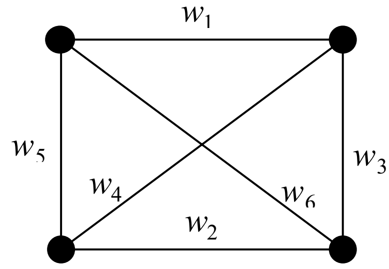
CHAPTER 4

THE IR1 ITERATIVE-REFINEMENT ALGORITHM

Our three general iterative-refinement-strategy-algorithms first compute an unconstrained MST, and then iteratively refine this MST by edge-replacement until the diameter constraint is satisfied. General iterative-refinement-algorithm IR1, which we present in this chapter, iteratively penalizes the edges near the center of the MST by increasing their weight and then recomputes the MST. This attempts to lower the diameter by breaking up long paths from the middle, replacing them by shorter ones.

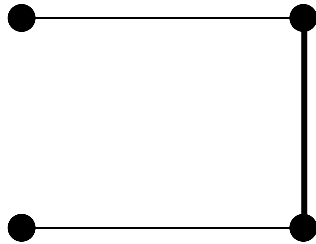
4.1 The Algorithm

The heart of Algorithm IR1 is a problem-specific *penalty function*. A penalty function succinctly encodes how many edges to penalize, which edges to penalize, and what the penalty amount must be, where the penalty is an increase in edge weight. In each iteration of IR1, as described in Algorithm 1, 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.

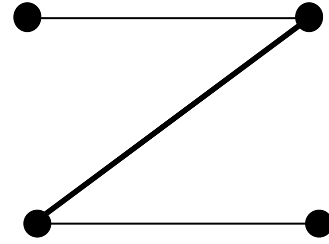


$$w_1 < w_2 < w_3 < w_4 < w_5 < w_6$$

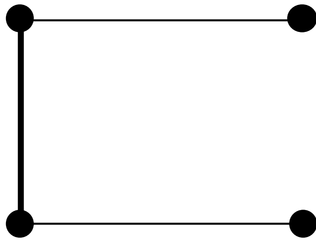
(a)



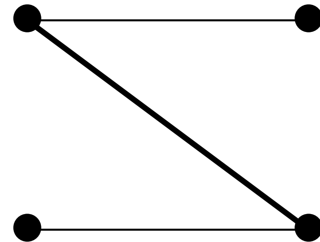
(b)



(c)



(d)



(e)

Figure 4.1 An example of cycling in IR1

Obviously, an edge at the center of a long path is a good candidate to be penalized, since

it would split each of the longest paths in the current tree into two subpaths of equal length. However, penalizing only one edge per iteration may not be sufficient, as illustrated by the example of Figure 4.1.

ALGORITHM 1 (IR1(G, k)).

```
begin
    fails := 0;
    G' := G;
    T_min := MST of G;
    T := T_min;
    while (((diameter of T) > k) and (fails ≤ 15)) do
        G' := G' with edges closest to the center of T_min penalized;
        T_min := MST of G' ;                /* computed using the new edge-weights */
        if (((diameter of T_min) < (diameter of T))
              or (((diameter of T_min) = (diameter of T)) and (W(T_min) < W(T))))
            then  begin
                      T := T_min;
                      fails := 0;
                  end
            else
                fails := fails + 1;
    end while
    return T
end.
```

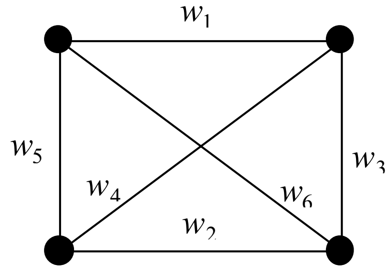
For this complete graph and a specified diameter bound of 2, the MST is the path (w_1, w_3, w_2), shown in Figure 4.1(b). After penalizing the center edge, w_3, and recomputing the MST, we get the path (w_1, w_4, w_2), shown in Figure 4.1(c). The center edge w_4 on this path is penalized next, producing the path in Figure 4.1(d). The algorithm fails to reduce the diameter of this tree as well, producing the tree in Figure 4.1(e), which, in the next iteration, regenerates the original MST. The iterative refinement cycles

among these paths of diameter 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 4.2. Such is the case for every edge-weighted graph with $n = 4$. But for $n = 5$, penalizing two edges per iteration may not be sufficient.

