An Improved Ant-Based Algorithm for the Degree-Constrained Minimum Spanning Tree Problem

Thang N. Bui, Xianghua Deng, and Catherine M. Zrncic

***Abstract*—The degree-constrained minimum spanning tree (DCMST) problem is the problem of finding the minimum cost spanning tree in an edge weighted complete graph such that each vertex in the spanning tree has degree $\leq d$ for some $d \geq 2$. The DCMST problem is known to be NP-hard. This paper presents an ant-based algorithm to find low cost degree-constrained spanning trees (DCST). The algorithm employs a set of ants which traverse the graph and identify a set of candidate edges, from which a DCST is constructed. Local optimization algorithms are then used to further improve the DCST. Extensive experiments using 612 problem instances show many improvements over existing algorithms.**

***Index Terms*—Ant-based algorithm, degree constrained spanning tree.**

I. Introduction

GIVEN A CONNECTED, edge weighted graph, the problem of finding a minimum cost spanning tree (MST) is well-known and is solvable in polynomial time [1]. There are applications in which additional constraints are imposed on the spanning tree, such as an upper bound on the degree of each node in the spanning tree. With a constraint on the degree of the nodes in the spanning tree the problem becomes NP-hard [2]. In this paper, we present an ant-based algorithm for the problem of finding the degree-constrained minimum spanning tree in a complete edge weighted graph.

The degree-constrained minimum spanning tree (DCMST) problem arises in various real world problems, including the design of networks for computers, telephone communication, transportation, and sewage systems [3], [4]. For example, a degree constrained spanning tree can be used to meet the requirements that nodes in a transportation network must be connected together and that there is a limit on the maximum number of roads that can meet at any one node. Construction costs or other costs are then modeled by having a cost function on the edges of the underlying graph. The objective is then to find the degree-constrained spanning tree of minimum cost. In some problems, limiting the degree of each node in a spanning tree also limits the potential impact if a node fails [5].

Not only is the DCMST problem NP-hard, approximating the optimal solution to within a constant factor is also NP-hard [6]. Thus, most works on this problem focus on heuristics that can find good solutions in a reasonable amount of time. Previous approaches to solving the DCMST problem include branch-and-bound [3], [4], neural networks [7], hill-climbing, simulated annealing, and genetic algorithms [5], [8]–[14]. In this paper, we present an ant-based algorithm for the DCMST problem. Ants are used to explore the input graph and identify a subset of the edge set that seems promising. The algorithm then takes this set of edges and constructs a degree-constrained spanning tree (DCST). Local optimization operations are used to further improve the constructed DCST. Note that our ant-based algorithm belongs to a class called ant-based optimization algorithms (ABO) [15] which was inspired by the well-known ant colony optimization (ACO) algorithms [16]–[18]. In ACO, a sequence of ants is used to solve the problem with later ants using information produced by the previous ants. In ABO, ants are only used to identify a promising area of the search space, which, in essence, reduces the search space of the problem. Then local search or construction methods are used to derive a solution.

We have tested our algorithm on over 150 well-known graphs, creating over 600 problem instances with degree constraints being equal to 2, 3, 4, and 5. The experimental results show that our algorithm found the best known values in 77% of the tested problem instances, found better results than the current best known in 9.8% of the instances, and did worse than the best known in 4.4% of the instances. The other 8.8% of the tested problem instances have no previous results. For a small subset of the problem instances that we tested, there exist data for the means of the published results. For these instances our algorithms gave the same mean value in 78% of the instances, better mean in 17% of the instances and worse mean in the remaining 5%.

The rest of this paper is organized as follows. In Section II, we give a formal definition of the problem and describe previous works. We describe our ant-based algorithm in Section III. Comparisons of the performance of our algorithm against that of some other existing algorithms are given in Section IV. In Section V, we consider the effects of some components of

Manuscript received July 25, 2010; revised December 15, 2010 and February 15, 2011; accepted February 15, 2011. Date of publication June 23, 2011; date of current version March 30, 2012.

T. N. Bui and C. M. Zrncic are with the Department of Mathematics and Computer Science, Pennsylvania State University at Harrisburg, Middletown, PA 17057 USA (e-mail: tbui@psu.edu; cmg212@psu.edu).

X. Deng was with the Pennsylvania State University at Harrisburg, Middletown, PA 17057 USA. He is now with Google, Inc., Mountain View, CA 94043 USA (e-mail: wdeng@google.com).

Digital Object Identifier 10.1109/TEVC.2011.2125971

our algorithm on its performance. Conclusions are given in Section VI.

II. Background

A *spanning tree* of a connected graph G is a subgraph of G that is a tree and has all the vertices of G. An undirected graph G is said to be *complete* if there is an edge between any pair of distinct vertices, and no self-loop is allowed. The *degree constrained minimum spanning tree* problem can be formally defined as follows. Given an undirected complete graph G with non-negative weights on the edges and an integer degree constraint $d \geq 2$, find a spanning tree of G with minimum total weight such that the degree of each vertex in the tree is at most d.[1]

As mentioned above, DCMST is NP-hard. In fact, for any fixed constant $d \geq 2$, the problem of finding a minimum cost degree-constrained spanning tree of degree at most d is still NP-hard [2]. It is also known that for any fixed rational $\alpha > 1$ and any fixed d, finding a spanning tree with degree constraint d and cost within a factor α of the optimal is NP-hard. However, there exists a polynomial time algorithm for the case where $d = O(b \log(n/b))$, $\alpha = O(\log(n/b))$, and b is the maximum degree of any vertex in the input graph [6].

Early heuristics for the DCMST problem used a branch and bound approach [3], [4], [19], where the branch and bound algorithm based on a Lagrangian relaxation [19] had the best results. Krishnamoorthy *et al.* [8] showed that the branch and bound heuristics method did not perform well on hard graphs. Other heuristics developed since then have included hill-climbing, simulated annealing, and neural networks [8], [9].

Genetic algorithms (GA) have been popular in solving the DCMST problem since the mid-1990s. Zhou and Gen [10] used Prüfer code to encode spanning trees for their GA. The encoding has been shown to be lacking locality when compared to other encodings [5]. Knowles and Corne [5] used a $|V| \times (d-1)$ array encoding in their GA. Their GA found the best solutions and compared favorably to earlier algorithms. Raidl and Julstrom [11], [12] used weighted and edge set encodings. Testing showed that the weighted encoding performed better than previous algorithms, and the edge-set encoding in turn produced even better results.

Ant algorithms also have been applied to the DCMST problem. An ant colony optimization approach [20] produced comparable results on several standard test instances. In 2006, Bui and Zrncic [21] gave an ant-based algorithm that generated better results when compared against then best known results for almost all standard test instances. Doan [22] extended that approach with a local optimization and provided some limited test results.

More recently, particle swarm optimization algorithms and their hybrids [23], [24] have produced the best results for many standard test instances.

The ant-based algorithm presented in this paper extends our previous work [21] with the addition of a local optimization phase. Additionally, this paper includes more test instances and produces significantly improved results over our previous work [21]. Compared to Doan's extension [22], this paper includes more sophisticated local optimization algorithms and provides a more extensive set of test instances as well as better results.

The algorithm in this paper was tested using the same data sets as many of the previous algorithms described above. Our solutions are compared to previously published results from each of the algorithms listed above, except for those from before the year 2000, the first version of the edge-set encoded GA [13], and the 2004 encoding by Soak, Corne, and Ahn [14]. The results of the algorithm in [14] were not used in our comparison because the specific graphs tested in that paper were not available for comparison. The algorithms before 2000 and that of [13] were excluded because more recent algorithms have produced better results. Those comparisons are detailed in Section IV.

