 on random graphs. The weight quality of the approximate DCMSTs obtained are charted in Figure 12, where approximate DCMST(4) was obtained using the special IR heuristic explained in Section 5. Comparing this figure with those obtained for the randomly-generated graphs forced to have unconstrained MST with diameter $(n - 1)$, it can be seen that the weight quality of approximate DCMST(10) in the graphs starting with MSTs of $(n - 1)$ diameter is better than that in unrestricted random graphs. This is because IR2 keeps removing edges close to the center of the constrained spanning tree, which contain more low-weight edges in unrestricted random graphs, coming from the unconstrained MST. But when the unconstrained MST has diameter $(n - 1)$, there are more heavy-weight edges near the center that were added in some earlier iterations of the algorithm. Therefore, the approximate DCMST for this type of graphs will lose less low-weight edges than in unrestricted random graphs.

Furthermore, the weight of approximate DCMST(10) was higher than that of approximate DCMST(4) in unrestricted random graphs, but it was lower than the weight of exact DCMST(3). Note that the approximate DCMST(4) heuristic approaches the diameter optimization from above, rather than from below. When the diameter constraint is small, it becomes more difficult for the general IR algorithms to find a solution and they allow large increases in tree-weight in order to achieve the required diameter. The approach from the upper bound,

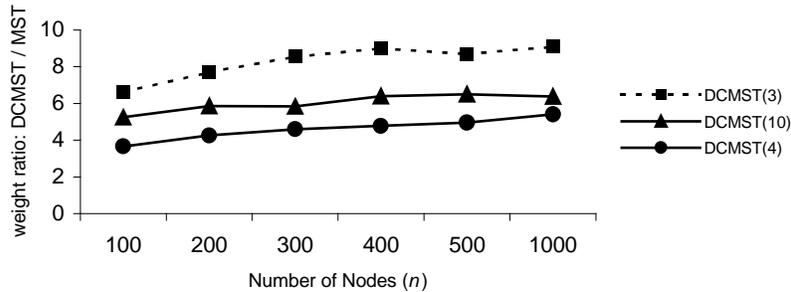


Figure 12. Quality of DCMST(4) heuristic and IR2 for unrestricted random graphs

however, guarantees the tree weight will not increase during the refinement process. When tested on unrestricted random graphs, the DCMST(4) algorithm produced approximate solutions with about half the weight of DCMST(3), as it did with randomly-generated graphs forced to have a Hamiltonian-path MST. However, the weight quality of approximate DCMST(10) produced by IR2 deteriorated in unrestricted random graphs, exceeding the weight of approximate DCMST(4) produced by the special-case algorithm. Clearly, the DCMST(4) algorithm provides a better solution for this type of graphs.

10 Conclusion

It is evident that the DCMST has various practical applications. We presented an empirical study of the behavior of MST-diameter in randomly generated graphs. Then, we developed four approximate algorithms for the DCMST problem. The special-case IR DCMST(4) algorithm is used in providing an upper bound on the weight of approximate solutions, but it is also the algorithm of choice for obtaining an approximate DCMST(4). The OTTC algorithm is based on Prim's algorithm, and is best suited for small values of the diameter constraint, k . When the diameter constraint is a large fraction of n , IR1 provides the smallest-weight approximate solutions in a short amount of time. IR2 is faster than OTTC, and it provides low-weight solutions when k is a constant larger than 15.

11 References

- [1] Abdalla, A., Deo, N., Franceschini, R., "Parallel heuristics for the diameter-constrained MST problem," *Congressus Numerantium*, 136 (1999), pp. 97-118.
- [2] Abdalla, A., Deo, N., Gupta, P., "Random-Tree Diameter and the Diameter-Constrained MST," *Technical Report CS-TR-00-02*, University of Central Florida, Orlando, FL, 2000.
- [3] Achuthan, N.R., Caccetta, L., Caccetta, P., Geelen J.F., "Computational methods for the diameter restricted minimum weight spanning tree problem," *Australasian Journal of Combinatorics*, 10 (1994), pp. 51-71.
- [4] Ahuja, R.K., Magnanti, T.L., Orlin, J.B., *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Upper Saddle River, NJ, 1993, pp. 79, 90.
- [5] Bala, K., Petropoulos, K., Stern, T.E., "Multicasting in a Linear Lightwave Network," *IEEE INFOCOM '93*, 3 (1993), pp. 1350-1358.
- [6] Bar-Ilan, J., Kortsarz, G., Peleg, D., "Generalized submodular cover problems and applications," In *Proceedings of the 4th Israel Symposium on Computing and Systems*, 1996, pp. 110-118.
- [7] Bookstein, A., Klein, S.T., "Compression of correlated bit-vectors," *Information Systems*, 16(4) (1991), pp. 387-400.
- [8] Deo, N., Kumar, N., "Constrained Spanning Tree Problems: Approximate Methods and Parallel Computation," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 40 (1998), pp. 191-217.
- [9] Erdős, P., Rényi, A., "On the evolution of Random Graphs," *Publication of Mathematical Institute of the Hungarian Academy of Sciences*, 5 (1960), pp. 17-61.
- [10] Garey, M.R., Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco. 1979.
- [11] Handler, G.Y., "Minimax location of a facility in an undirected tree graph," *Transportation Science*, 7 (1973), pp. 287-293.
- [12] Harary, F., Prins, G., "The number of homeomorphically irreducible trees, and other species," *Acta Math*, 101 (1959), pp. 141-162.
- [13] Horowitz, E., Sahni, S., *Fundamentals of Computer Algorithms*. Computer Science Press, Potomac, MD, 1978, pp. 317.
- [14] Kumar, V., Deo, N., Kumar, N., "Parallel generation of random trees and connected graphs," *Congressus Numerantium*, 130 (1998), pp. 7-18.
- [15] Moret, B.M.E., Shapiro, H.D., "An empirical analysis of algorithms for constructing a minimum spanning tree," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15 (1994), pp. 99-117.
- [16] Paddock, P.W., *Bounded Diameter Minimum Spanning Tree Problem*, M.S. Thesis, George Mason University, Fairfax, VA, 1984.
- [17] Raymond, K., "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems*, 7(1) (1989), pp. 61-77.

- [18] Rényi, A., Szekeres, G., "On the height of trees," *Journal of the Australian Mathematical Society*, 7 (1967), pp. 497-507.
- [19] Riordan, J., "The enumeration of trees by height and diameter," *IBM Journal of Research and Development*, 4 (1960), pp. 473-478.
- [20] Szekeres, G., "Distribution of labelled trees by diameter," *Lecture Notes in Math.*, 1036 (1983), pp. 392-397.
- [21] Wall, D.W., Owicki, S.S., "Center-based broadcasting," *Technical Report No. 189, Computer Systems Laboratory, Stanford University*, 1980.