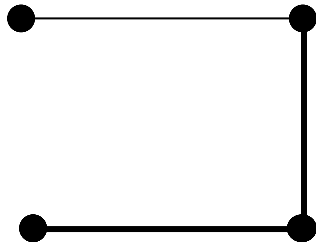
To reduce the possibility of cycling, the number of edges to be penalized per iteration should increase with n. However, it must be kept in mind that penalizing too many edges may result in the solution being too far from optimal. This is because in the space of all n^{n-2} labeled spanning trees, the iterative refinement in such a case would jump (in a single iteration) from one tree to another, which is many edges different, thereby missing a number of feasible solutions with perhaps smaller weight. Therefore, the number of edges penalized must be a slow-growing function of n, say $\log_2 n$. All the edges penalized should be close to the center of the current spanning tree where the center of a tree consists either of one node or one edge, depending on whether its diameter is even or odd. The edges to be penalized should be the ones incident to the center. If more edges are required to be penalized (when the degree of the center node is less than $\log_2 n$), then the edges at distance two from the center node should be chosen, and so on. A tie can be broken by choosing the higher-weight edge to penalize.

Another issue to consider in designing a penalty function is the penalty amount. 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

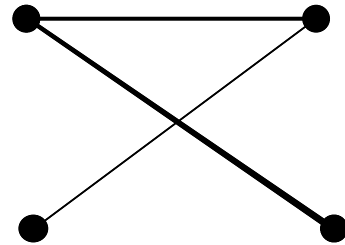


$w_1 < w_2 < w_3 < w_4 < w_5 < w_6$

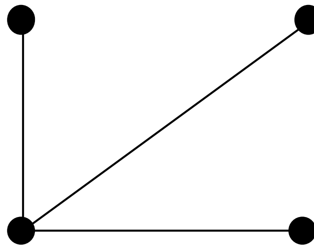
(a)



(b)



(c)



(d)

Figure 4.2 Finding an approximate DCMST(2) by penalizing 2 edges per iteration

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 a unique node, v_c, all the edges l incident to v_c have $distc(l) = 1$, the ones at distance one from v_c have $distc(l) = 2$, and so on. 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 the tree edge l is given by:

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

where $\varepsilon > 0$ is a minimum mandatory penalty imposed on an edge, chosen to be penalized. This minimum penalty ensures that the iterative refinement makes progress in every iteration, and does not stay at the same spanning tree by imposing zero penalties to all the edges (in situations, for example, when all the penalized edges have weights equal to $w_{\min}$). In a typical implementation, in which weights are stored as integer values, the value of ε may be set to 1.

Clearly, the penalty amount is proportional to the weight of the penalized edge and inverse-proportional to its distance from the center of the current MST. The penalty amount can be as high as $w_{\max}/distc(l)$, and it decreases as the penalized edge becomes farther away from the center of the tree. This was done because replacing an edge with a small $distc(l)$ in the current tree can break a long path into two significantly shorter subpaths, rather than a short subpath and a long one. Also, an edge with a smaller weight is penalized by a smaller amount than one with a larger weight if they have the same

value of *distc*(.) to makes it less likely for the larger-weight edge to appear in the next MST.

4.2 Implementation

We parallelized Algorithm IR1 and implemented it on the MasPar MP-1. We ran the code for IR1 on random graphs with up to 3000 nodes, whose minimum spanning trees are forced to be Hamiltonian paths, and whose edge weights were randomly selected numbers between 1 and 1000. The tree weights resulting from IR1 are reported as factors of the unconstrained MST weight. The average constrained spanning-tree weights with diameter $n/10$ were 1.068, 1.036, and 1.024 for n = 1000, 2000, and 3000, respectively. This indicates remarkable performance of this iterative-refinement algorithm 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 iterative-refinement algorithm was not able to obtain approximate DCMST(k) when k is a small fraction of the number of nodes, such as $n/20$. Thus, it should be used only for large values of k.