III. Algorithm

A. Overview

The main idea of our ant-based algorithm is to use ants to effectively narrow the search space and then use simple and fast local optimization algorithms to find low cost degree-constrained spanning trees. More precisely, our algorithm consists of a number of iterations, each of which has three phases: exploration, construction, and optimization. Conceptually, the construction and optimization phases can be thought of as the exploitation stage in the typical exploration/exploitation paradigm.

In the exploration phase, ants explore the graph by traversing along the edges moving from one vertex to another and leaving pheromone on the edges. All ants follow the same set of rules designed to allow ants to identify edges that are potentially good candidates to be in a low cost DCST. In the construction phase, we first select a predetermined number of edges in the graph in order of decreasing pheromone level. A degree-constrained spanning tree is then constructed from this set of edges. If the selected set of edges is not sufficient to construct a degree-constrained spanning tree, then more edges of the graph, in order of decreasing pheromone level, are added to the set until a degree-constrained spanning tree can be constructed. More details are given in the following sections. Finally, in the optimization phase we employ two local optimization algorithms to improve the degree-constrained spanning tree constructed in the previous phase.

Each iteration in the algorithm produces a DCST and the best tree found is remembered and returned at the end of the algorithm. Fig. 1 shows the main ant-based algorithm (AB-DCST) for finding degree-constrained spanning trees.

B. Data Structures

To facilitate the description of the AB-DCST algorithm, we first briefly describe some main data structures used in the algorithm. AB-DCST has three major data structures: the graph, (partial) spanning trees, and ants. The graph consists

[1]It is a common practice to assume that G is complete so that a spanning tree with degree constraint d always exists. A connected but not complete graph can be converted to a complete one by adding missing edges with costs $\gg$ all other edge costs.

AB-DCST(G, w, d)
Input: $G = (V, E)$ a complete graph
w is the positive edge cost function
(generalized to return the total cost of the edges when a set of edges is given).
d is the degree constraint with $d \geq 2$.
Output: A spanning tree with each vertex having a degree $\leq d$ in the tree.
begin
// Initialization
$i \leftarrow 1$; $i_{best} \leftarrow 0$; $i_{restart} \leftarrow 0$
Create the set of ants A of size $|V|$
InitAntsEdges(A, V, E, w)
$T_{best} \leftarrow$ **ConstructSpanningTree**(E, d)
while $i < i_{max}$ *and* $i - i_{best} < i_{stop}$ **do**
AntsMove(A, E) // Exploration
$T \leftarrow$ **ConstructSpanningTree**(E, d)
// local optimization
$T \leftarrow$ **2-EdgeReplacement**(T, E, V, w)
$T \leftarrow$ **1-EdgeReplacement**(T, E, V, w)
if $w(T) < w(T_{best})$ **then**
$T_{best} \leftarrow T$
$i_{best} \leftarrow i$
foreach $e \in T_{best}$ **do** // enhance
$e.phm \leftarrow \gamma \times e.phm$
if $i - \max(i_{best}, i_{restart}) > R$ **then**
$i_{restart} \leftarrow i$
Restart(T_{best})
$i \leftarrow i + 1$
ResetAnts(A)
Update parameters γ and η
return T_{best}

Fig. 1. AB-DCST algorithm.

of a vertex set V and an edge set E where each edge in E is augmented with some pheromone information. More specifically, each edge contains three fields: pheromone level of the edge (*phm*), the number of times the edge is traversed by ants since the last pheromone update (*nVisited*), and the initial pheromone level of the edge (*initPhm*). A (partial) spanning tree is modeled as a set of edges; when a spanning tree is complete, it should have $|V| - 1$ edges. For the ants, the algorithm maintains a fixed set of cardinality $|V|$. Each ant maintains its current location, *location*, which is a vertex and a tabu list (*tabuList*) which is a fixed length queue storing only its most recently visited vertices.

C. Initialization Phase

In this phase, the algorithm first creates $|V|$ ants. Then it initializes the ants and edges as shown in Fig. 2. The algorithm puts an ant on each vertex of the graph and sets the tabu list of each ant to be empty. For the initialization of the pheromone level of edges, we use a scaling method similar to that of [25] that sets an edge with a lower cost to have a higher pheromone level and the initial pheromone level of each edge is in the range of $[(M - m)/3, 4(M - m)/3]$ where M and m are the

InitAntsEdges(A, V, E, w)
begin
for $i = 0$ **to** $|V| - 1$ **do**
$A[i].location \leftarrow V[i]$
$A[i].tabuList \leftarrow \emptyset$
foreach $e \in E$ **do**
$e.initPhm \leftarrow (M - w(e)) + (M - m)/3$
$e.phm \leftarrow e.initPhm$
$e.nVisited \leftarrow 0$

Fig. 2. Initialize ants' location and pheromone levels on the edges.

maximum and minimum edge costs in the graph, respectively. Specifically, the initial pheromone level of an edge e is defined as follows:

$$e.initPhm = (M - w(e)) + (M - m)/3$$

where $w(e)$ is the cost of edge e. Therefore, given any two edges e_1, e_2, we have $e_1.initPhm \leq 4 \times e_2.initPhm$. The goal of this scaling method is to prevent extremely large gaps in the initial pheromone levels and at the same time to allow desirable edges (those with low costs) to be differentiated from undesirable edges (those with high costs).

D. Exploration Phase

In the exploration phase, ants walk around the graph and lay pheromone along the trails. The goal of the ants is to discover low cost edges that are closely connected and mark them with a high level of pheromone. A number of these high pheromone edges are selected to form a set of candidate edges, from which a spanning tree is constructed. As edges with a high level of pheromone are expected to be closely connected, the candidate set will thus likely be connected making it more feasible to construct a spanning tree from these edges.

The movements of the ants are divided into a number of steps. In each step all the ants move in parallel, then the pheromone levels on the edges are updated. The process is then repeated until a specified number of steps have been made. For efficiency we found that it is better to update the pheromone on the edges only after some fixed number of steps instead of after every single step.

The movement of an ant is determined by the pheromone levels of the edges that are incident to the current location of the ant. Specifically, at each step, an ant chooses an edge among those edges incident to the current location of the ant and traverses along that edge. Each such edge is selected by the ant with a probability that is proportional to the pheromone level of that edge. When an ant traverses an edge, the edge pheromone level is scheduled to be increased by an amount equal to the initial pheromone level, *initPhm*, of that edge. As the initial pheromone level of an edge is inversely proportional to the cost of that edge, it follows that edges that have lower cost and edges that have been traversed more frequently by the ants are more likely to be selected by an ant. A number of mechanisms are used to mitigate the problem of getting stuck in local optima and to encourage more exploration. First, we keep a tabu list for each ant. The tabu list of an ant is a fixed

```
AntsMove(A, E)
begin
    for s = 1 to maxSteps do
        if s mod updatePeriod = 0 then
            UpdatePheromone(E)
        foreach a ∈ A do
            nAttempts ← 0
            moved ← False
            while not moved and nAttempts < 5 do
                v1 ← a.location
                Select an edge (v1, v2) at random and
                proportional to (v1, v2).phm
                if v2 ∉ a.tabuList then
                    add v2 into a.tabuList
                    a.location ← v2
                    (v1, v2).nVisited++
                    moved ← True
                else nAttempts++

UpdatePheromone(E)
begin
    foreach e ∈ E do
        e.phm ← (1 − η) × e.phm +
                e.nVisited × e.initPhm
        e.nVisited ← 0
        if e.phm > maxPhm then
            e.phm ← maxPhm − e.initPhm
        if e.phm < minPhm then
            e.phm ← minPhm + e.initPhm
```