4.3 Convergence

One problem with the approach of Algorithm IR1 is that it recomputes the MST in every iteration, which sometimes reproduces trees that were already examined, even when the replacement increases the diameter. Algorithm IR1 terminates when the current MST diameter is no more than k, or when it cannot improve the current MST further. The latter case is identified by 15 consecutive iterations that reduce neither the diameter nor the weight of the current MST. Our empirical study showed that allowing IR1 to continue past 15 consecutive non-improving iterations did not result in better solutions when the edge weights ranged from 1 to 10000. When it was allowed to run for 500 iterations (regardless of non-improving iterations), Algorithm IR1 succeeded in finding a solution when the diameter constraint $k \geq n/10$, but failed to find a DCMST when k was a small constant. We present a different iterative-refinement algorithm in the next chapter that avoids the cycling problem, and produces solutions with smaller values of k.

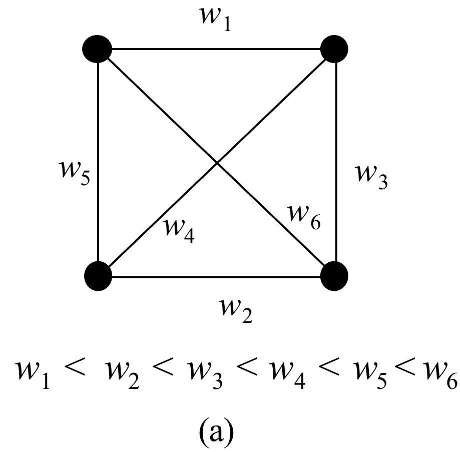
CHAPTER 5

THE IR2 ITERATIVE-REFINEMENT ALGORITHM

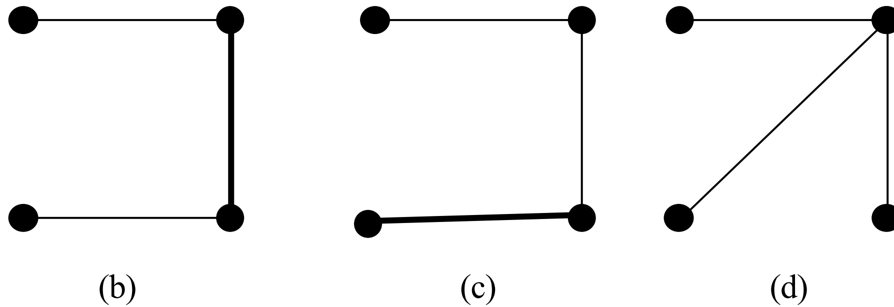
The next iterative-refinement algorithm, IR2, does not recompute the MST in every iteration; rather, a new spanning tree is computed by modifying the previously computed one. The modification performed does not regenerate previously generated trees and it guarantees the algorithm will terminate. Unlike IR1, this algorithm removes one edge at a time and prevents cycling by moving away from the center of the spanning tree whenever cycling becomes imminent. Figure 5.1 illustrates how this technique prevents cycling for the original graph of Figure 4.1. After computing the MST, the algorithm considers the middle edge (shown in bold) as the candidate for removal, as in Figure 5.1(b). But this edge does not have a replacement that can reduce the diameter, so the algorithm considers edges a little farther away from the center of the tree. The edge shown in bold in Figure 5.1(c) is the highest-weight such edge. As seen in Figure 5.1(d), the algorithm is able to replace it by another edge, and that reduces the diameter. This algorithm guarantees that the diameter does not increase in any iteration and in fact can reduce the diameter to a small constant (less than 1% of the number of nodes in the graph).

IR2 starts by computing the unconstrained MST for the input graph $G = (V, E)$. Then, in each iteration, it removes one edge that breaks a longest path in the spanning tree and

replaces it by a non-tree edge without increasing the diameter. The algorithm requires computing eccentricity values for all nodes in the spanning tree in every iteration.



(a)



(b) (c) (d)

Figure 5.1 An example of IR2

The initial MST can be computed 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 identifies one edge to remove from the 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 $3n$ iterations. Each iteration consists of two parts. In the first part, described in Section 5.1, we find an edge whose removal can contribute to reducing the diameter, and in the second part, described in Section 5.2, we find a good replacement edge. The IR2 algorithm is shown in Algorithm 2, and its two different edge-replacement subprocedures are shown in Algorithms 3 and 4. We use $ecc_T(u)$ to denote the eccentricity of node u with respect to spanning tree T; the maximum distance from u to any other node in T. The diameter of a spanning tree T is given by $MAX\{ecc_T(u)\}$ over all nodes u in T.

```
ALGORITHM 2 (IR2(G, T, k)).
begin
    if (T is undefined)
        then
            T := MST of G;
    compute ecc_T(z) for all z in V;                          /* G = (V, E) */
    C := ∅;
    move := false;
    repeat
        diameter := MAX_{z∈V} {ecc_T(z)} ;
        if (C = ∅)
            then
                if (move = true)
                    then   begin
                               move := false;
                               C := edges (u, z) that are one edge farther from the
                                        center of T than in the previous iteration;
                           end
                    else
                        C := edges (u, z) at the center of T;
        repeat
            (x, y) := highest weight edge in C;
            /* This splits T into two trees: subtree1 and subtree2 */
        until ((C = ∅) or ( MAX_{u∈subtree1} {ecc_T(u)} = MAX_{z∈subtree2} {ecc_T(z)} ));
        if (C = ∅)
            then                                  /* no good edge to remove was found */
                move := true;
            else   begin
                       remove (x, y) from T;
                       get a replacement edge and add it to T;
                       recompute ecc_T(z) for all z in V;
                   end
    until ((diameter ≤ k) or (edges to be removed are farthest from center of T));
    return T
end.
```

5.1 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 spanning tree T can be found by identifying the nodes u in T with $ecc_T(u) = \lceil diameter/2 \rceil$, the node (or two nodes) with minimum eccentricity.

Since we may have more than one edge candidate for removal, we keep a sorted list of candidate edges. This list, which we call C, is implemented as a max-heap 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. The end nodes of a longest path in T have maximum eccentricity, which is equal to the diameter of T. Therefore, we must verify that removing an edge splits the tree T into two subtrees, *subtree*1 and *subtree*2, such that each of the two subtrees contains a node v with $ecc_T(v)$ equal to the diameter of the tree T. If the highest-weight edge from heap C does not satisfy this condition, the algorithm removes it from C and considers the next highest. This process continues until the algorithm either finds an edge that breaks a longest path in T or the heap, C, becomes empty.

If the algorithm goes through the entire heap, C, without finding an edge to remove, it must consider edges farther from the center. This is done by identifying the nodes u with $ecc_T(u) = \lceil diameter/2 \rceil + bias$, where $bias$ is initialized to zero, and incremented by 1 every time the algorithm goes through C without finding an edge to remove. Then, the

algorithm recomputes C as all the edges incident to set of nodes u. Every time the algorithm succeeds in finding an edge to remove, *bias* is reset to zero.

This method of examining edges helps prevent cycling since we consider 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 T. This will ensure that the refinement process will terminate, since it will either reduce the diameter below the bound, k, or *bias* will become so large that the algorithm tries to remove the edges incident to the end-points of the longest paths in the tree.

In the worst case, computing heap C examines many edges in T, thereby requiring $O(n)$ comparisons. In addition, sorting C will take $O(n \log n)$ time. A replacement edge is found in $O(n^2)$ time since the algorithm must recompute eccentricity values for all nodes to find the replacement that helps reduce the diameter. Therefore, the iterative process, which removes and replaces edges for n iterations, will take $O(n^3)$ time in the worst case. Since heap 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 tree T fairly uniform so that it has a small number of edges near the center, as we will show in the next section. Since C is constructed from edges near the center of T, this will keep C small.