Fig. 3. Ant exploration algorithm.

length queue that stores only the most recently visited vertices by the ant. Ants are prohibited from selecting edges that lead to vertices on their tabu list. Second, the pheromone level of each edge is periodically evaporated, i.e., reduced by a time varying factor. More precisely, the evaporation rate is higher in the beginning of the algorithm, and is gradually decreased as the algorithm progresses. This allows ants to be more exploratory at first and less so near the end. Finally, if pheromone levels are not bounded, ants will tend to converge quickly on a small number of edges causing the algorithm to be stuck in local optima. Thus, we limit the pheromone levels to be in the range of $[minPhm, maxPhm]$ defined as follows:

$$\begin{aligned} maxPhm &= 1000 \times (M - m) + (M - m)/3 \quad \text{and} \\ minPhm &= (M - m)/3 \end{aligned}$$

where m and M are the minimum and maximum edge costs, respectively, of the edges.

As mentioned at the beginning of this section, the pheromone level of an edge is updated only periodically. Each edge in the graph keeps a count of the number of times it has been visited by ants since the last pheromone update. When it is time to update the pheromone, the pheromone level of an edge e in the graph is updated as discussed above, that is

$$e.phm = (1 - \eta)e.phm + e.nVisited \times e.initPhm$$

where η is the evaporation factor and $e.initPhm$ is the initial pheromone level of edge e given in Section III-C. The evaporation factor η is assigned an initial value of 0.5. It is then gradually decreased as the algorithm progresses. Note that this is similar to the strategy that is used in max-min ant system [26]. As discussed above we want to make sure that the pheromone level of an edge always lies within certain range. Furthermore, if the new pheromone level of an edge e is outside of the range, we do not reset the pheromone level to the exact boundary. Rather, we adjust it by the amount of $\pm e.initPhm$ to help the ants to distinguish desirable edges from undesirable ones, even if the pheromone levels for many edges are approaching the boundaries, $maxPhm$ or $minPhm$. Fig. 3 shows the details of how an ant moves and how pheromone levels are updated.

```
ConstructSpanningTree(E, d)
begin
    T_n ← ∅
    Sort E in order of decreasing pheromone level
    C ← top nCandidates edges from E
    Sort C in order of increasing edge cost
    while |T_n| < |V| − 1 do
        if C ≠ ∅ then
            Let e be the next edge in C
            Remove e from C
            if adding e to T_n would not create a cycle or
               violate the degree constraint then
                T_n ← T_n ∪ {e}
        else
            C ← next nCandidates edges from E
            Sort C in increasing order of edge cost
    return T_n
```

Fig. 4. Tree construction algorithm.

E. Construction Phase

It is expected that the ants in the exploration phase have discovered edges that are closely connected and have low costs. These edges are marked by the ants with a high level of pheromone. The construction phase of the algorithm utilizes this information to construct a degree-constrained spanning tree. First, the edge set is sorted in order of decreasing pheromone level and the top *nCandidates* edges in this sorted list are selected to form a candidate set. This set of edges is then sorted in order of increasing edge cost. A spanning tree is constructed from this sorted list of edges using Kruskal's algorithm [1] with a simple modification to avoid violating the degree constraint. If the set of edges is exhausted before the tree is completed, then the next *nCandidates* edges with highest pheromone level are selected and the process is repeated until we have a spanning tree. The algorithm for the construction phase is shown in Fig. 4.

2-EdgeReplacement(T, E, V, w)
Input: T is a valid spanning tree.
Output: A valid spanning tree
begin
  $T_n \leftarrow T$
  $nTries \leftarrow 0$
  while $nTries < |V|/2$ **do**
    $e_1 \leftarrow$ a random edge in T_n
    $e_b \leftarrow e_1$; $c_b \leftarrow 0$
    foreach $e_2 \in T$ *and* e_1, e_2 *are not adjacent* **do**
      $\{e_{r_1}, e_{r_2}\} \leftarrow rep2Opt(e_1, e_2)$
      if $w(e_1) + w(e_2) - w(e_{r_1}) - w(e_{r_2}) > c_b$ **then**
        $c_b \leftarrow w(e_1) + w(e_2) - w(e_{r_1}) - w(e_{r_2})$
        $e_b \leftarrow e_2$
    if $c_b > 0$ **then**
      $T_n \leftarrow (T_n - \{e_1, e_b\}) \cup rep2Opt(e_1, e_b)$
      $nTries \leftarrow 0$
    else
      $nTries \leftarrow nTries + 1$
  return T_n

Fig. 5. 2-EdgeReplacement local optimization algorithm.

1-EdgeReplacement(T, E, V, w)
Input: T is a valid spanning tree.
Output: A valid spanning tree
begin
  $T_n \leftarrow T$
  Sort E in order of increasing edge cost
  repeat
    changed $\leftarrow$ False
    Sort T_n in order of increasing edge cost
    $j \leftarrow |V| - 1$
    while $j \geq 0$ **do**
      $k \leftarrow 0$
      while $w(T_n[j]) > w(E[k])$ **do**
        if *swapping* $E[k]$ *and* $T_n[j]$ *creates no cycle in* T_n *and satisfies degree constraint* **then**
          $T_n \leftarrow T_n \cup \{E[k]\} - \{T_n[j]\}$
          changed $\leftarrow$ True
          break
        $k \leftarrow k + 1$
      $j \leftarrow j - 1$
  until *not changed*
  return T_n

Fig. 6. 1-EdgeReplacement local optimization algorithm.

F. Optimization Phase

After a spanning tree is constructed, the algorithm tries to improve the tree using two local optimization algorithms: 2-EdgeReplacement (shown in Fig. 5) and 1-EdgeReplacement (shown in Fig. 6). As its name suggests, the 2-EdgeReplacement algorithm selects two edges from the spanning tree and replaces them with two new edges so that the degree constraint is maintained and the cost of the tree is smaller. The 1-EdgeReplacement algorithm behaves similarly except that only one edge is replaced. In what follows, we describe these algorithms in more detail.

The 2-EdgeReplacement algorithm goes through a number of iterations until there are $|V|/2$ consecutive iterations in which no improvement is made. In each iteration it selects two edges in the tree and replaces them with two new edges, if possible, so that the degree constraint is not violated and the resulting tree has lower cost. The process for selecting the two edges to be replaced is as follows. First, randomly pick an edge, say e_1. Among all edges that are not adjacent to e_1, select the edge, say e_2, that would give the largest improvement to the tree cost if e_1 and e_2 are replaced by two new edges. The process of replacing two tree edges with two new edges is described in details next.