5.2 Selecting a Replacement Edge

When an edge is removed from a tree T, the tree T is split into two subtrees: *subtree*1 and *subtree*2. 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 have been so broken. We develop two methods, ERM1 and ERM2, to find such replacement edges.

5.2.1 Edge-Replacement Method ERM1

The first edge-replacement-method, shown in Algorithm 3, selects a minimum-weight edge (a, b) in G connecting a central node a in *subtree*1 to a central node b in *subtree*2. Among all edges that can connect *subtree*1 to *subtree*2, no other edge (c, z) will produce a tree such that the diameter of $(subtree1 \cup subtree2 \cup \{(c, z)\})$ is smaller than the diameter of $(subtree1 \cup subtree2 \cup \{(a, b)\})$. However, such an edge (a, b) is not guaranteed to exist in incomplete graphs.

ALGORITHM 3 (ERM1(*G*, *T*, *subtree*1, *subtree*2, *move*)).

```
begin
    recompute ecc_subtree1(.) and ecc_subtree2(.) for all nodes in each subtree;
    m1 := MIN_{u ∈ subtree1} {ecc_subtree1(u)};
    m2 := MIN_{u ∈ subtree2} {ecc_subtree2(u)};
    (a, b) := minimum-weight edge in G that has:
        (a ∈ subtree1) and (b ∈ subtree2) and (ecc_subtree1(a) = m1)
        and (ecc_subtree2(b) = m2);
    if (such an edge (a, b) is found)
        then
            add edge (a, b) to T;
        else  begin
                add the removed edge (x, y) back to T;
                move := true;
              end
    if ((C = ∅) or (bias = 0))
        then  begin
                move = true;
                C = ∅;
              end
    return edge (a, b)
end.
```

Since there can be at most two central nodes in each subtree, there are at most four edges to select from. The central nodes in the subtrees can be found by computing $ecc_{subtree1}(.)$ and $ecc_{subtree2}(.)$ in each subtree, then taking the nodes v with $ecc_{subtree}(v) = MIN\{ecc_{subtree}(u)\}$ over all nodes u in the subtree that contains v. This selection can be done in $O(n^2)$ time.

Finally, the boolean variable *move* is set to *true* every time an edge incident to the center of the tree is removed. This causes the removal of edges farther from the center of the tree in the next iteration of the algorithm, which prevents removing the recently added edge, (a, b).

This edge-replacement method seems fast at the first look, because it selects one out of four edges. However, in the early iterations of the algorithm, this method creates nodes of high degree near the center of the tree, which causes C to be very large. This, as we have shown in the previous section, causes the time complexity of the algorithm to increase by a constant factor. Furthermore, having at most four edges from which to select a replacement often causes the tree weight to increase significantly.

5.2.2 Edge-Replacement Method ERM2

The second edge-replacement-method, shown in Algorithm 4, computes $ecc_{subtree1}(.)$ and $ecc_{subtree2}(.)$ values for each subtree individually, as in ERM1. 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 (i) guarantees that the new edge does not increase the diameter, and (ii) guarantees reducing the length of a longest path in the tree at least by one. We enforce condition (i) by:

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

and condition (ii) by:

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

If no such edge (a, b) is found, we must remove an edge farther from the center of the tree, instead.

```
ALGORITHM 4 (ERM2(G, T, subtree1, subtree2, x, y, move)).
begin
    recompute ecc_subtree1(.) and ecc_subtree2(.) for all nodes in each subtree;
    m1 := ecc_subtree1(x);
    m2 := ecc_subtree2(y);
    (a, b) := minimum-weight edge in G  that has:
        (a ∈ subtree1) and (b ∈ subtree2) and (ecc_subtree1(a) ≤ m1)
        and (ecc_subtree2(b) ≤ m2) and ((ecc_subtree1(a) < m1 or (ecc_subtree2(b) < m2));
    if (such an edge (a, b) is found)
        then
            add edge (a, b) to T;
        else   begin
                    add the removed edge (x, y) back to T;
                    move := true;
               end
    return edge (a, b)
end.
```