Let (i, j) and (k, l) be the two tree edges to be replaced. We want to efficiently find two new edges to replace (i, j) and (k, l) so that we still have a spanning tree. When (i, j) and (k, l) are removed from the tree, we have three disjoint components each of which is a tree. Furthermore, since (i, j) and (k, l) are not adjacent, the four vertices i, j, k, and l are distinct. Fig. 7 shows the three disjoint components that would result if (i, j) and (k, l) are removed from the tree. There are many ways to reconnect these three components to form a tree. However, we restrict ourselves to reconnect these components only through

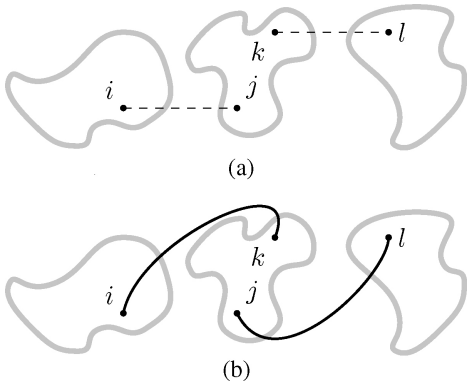


Fig. 7. Operation *rep2Opt*. (a) Removed edges: (i, j) and (k, l). (b) Replacement edges: (i, k) and (j, l).

the four vertices i, j, k, and l. With the vertices labeled as in Fig. 7 it is easy to check that there are three different sets of replacement edges that can be used to reconnect the three components to create a tree: $\{(i, k), (i, l)\}$, $\{(i, l), (j, l)\}$, and $\{(i, k), (j, l)\}$. With the first two sets of replacement edges, we need to check that the degree constraints at i and at l are still satisfied. The third set of replacement edges will satisfy the degree constraint automatically. We have done limited experiments with two methods of selecting the set of replacement edges. The first method selects the best set among the three available replacement sets, i.e., the one that gives the most reduction in the tree cost. The second method simply uses the third replacement set, i.e., $\{(i, k), (j, l)\}$. Our experiment showed that the two methods are comparable, and we used the second method in our algorithm.

The main problem is how to identify the replacement edges (i, k) and (j, l). Let $dist(v_1, v_2)$ denote the length of the path from v_1 to v_2 in the spanning tree. We observe that $dist(k, j)$ is

TABLE I
AB-DCST ALGORITHM PARAMETERS

Parameter	Value	Comments
$i_{\max}$	10 000	Maximum allowed iterations
i_{stop}	2500	Maximum iterations without improvement
maxSteps	75	Number of steps that an ant traverses each iteration
nCandidates	$5\|V\|$	Candidate set cardinality
η	0.5	Initial pheromone evaporation factor
γ	1.5	Initial pheromone enhancement factor
$\Delta\eta$	0.95	Update constant applied to η
$\Delta\gamma$	1.05	Update constant applied to γ
updatePeriod	maxSteps/3	Pheromone update period
R	100	Number of iterations without improvement before escaping

the smallest among $dist(i, k)$, $dist(i, l)$, $dist(j, k)$, and $dist(j, l)$. Hence, the problem is reduced to identifying the end vertex of edge (k, l) that is closer to the end vertices of the edge (i, j). We propose an efficient technique that has an amortized cost of $O(1)$ for each edge (k, l): first do a breadth-first search (BFS) starting from i with (i, j) removed and gets a BFS tree T_i; then for each edge (k, l) in T_i, the vertex with smaller BFS number is closer to (i, j). Symmetrically, a BFS can be done starting from j with (i, j) removed. Since for any edge (i, j), there are at least $|V| - 1 - 1 - 2(d - 2) = |V| - 2d + 2$ non-adjacent edges and the BFS takes $O(|V|)$ operations, the amortized cost for each non-adjacent edge (k, l) is $O(1)$ assuming that $d \ll |V|$. Therefore, the processing of one edge (i, j) is in $O(|V|)$. The operation of finding the replacement set of edges for a pair of edges $\{e_1, e_2\}$ is referred to as $rep2Opt(e_1, e_2)$ in the algorithm of Fig. 5.

The main idea in the 1-EdgeReplacement algorithm is to pick a tree edge e and replace it with an edge e' such that: 1) e' is not in the tree; 2) e' has a lower cost than that of e; and 3) the result is a spanning tree that does not violate the degree constraint. The 1-EdgeReplacement algorithm, shown in Fig. 6, keeps applying the one edge replacement operation until no such replacement is possible. For each tree edge, the algorithm examines all non-tree edges that have lower cost. In the worst case, the processing of one tree edge is $O(|V|^2)$ since it may need to consider all the replacement candidate edges and the number of which is $|V|(|V| - 1)/2 - |V| + 1$.

G. Post Processing

Five operations are carried out after the local optimization phase. First, the resulting tree is compared against the best tree in terms of cost. If it has a lower cost, then the best tree is set to be the current tree. Second, the pheromone levels of the edges in the best tree are enhanced by multiplying an enhancement factor ($\gamma \geq 1$) to emphasize those edges. Third, when the algorithm is stuck in a local optimum, that is, after a certain number of iterations without improving the best tree, the algorithm will try to jump out of the local optimum by running a restart algorithm, shown in Fig. 8. The restart algorithm reduces the pheromone level of each edge in the current best tree to some random value between 10% and 30% of its current pheromone level. Fourth, the ants are reset to prepare for the next iteration: half of all ants stay at their current locations; the other half are randomly placed in the vertices as shown in Fig. 8. Fifth, the enhancement factor γ and the evaporation factor η are updated. More precisely, γ has an initial value of 1.5 and is multiplied by 1.05 every 500 iterations. Similarly, the pheromone evaporation factor η has an initial value of 0.5 and is multiplied by 0.95 every 500 iterations. The setup of those factors reflects the two stages of the algorithm: exploration and exploitation. During the exploration phase (near the beginning of the algorithm) we do not put too much emphasis on the found trees and the evaporation rate is higher to encourage ants to explore a larger region of the graph. As the algorithm gradually moves to the exploitation phase, we want the ants to zoom in on those desirable edges. Therefore, the edges of the best tree should be emphasized more by increasing the value of the enhancement factor γ and at the same time decreasing the evaporation rate η.

```
Restart(T_b)
begin
    foreach e ∈ T_best do
        e.phm ← e.phm × Random(0.1, 0.3)

ResetAnts(A)
begin
    foreach a ∈ A do
        if Random(0, 1) < 0.5 then
            a.location ← Random(0, |V| − 1)
        a.tabuList ← ∅
```

Fig. 8. Restart and reset ants.

H. Parameters

The parameters in the AB-DCST algorithm are listed in Table I. Note that these parameters are not targeted for any specific type of graphs and have been used with many different classes of graphs without any modification as described in Section IV. The values of the parameters were decided based on comparative testing using a set of three randomly chosen graphs.

TABLE II
DATA SETS

Graph Class	# of Graphs Used	$\|V\|$ Range
CRD	40	30 to 100
SYM	30	30 to 70
STR	41	30 to 200
SHRD	10	15 to 150
RANDOM	11	15 to 1000
R	9	50 to 200
M	12	50 to 500
Total	153	15 to 1000

IV. EXPERIMENT

A. Data Sets

To test our AB-DCST algorithm we used a total of 153 graphs that belong to seven different classes. Table II shows a summary of these classes. These graphs are used extensively in the literature and can be divided into two categories: Euclidean graphs (including CRD [3], SYM [19], STR [8]) and non-Euclidean graphs (including SHRD [8], RANDOM [27], R [9], M [5]). In Euclidean graphs, vertices represent points in some space and edge costs are the Euclidean distances between end points. The CRD graphs represent points and edges in 2-D spaces. SYM and STR graphs are based on higher dimensional spaces. The RANDOM graphs are generated by assigning random weights to each edge. The structured hard (SHRD), random hard (R), and misleading hard (M) graphs are deliberately created to make finding a DCMST difficult: SHRD graphs are constructed to make the number of optima limited; R and M graphs are created with the unconstrained MST to contain star patterns with high degrees. Furthermore, M graphs are designed to mislead greedy algorithms.