Since ERM2 is not restricted to the centers of the two subtrees, it works better than ERM1 on incomplete graphs. In addition, it can produce DCMSTs with smaller weights because it selects a replacement from a large set of edges, instead of 4 or fewer edges as in ERM1. The larger number of edges increases the total time complexity of IR2 with ERM2 by a constant factor over IR2 with ERM1. However, this method does not create nodes of high degree near the center of the tree as in ERM1. This helps keep the size of heap C small in the early iterations, reducing the time complexity of IR2 by a constant factor.

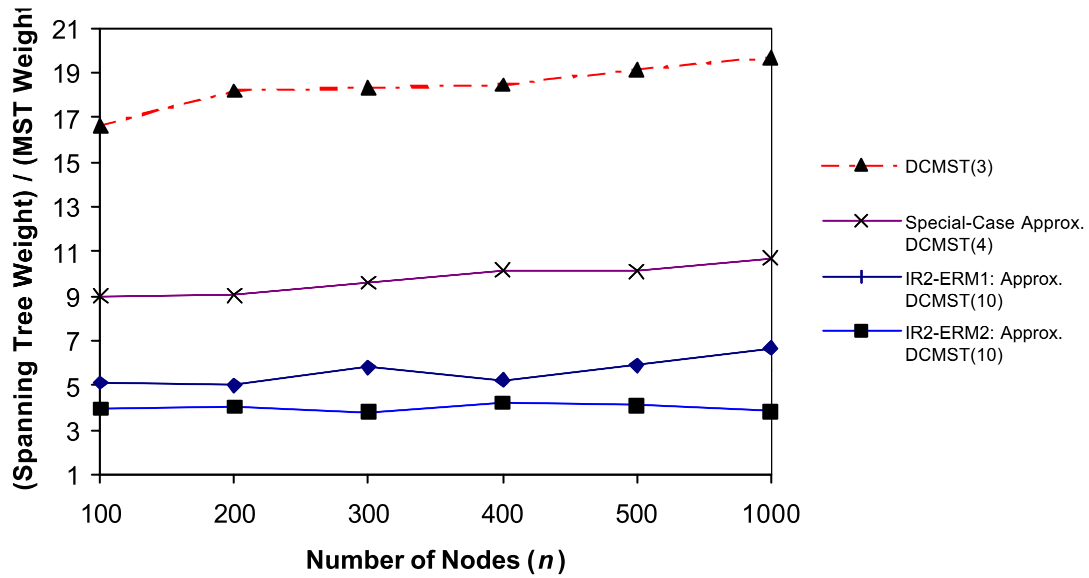


Figure 5.2 Weight quality of approximate solution, in randomly weighted complete-graphs with Hamiltonian-path MSTs, produced by IR2 using two different edge-replacement methods

5.3 Implementation

First, we parallelized Algorithm IR2 and implemented it on the MasPar MP-1, using complete random-graphs and complete graphs forced to have Hamiltonian-path MSTs, where edge weights were randomly selected integers between 1 and 1000. We also implemented IR2 sequentially on a PC with a Pentium III / 500 MHz processor using random-graphs and graphs forced to have Hamiltonian-path MSTs, where edge weights

were randomly selected integers between 1 and 10000, and the graph densities ranged from 20% to 100%. All input graphs had orders ranging from 50 to 2000, where 20 different graphs were generated for each order, density, and type of graphs. As expected, IR2 did not enter an infinite loop, and it always terminated within $3n$ iterations.