For each of the 153 graphs, we create four problem instances one for each of the degree constraints, $d = 2, 3, 4$, and 5, giving a total of 612 problem instances. These degree constraints are the most often used in the literature. We note that no single existing algorithm appears to have published data for all these 612 problem instances, or across all classes of graphs as we have here.

B. Setup

Our AB-DCST algorithm is implemented in C++ and compiled using g++ with the $-$O3 flag. The experiment was conducted on several machines running Solaris 10. The machines have the same configuration consisting of Intel Core2 Duo E8600 at 3.33 GHz and either 4 GB or 6 GB of RAM. Although they have different memory, the source code is compiled into 32 bit binary and thus there should be no significant influence by the memory difference. For each problem instance the algorithm is run 50 times. We used the Mersenne Twister random number generator [28] in our implementation.

C. Results

Since most available results for the problem instances tested in this paper are in the form of best results produced by published algorithms, we first describe the best results produced by AB-DCST as compared to the best known results. We then provide some comparisons of the means for those problem instances where such results exist. The entire set of AB-DCST results including the best, the mean, and the standard deviation, of all problem instances are given in a supplementary file in the IEEExplore database.

Overall, for the 612 problem instances that we tested, AB-DCST found the best known values in 77% of the instances, found better results in 9.8% of the instances, did worse than the best known results in 4.4% of the instances. There was no known result for the remaining 8.8% of the instances for us to compare against. When AB-DCST gives results that are better than the best known values, the improvements range from 0.02% to 65.42% with a median of 0.33%. When AB-DCST performs worse, the differences from the best known values range from 0.01% to 4.03% with a median of 1.60%.

Among the instances with best known results there are 414 instances with known optimal solution. Of these 414 problem instances, AB-DCST found the optimal solution in 405 cases. For the remaining nine cases, AB-DCST missed the optimal solution by less than 1% in four cases and between 1% and 4.03% in five cases. Fig. 9 shows a summary of the performance of the AB-DCST algorithm for each class of graphs and each degree constraint. More detailed information are provided for each class below.

For each problem instance, we report the following measures:

$$\text{Difference} = \frac{\text{best known} \ - \text{our best of 50 runs}}{\text{best known}}$$

and

$$\text{Coefficient of Variation} = \frac{\sigma}{\mu} \times 100$$

where σ and μ are the mean and standard deviation of the results from 50 runs of the problem instance, respectively.

There are 40 graphs in the CRD class, which yield 160 problem instances. AB-DCST performs well on this class of graph. Specifically, AB-DCST provides the same results as previously best known values in 145 instances and it provides better results in 13 instances, with improvement ranging from 0.11% to 2.21%. In the remaining two instances, AB-DCST gives worse results but is still within 0.1% of the best known. We note that all the better results provided by AB-DCST were for those instances with $d = 2$.

For the M class, we have 12 graphs (or 48 problem instances). Of these 48 instances, there are known optimal solution for nine instances, all with $d = 5$. AB-DCST found all these optimal solutions. Of the remaining three instances with $d = 5$, AB-DCST did not find the best known results for two of these instances, missing by 0.05% and 0.24%, and found a better solution for the other instance (better by 0.02%). The remaining 36 instances of the M class, for $d = 2, 3$, and 4 have no previous results.

The R class has nine graphs, that gives 36 instances. There are previous results for only 18 instances with $d = 4, 5$. Among the instances with previous results, AB-DCST matched the best known for two of these instances and gave better results on the remaining 16 instances, with improvement ranging from 0.13% to 8.62%.

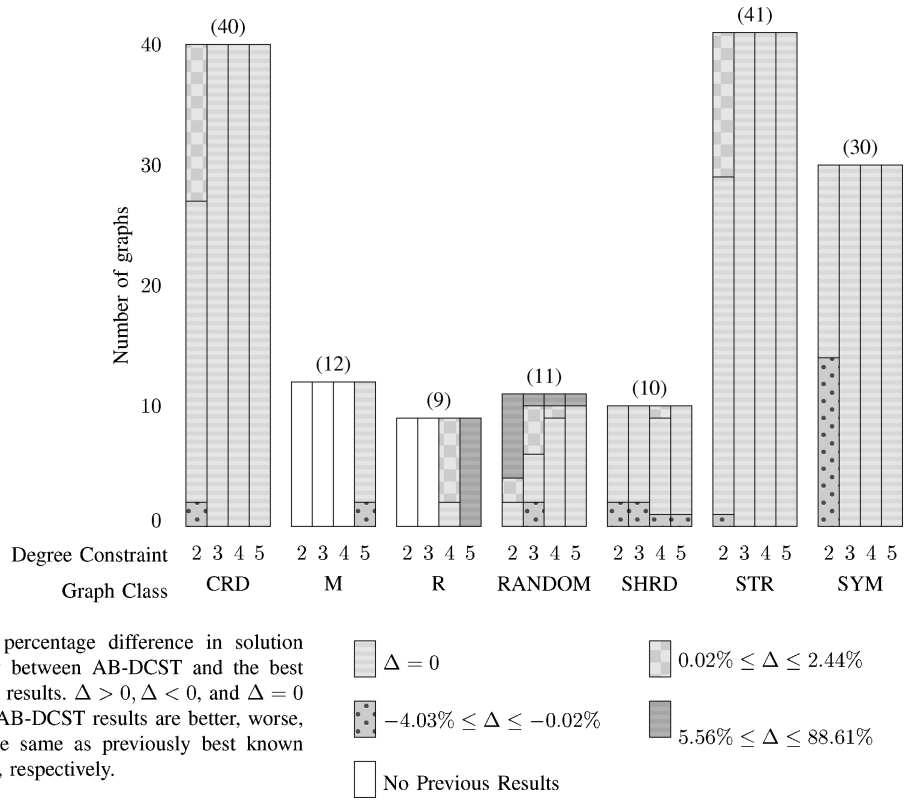


Fig. 9. Summary of the performance of AB-DCST.

For the RANDOM class, we have 11 graphs, and thus 44 instances. AB-DCST did not match the best known solution in two instances, missing by 2.38% and 2.78%. It matches the best known results in 25 instances, and gave better solution in the remaining 17 instances with improvement ranging from 0.12% to 88.61%.

There are 10 graphs in the SHRD class giving 40 instances. There are 33 instances with known optimal solution, AB-DCST matched all of these known optimal solutions. Of the remaining seven instances with no known optimal solution, AB-DCST gave better results than the best known solutions in one case, with an improvement of 0.02%. For the remaining six cases, AB-DCST did worse than the best known solutions, with differences ranging from 0.01% to 0.02%.

The STR class has 41 graphs, i.e., 164 problem instances. Of these, there are 138 instances with known optimal and AB-DCST found all these optimal solutions. Of the remaining 26 instances with best known solutions, AB-DCST found the best known solutions in 13 cases, did better in 12 cases (with improvement ranging from 0.02% to 0.06%), and did worse in one case, for which it missed by 0.04%.

Finally, we have 30 graphs and hence 120 problem instances in the SYM class. AB-DCST found the best known solutions in 106 of the cases and missed the best known solution in the remaining 14 cases, with differences ranging from 0.27% to 4.03%.

Due to space consideration,[2] we will provide only data for instances where AB-DCST performs worse or better than the best known results. Tables IV–VII show these data grouped by the degree constraint. In each table, we give the graph name, the number of vertices in the graph, the best known result, the best result found by AB-DCST in 50 runs, the percentage difference between the previous two columns, the coefficient of variation, i.e., the standard deviation divided by the mean (computed from the 50 runs per instance), and the average running time of the 50 runs of the algorithm in seconds.