The weight quality of approximate DCMST(10) successfully obtained by this iterative-refinement algorithm using the two different edge replacement methods, ERM1 and ERM2, for graphs with Hamiltonian-path MSTs is shown in Figure 5.2. The diagram shows the weight of the computed approximate DCMST as a multiple of the weight of the unconstrained MST. It is clear that IR2 produced approximate solutions lower than the upper bounds, and IR2 using ERM2 produced lower weight solutions than IR2 using ERM1. As expected, the time required by IR2 using ERM1 to obtain approximate DCMSTs was greater than the time required by IR2 using ERM2. In addition, ERM1 required more memory space than ERM2, because the size of C when we use ERM1 is significantly larger than its size when ERM2 is used. This is caused by the creation of high-degree nodes by ERM1, as explained in Section 5.2. For the remainder of this dissertation, we will discuss the behavior of Algorithm IR2 only using ERM2 as the edge-replacement method.

When IR2 (using ERM2) was tested on random complete-graphs, the weight quality of approximate DCMST(10) produced by IR2 exceeded the weight of approximate DCMST(4) produced by the special-case algorithm when the edge weights were randomly selected integers between 1 and 1000, but not when the range of edge weights was 1 to 10000. In the latter case, IR2 also produced approximate DCMST(5) with

weight lower than the approximate DCMST(4) produced by the special-case algorithm. No spanning tree of diameter 3 was found in our samples of sparse graphs, and therefore, the special-case heuristic did not obtain any spanning trees of diameter 4 in those graphs. The average time required to produce approximate solutions with $n = 2000$ for DCMST(5) and DCMST(10), respectively, was 1924 and 1296 seconds in random complete-graphs, and 1231 and 538 seconds in random graphs with 20% density. The average weight of solutions with $n = 2000$ for DCMST(5) and DCMST(10), respectively, as a factor of the unconstrained MST weight, was 159 and 48 in random complete-graphs and 29 and 10.8 in random graphs with 20% density. In random graphs of all tested densities, the weight of solutions, as a factor of the unconstrained-MST weight, increased with n.

In graphs with Hamiltonian-path MSTs, the weight of approximate DCMST(10) produced by IR2 (using ERM2) was lower than the weight of approximate DCMST(4) produced by the special-case algorithm, regardless of the range of edge weights. The upper bounds (trees of diameter 3 and 4) were not available for sparse graphs of this type, either. The average time required by IR2 to produce approximate solutions, in graphs with Hamiltonian MSTs, with $n = 2000$ for DCMST(5) and DCMST(10), respectively, was 1488 and 1038 seconds in random complete-graphs, and 3038 and 1053 seconds in random graphs with 20% density. The average weight of solutions as a factor of the unconstrained MST weight, was approximately 26 and 9 for DCMST(5) and DCMST(10), respectively, in random complete-graphs, independent of n. In random graphs with 20% density, the weight of solution, as a factor of the unconstrained MST

weight, decreased with n. The weights of DCMST(5) and DCMST(10), respectively, as a factor of MST weight, was 44.6 and 18.9 for $n = 50$ and 21.5 and 11.1 for $n = 2000$.

The weight of solutions, as a factor of MST weight, in our samples of graphs with Hamiltonian-path MSTs did not increase with n because of the way these graphs were generated. To force a randomly generated graph to have a Hamiltonian-path MST, we randomly selected edges to include in the Hamiltonian path and randomly assigned them integer weights between 1 and 100. The rest of the edges were randomly generated integer-weights between 101 and 10000. Therefore, the average weight of an MST-edge is 50, and the average weight of a non-MST edge is 5050. However, there are only $(n - 1)$ edges in the MST and there are $O(n^2)$ non-tree edges in the rest of the graph. Thus, as n increases, the ratio:

(average weight of a non-MST edge) / (average weight of an MST edge)

decreases. This effect becomes clearer as the number of edges exceeds 10000. Consequently, we evaluate the solutions' weights in this type of graphs based on the upper and lower bounds (whenever available) calculated for the same set of graphs. However, the time taken by the algorithm can be compared with other types of graphs, where it can be seen that IR2 requires a longer time to obtain a solution when the diameter of the unconstrained MST is larger.