TABLE III
SUMMARY OF AB-DCST PERFORMANCE

Graph Class	#Instances (Compared to Best Known)				
	Total	Equal	Better	Worse	No Previous
CRD	160	145	13	2	0
M	48	9	1	2	36
R	36	2	16	0	18
RANDOM	44	25	17	2	0
SHRD	40	33	1	6	0
STR	164	151	12	1	0
SYM	120	106	0	14	0
All	612	471	60	27	54

If optimal values are known and are not achieved by existing algorithms, a more accurate measure of the performance of our algorithm can be obtained as follows:

$$\frac{\text{our result} - \text{optimal result}}{\text{best known} - \text{optimal result}}.$$

Intuitively, this ratio shows the improvement made by the algorithm in closing the gap between the best known result and the optimal value. Due to the lack of optimal results for

[2]The full data set is provided in the supplementary file in the IEEExplore database.

most graphs, we use lower bounds provided by Ernst [24] to approximate the optimal results, that is

$$\text{effective improvement} = \frac{\text{our result} - \text{lower bound}}{\text{best known} - \text{lower bound}}.$$

The effective improvements are shown in Table VIII for instances with known lower bounds. From the table, we can see that the effective improvements are much more significant than the percentage differences. Effective improvements are not computed for data that have negative percentage differences, as those are the cases where our results are worse than the current best known results.

We note that in evaluating the performance of AB-DCST above, we have used data from many different sources as no single previous algorithm was run on all the instances that we have here.

As mentioned before, AB-DCST was run 50 times for each problem instance. The mean (μ), standard deviation (σ), and the coefficient of variation (CV = $(\sigma/\mu) \times 100$), of the results for each instance are computed. We observe that the coefficient of variations are quite small, ranging from 0% to 4.09%, an indication that the performance of the algorithms is quite stable.

For a small set of problem instances we have data for the mean of the results [29], [24]. We summarize the results in Tables IX and X. The algorithm given in [24] represents one of the current best algorithms for the DCMST problem. As can be seen in Table IX, the means of the results produced by this algorithm and that of AB-DCST are mostly the same. AB-DCST provided better mean for 7 problem instances and worse mean for 2 problem instances out of 42 problem instances.

In [29], the following type of data is provided for a subset of the graphs in the M class (with degree constraint equal to 5). The values provided in [29] are the relative gap between the mean (over 20 runs) result and the optimal value, that is

$$\text{Gap} = \frac{C_{\text{mean}} - C_{\text{opt}}}{C_{\text{opt}}} \times 100$$

where C_{mean} and C_{opt} are the mean and the optimal value, respectively. Thus, a smaller gap value means a better solution quality. We compute the same gap values for the results produced by AB-DCST over the same set of problem instances. The results are shown in Table X. As can be seen, AB-DCST gives the same performance for 3 problem instances and better results for 6 problem instances out of 9 problem instances. As mentioned before, graphs in the M class are designed to mislead greedy type algorithms and they are generally harder instances than graphs from other classes.

V. Discussion

In this section, we consider the contribution of various components of the AB-DCST algorithm to its performance. Specifically, we consider whether or not the exploration phase by the ants provides a positive impact to the algorithm's performance. We also consider whether we really need both local optimization algorithms or if one is sufficient to provide the same performance.

TABLE IV
Instances Where AB-DCST Performs Better Or Worse Than the Best Knowns

$d = 2$ Graph	$\lvert V\rvert$	Best Known (* – *optimal value*)	AB-DCST	Difference $\left(\frac{\text{best known} - \text{our best}}{\text{best known}}\right)$	Coeff. of Variation	Average Time (sec)
sym700	70	2012*	2093	–4.03%	3.15	10
sym702	70	1864*	1937	–3.92%	1.65	9.98
sym704	70	2017	2089	–3.57%	1.91	9.70
sym709	70	1749	1798	–2.80%	3.01	10
sym703	70	1497*	1536	–2.61%	1.77	10
sym706	70	1489	1522	–2.22%	2.78	11
sym701	70	1931	1966	–1.81%	3.26	10
sym708	70	1816	1848	–1.76%	2.24	9.88
sym502	50	2116*	2150	–1.61%	1.54	5.04
sym707	70	2371	2409	–1.60%	2.64	10
sym503	50	1965*	1991	–1.32%	1.76	5.02
sym705	70	2156	2175	–0.88%	1.88	10
sym506	50	1917*	1932	–0.78%	1.69	5.52
sym508	50	1861	1866	–0.27%	1.79	5.44
crd107	100	7720	7728	–0.10%	0.34	18
crd101	100	7707	7714	–0.09%	0.51	23
str1009	100	12 403	12 408	–0.04%	0.06	21
shrd1000	100	48 141	48 145	–0.01%	0.01	24
shrd1500	150	109 681	109 691	–0.01%	0.00	52
str1001	100	5001	5000	0.02%	0.11	22
str1007	100	10 531	10 529	0.02%	0.05	22
str1005	100	8702	8699	0.03%	0.07	25
str1006	100	10 532	10 529	0.03%	0.05	23
str2008	200	15 876	15 872	0.03%	0.08	95
str1002	100	7092	7089	0.04%	0.08	23
str1003	100	7092	7089	0.04%	0.08	22
str1008	100	12 433	12 428	0.04%	0.07	25
str1508	150	13 892	13 887	0.04%	0.13	65
str1500	150	14 016	14 009	0.05%	0.12	62
str502	50	6129	6126	0.05%	0.01	4.54
str1004	100	8704	8699	0.06%	0.06	24
crd707	70	6474	6467	0.11%	0.38	9.88
crd706	70	6679	6671	0.12%	0.43	10
crd503	50	5088	5079	0.18%	0.00	3.34
crd104	100	7557	7541	0.21%	0.43	19
crd100	100	7063	7044	0.27%	0.54	20
crd108	100	7165	7137	0.39%	0.31	19
crd702	70	6790	6763	0.40%	0.28	11
crd102	100	7582	7549	0.44%	0.49	22
crd502	50	5507	5480	0.49%	0.05	4.30
crd103	100	7701	7660	0.53%	0.55	22
crd704	70	6405	6370	0.55%	0.41	7.68
crd109	100	7333	7286	0.64%	0.50	19
random25	25	57	56	1.75%	0.88	0.82
random20	20	49	48	2.04%	0.00	0.40
crd105	100	7235	7075	2.21%	0.43	21
random50	50	126	119	5.56%	1.64	4.38
random100	100	271	246	9.23%	4.09	30
random400	400	1489	1277	14.24%	2.16	375
random200	200	681	574	15.71%	2.90	111
random300	300	1095	840	23.29%	4.09	362
random500	500	2356	1706	27.59%	3.01	810
random1000	1000	11 310	3911	65.42%	2.49	6440

It is natural to ask whether the algorithm would perform as well without the ants. We implemented two new algorithms to study the effect of the ant exploration phase. The first algorithm, called No-Ants, is obtained by taking the AB-DCST algorithm and removing the ant exploration phase. The second algorithm, called Perturbation, is obtained by taking