With all input graphs used for IR2, the weights of solutions and time required to obtain them increased whenever the diameter bound, k, was decreased. The quality of IR2 will be discussed further, in Chapter 8, when it is compared to the other algorithms we developed for the DCMST problem.

5.4 Convergence

As was shown in Sections 5.2 and 5.3, Algorithm IR2 is guaranteed to terminate, but it is not guaranteed to produce a solution. The failure rate of IR2 does not depend on what fraction of n the value of the bound on diameter, k, is. Rather, it depends on how small the constant, k, is. To see this, we must take a close look at the way we move away from the center of the tree while selecting edges for removal. Note that the algorithm will fail only when it tries to remove edges incident to the end-points of the longest paths in the spanning tree. Also note that the algorithm moves away from the center of the spanning tree every time it goes through the entire set C without finding a good replacement edge, and it returns to the center of the spanning tree every time it succeeds. Thus, the only way the algorithm fails is when it is unable to find a good replacement edge in $\lceil diameter/2 \rceil$ *consecutive* attempts, each of which includes going through a different set of C. Empirical results show that it is unlikely that the algorithm will fail for 8 consecutive times, which makes it suitable for finding an approximate DCMST when the value of k is a constant greater than or equal to 15.

When the input graphs were forced to have Hamiltonian-path MSTs, Algorithm IR2 was unable to find a spanning tree with diameter no more than 10 in some cases. In graphs with $100 \le n \le 2500$, our empirical results show a failure rate of 10% for $k = 10$ and 15% for $k = 5$. The success rate of IR2 (using ERM2) with (unrestricted) random complete-graphs was 90% for $n \ge 200$. In all graphs, the times required by IR2 to obtain a solution increased when the value of k was decreased.

When tested on incomplete graphs, Algorithm IR2 (using ERM2) was more than 65% successful in obtaining an approximate DCMST(5) for random graphs and graphs with Hamiltonian-path MSTs, where the density was at least 20% and $n \geq 500$. The success rate dropped slowly as the density of the input graph was decreased. For the same types of graphs and the same densities, the success rate also dropped when n was reduced below 500, where Algorithm IR2 becomes only 30% successful in finding an approximate DCMST(5) in graphs with $n = 50$ and density = 20%. This is understandable since the number of edges grows faster than the number of nodes. For example, when density is 20%, there are 24950 edges in a graph of 500 nodes, but only 245 edges in a graph of 50 nodes.

We measured and analyzed the time taken by IR2 to terminate. We measured the time taken by IR2 (using ERM2) to terminate successfully on the Pentium III / 500 MHz machine, and we obtained the following equations using a polynomial-fit program. When using complete random-graphs, IR2 required $(0.111n^3 + 62.7n^2 - 29583.7n + 2170981)$ and $(0.0736n^3 - 21.5n^2 + 10100n - 1230000)$ microseconds for $k = 5$ and $k = 10$, respectively. For random graphs with 20% density, IR2 required $(0.191n^3 + 77.2n^2 - 42250.8n + 3152147)$ and $(0.0639n^3 + 9.55\ n^2 - 5573.5n - 342626)$ microseconds for $k = 5$ and $k = 10$, respectively. This shows that the time required by IR2 for this type of graphs is almost unaffected by the change in graph density. When using complete graphs with Hamiltonian-path MSTs, IR2 required $(0.187n^3 + 50.2n^2 - 28288.1n + 2119730)$ and $(0.121n^3 + 22n^2 - 11709.6n + 74699121)$ microseconds for $k = 5$ and $k = 10$, respectively. For graphs with Hamiltonian-path MSTs and 20%

density, IR2 required ($0.248n^3 + 38.4n^2 - 26746.2n + 2120446$) and ($0.181n^3 - 133.8n^2 + 71780.63n - 7707461$) microseconds for $k = 5$ and $k = 10$, respectively. This shows that, in this type of graphs, the time required by IR2 increases slightly when the graph density is reduced.