TABLE V
INSTANCES WHERE AB-DCST PERFORMS BETTER OR WORSE THAN THE BEST KNOWNS WHERE $d = 3$

$d = 3$ Graph	$\|V\|$	Best Known	AB-DCST	Difference $(\frac{\text{best known} - \text{our best}}{\text{best known}})$	Coeff. of Variation	Average Time (sec)
random20	20	36	37	–2.78%	0.00	0.32
random25	25	42	43	–2.38%	0.00	0.54
shrd1500	150	72 678	72 682	–0.01%	0.00	84
random200	200	379	378	0.26%	0.18	111
random400	400	732	723	1.23%	0.23	355
random300	300	530	518	2.26%	0.31	276
random500	500	901	879	2.44%	0.33	563
random1000	1000	16 293	1855	88.61%	0.22	3579

TABLE VI
INSTANCES WHERE AB-DCST PERFORMS BETTER OR WORSE THAN THE BEST KNOWNS WHERE $d = 4$

$d = 4$ Graph	$\|V\|$	Best Known	AB-DCST	Difference $(\frac{\text{best known} - \text{our best}}{\text{best known}})$	Coeff. of Variation	Average Time (sec)
shrd1000	100	23 649	23 652	–0.01%	0.01	28
shrd1500	150	54 205	54 196	0.02%	0.00	74
random500	500	848	847	0.12%	0.03	316
R200n3	200	15.65	15.63	0.13%	0.00	103
R50n2	50	4.27	4.26	0.23%	0.00	4.36
R100n2	100	8.08	8.06	0.25%	0.00	21
R200n1	200	16.13	16.09	0.25%	0.00	81
R100n1	100	8.09	8.06	0.37%	0.00	19
R100n3	100	8.07	8.02	0.62%	0.00	19
R200n2	200	15.79	15.69	0.63%	0.00	113
random1000	1000	13 529	1774	86.89%	0.09	2808

AB-DCST and replacing the ant exploration phase with a random perturbation on the costs of 10% of the edges. The (modified) Kruskal algorithm and the two local optimization algorithms are the same in both algorithms as detailed previously in the description of AB-DCST. The objective of the random perturbation of the edge costs is to help the algorithm getting out of local optima. We ran AB-DCST, No-Ants, and Perturbation algorithms on the same complete set of data and the same experimental setup as described in Section IV. For each problem instance, we recorded the mean and the best results obtained by the algorithms. As the data does not seem to be normally distributed we performed a non-parametric one-sided paired *t*-test (i.e., Wilcoxon test) [30], which is known to be robust for non-normal data, on the means, and separately on the best values, of the results of the three algorithms. We compared AB-DCST against No-Ants and AB-DCST against Perturbation. Table XI shows the *p*-

TABLE VII
INSTANCES WHERE AB-DCST PERFORMS BETTER OR WORSE THAN THE BEST KNOWNS WHERE $d = 5$

$d = 5$ Graph	$\|V\|$	Best Known (* – *optimal value*)	AB-DCST	Difference $(\frac{\text{best known} - \text{our best}}{\text{best known}})$	Coeff. of Variation	Average Time (sec)
m400n1	400	54.69*	54.82	–0.24%	0.10	360
m500n1	500	79.34*	79.38	–0.05%	0.05	620
shrd1500	150	43 098	43 105	–0.02%	0.01	68
m300n1	300	40.69*	40.68	0.02%	0.10	370
R100n2	100	8.08	7.58	6.19%	0.00	21
R100n3	100	8.07	7.50	7.06%	0.00	22
R100n1	100	8.09	7.51	7.17%	0.00	23
R200n1	200	16.23	15.06	7.21%	0.00	105
R200n2	200	15.79	14.65	7.22%	0.00	88
R50n1	50	4.36	4.04	7.34%	0.00	4.38
R200n3	200	15.75	14.59	7.37%	0.00	115
R50n2	50	4.27	3.94	7.73%	0.00	4.44
R50n3	50	4.29	3.92	8.62%	0.00	4.94
random1000	1000	12 424	1760	85.83%	0.00	1684

TABLE VIII
INSTANCES WHERE AB-DCST PERFORMS BETTER THAN THE BEST KNOWNS AND HAVE LOWER BOUNDS

Graph	Degree	Best Known	Lower Bound	AB-DCST	% Improvement
str2008	2	15 876	15820.00	15 872	7.14
crd503	2	5088	4995.97	5079	9.78
crd104	2	7557	7456.87	7541	15.98
str1500	2	14 016	13975.80	14 009	17.41
crd707	2	6474	6445.82	6467	24.84
crd100	2	7063	6991.56	7044	26.60
crd706	2	6679	6651.38	6671	28.96
crd704	2	6405	6288.89	6370	30.14
shrd1500	4	54 205	54179.30	54 196	35.02
str1005	2	8702	8693.80	8699	36.59
crd103	2	7701	7591.67	7660	37.50
str1008	2	12 433	12419.90	12 428	38.17
crd702	2	6790	6720.45	6763	38.82
str1007	2	10 531	10526.00	10529	40.00
crd102	2	7582	7501.33	7549	40.91
str1508	2	13 892	13880.20	13 887	42.37
crd502	2	5507	5445.20	5480	43.69
str1002	2	7092	7085.73	7089	47.85
str1006	2	10 532	10525.90	10 529	49.18
str1001	2	5001	4998.98	5000	49.50
crd109	2	7333	7243.30	7286	52.40
str1004	2	8704	8694.58	8699	53.08
crd108	2	7165	7119.85	7137	62.02
str1003	2	7092	7087.45	7089	65.93
str502	2	6129	6124.48	6126	66.37
crd105	2	7235	7020.86	7075	74.72

TABLE IX
COMPARISON OF BEST AND MEAN BETWEEN AB-DCST AND THE COLAPSO BY ERNST [24]

			CoLaPSO		AB-DCST	
Graph	$\|V\|$	d	Best	Mean	Best	Mean
crd501	50	2	5555.59	5598.62	**5553**	**5564**
crd501	50	3	5130.30	5130.30	**5126**	**5126**
crd700	70	2	**6306.49**	6357.71	6308	**6340**
crd700	70	3	5789.55	5789.55	5789	5789
crd100	100	2	7096.44	7173.34	**7044**	**7105**
crd100	100	3	6199.05	6199.05	**6196**	**6196**
shrd159	15	2	904.00	904.00	904	904
shrd159	15	3	597.00	597.00	597	597
shrd159	15	4	430.00	430.00	430	430
shrd159	15	5	332.00	332.00	332	332
shrd200	20	2	1679.00	1679.00	1679	1679
shrd200	20	3	1088.00	1088.00	1088	1088
shrd200	20	4	802.00	802.00	802	802
shrd200	20	5	627.00	627.00	627	627
shrd259	25	2	2714.00	2714.00	2714	2714
shrd259	25	3	1756.00	1756.00	1756	1756
shrd259	25	4	1292.00	1292.00	1292	1292
shrd259	25	5	1016.00	1016.00	1016	1016
shrd300	30	2	3992.00	3992.00	3992	3992
shrd300	30	3	2592.00	2592.00	2592	2592
shrd300	30	4	1905.00	1905.00	1905	1905
shrd300	30	5	1506.00	1506.00	1504	1504
sym500	50	2	1759.00	**1767.74**	1759	1851
sym500	50	3	1156.00	1156.00	1156	1156
sym500	50	4	1105.00	1105.00	1105	1105
sym500	50	5	1098.00	1098.00	1098	1098
sym701	70	2	**1931.00**	**1931.00**	1966	2169
sym701	70	3	1270.00	1270.00	1270	1270
sym701	70	4	1198.00	1198.00	1198	1198
sym701	70	5	1186.00	1186.00	1186	1186
str500	50	2	4420.00	4420.00	4420	4420
str500	50	3	4118.00	4118.00	4118	4118
str500	50	4	3956.00	3956.00	3956	3956
str500	50	5	3807.00	3807.00	3807	3807
str700	70	2	4696.00	4700.92	**4693**	**4696**
str700	70	3	4397.00	4397.00	4397	4397
str700	70	4	4245.00	4245.00	4245	4245
str700	70	5	4100.00	4100.00	4100	4100
str100	100	2	5003.00	5018.26	**5000**	**5007**
str100	100	3	4702.00	4702.00	4702	4702
str100	100	4	4546.00	4546.00	4546	4546
str100	100	5	4403.00	4403.00	4403	4403

values for these tests for different classes of graphs as well as for all graphs. As the p-values are much smaller than 0.05 for all but the SHRD class the differences between the AB-DCST and No-Ants; and between AB-DCST and Perturbation are statistically significant. Furthermore, since these are one-sided t-tests with the alternative hypothesis being the mean (the best) of the simplified algorithms greater than that of the AB-DCST algorithm, we can conclude that the exploration phase of the ants does provide a statistically significant contribution to the performance of the algorithm. It has been observed often in practice that the performance of local optimization heuristics is usually dependent on the quality of the initial solutions, i.e., local optimization heuristics usually yield better solutions if they start out with a good solution. It seems from our statistical tests that the exploration phase of the ants does help provide the local optimization algorithms with a good starting point.

It is also natural to ask whether both 2-EdgeReplacement and 1-EdgeReplacement local optimization algorithms were

TABLE X
COMPARISON OF MEAN GAP BETWEEN AB-DCST AND CB-TCR BY SOAK *et al.* [29] ON THE M GRAPHS WITH $d = 5$

	$\frac{c_{\text{mean}}-c_{\text{opt}}}{c_{\text{opt}}} \times 100$	
Graph	CB-TCR	AB-DCST
m50n1	0.00	0.00
m50n2	0.00	0.00
m50n3	0.00	0.00
m100n1	0.07	0.00
m100n2	0.12	0.00
m100n3	0.32	0.10
m200n1	1.71	0.38
m200n2	1.42	0.16
m200n3	2.25	0.00

TABLE XI
p-VALUES OF NONPARAMETRIC PAIRED t-TEST (WILCOXON) BETWEEN AB-DCST AND NO-ANTS AND BETWEEN AB-DCST AND PERTURBATION

Graph	AB-DCST Versus No-Ants		AB-DCST Versus Perturbation	
Class	Mean	Best	Mean	Best
All	<2.22E-16	<2.22E-16	<2.22E-16	<2.22E-16
CRD	3.29E-7	3.87E-8	3.59E-6	2.53E-3
M	1.20E-9	3.78E-8	8.37E-8	5.79E-7
R	1.73E-4	2.08E-6	2.93E-5	4.87E-4
RANDOM	4.58E-4	2.46E-4	7.34E-4	7.34E-4
SHRD	5.24E-1	2.26E-2	6.13E-1	3.10E-2
STR	<2.22E-16	<2.22E-16	3.89E-6	1.44E-4
SYM	4.36E-14	8.28E-14	8.65E-7	5.16E-5

necessary. That is, can one of them be removed? We have conducted comparative experiments using four graphs from the SYM class with degree constraint equals to 2. The environment of the experiments is the same as the one described in Section IV and the algorithm is run 50 times for each graph. The results are shown in Table XII, where columns 2&1, 1, and 2 show results when both 2-EdgeReplacement and 1-EdgeReplacement algorithms are used, only 1-EdgeReplacement is used, and only 2-EdgeReplacement is used, respectively. From the table, it is clear that the algorithm with both 2-EdgeReplacement and 1-EdgeReplacement gets better results and runs faster than algorithms with only 1-EdgeReplacement or 2-EdgeReplacement. For these graphs, it is the case that better results are obtained when both local optimization algorithms are used. The intuition behind the better results is that 2-EdgeReplacement is somewhat more disruptive and provides an opportunity for the algorithm to explore more and also a better chance of getting out of a local optimum. On the other hand, 1-EdgeReplacement is less disruptive and allows for more exploitation and fine tuning of the current configuration. Therefore, the combination of the two local optimization algorithms, in the order that they are applied, allows the algorithm to obtain superior results than either of the local optimization algorithms by themselves.

We also observe that the algorithm takes significantly less time when both local optimization algorithms are used. We observed previously that 2-EdgeReplacement has a time complexity of $O(|V|)$ for each replacement and 1-

TABLE XII
COMPARISON WITH ONLY 1-EDGEREPLACEMENT AND 2-EDGEREPLACEMENT LOCAL OPTIMIZATIONS

Graph	$\|V\|$	Best Cost			Avg. Time (s)		
		2&1	1	2	2&1	1	2
sym500	50	1759	1844	1814	5.06	31	31
sym503	50	1991	2047	2013	5.02	31	28
sym700	70	2093	2202	2139	10	89	63
sym706	70	1522	1529	1560	11	114	56

EdgeReplacement is $O(|V|^2)$ for each replacement. Intuitively, 2-EdgeReplacement will push the result very close to some local optimum and pass the baton to 1-EdgeReplacement. The algorithm with only 2-EdgeReplacement runs faster each iteration but take more iterations to reach a good local optimum without the final push from 1-EdgeReplacement. As for the algorithm with only 1-EdgeReplacement, it takes much longer for each iteration. Therefore, the combination (first 2-EdgeReplacement and then 1-EdgeReplacement) will make the algorithm reach a local optimum faster than 1-EdgeReplacement or 2-EdgeReplacement alone and ultimately have shorter running time.

VI. CONCLUSION

We presented an ant-based algorithm (AB-DCST) for the classic DCMST problem. The ants are used to select a subset of candidate edges from which a low cost DCST is built. Then two local optimization algorithms are applied to further improve the DCST. Our extensive experimental study, which uses most of popular problem instances, demonstrates the effectiveness of the algorithm. More precisely, AB-DCST found either the best known results or better results in 86.8% the instances and worse results in 4.4% of the instances when compared against current best known (the remaining 8.8% has no known result). For a small subset of the problem instances when the mean results are available, we found that AB-DCST also had similar performance when compared against these results using the mean.

The algorithm also exhibits a fairly stable behavior. Specifically, for the 612 instances that we have, the coefficient of variation (based on 50 runs for each instance) is zero for 74% of the instances, between 0 and less than 1.00 for 21% of the instances and between 1.00 and 4.09 for 5% of the instances.

For future work, we plan to investigate the parallelization of the algorithm. For example, the exploration phase where ants explore the graph is an ideal candidate for parallelization.

ACKNOWLEDGMENT

The authors would like to thank J. Knowles, G. Raidl, A. Ernst, and K. Honda for providing their test graphs, and in particular, A. Ernst for supplying the current best known results for many problem instances. They also would like to thank J. Zhu for his help with the statistical tests done in this paper.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.
[2] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
[3] S. Narula and C. Ho, "Degree-constrained minimum spanning tree," *Comput. Oper. Res.*, vol. 7, no. 4, pp. 239–249, 1980.
[4] M. Savelsbergh and T. Volgenant, "Edge exchanges in the degree-constrained minimum spanning tree problem," *Comput. Oper. Res.*, vol. 12, no. 4, pp. 341–348, 1985.
[5] J. Knowles and D. Corne, "A new evolutionary approach to the degree-constrained minimum spanning tree problem," *IEEE Trans. Evol. Comput.*, vol. 4, no. 2, pp. 125–134, Jul. 2000.
[6] R. Ravi, M. Marathe, S. Ravi, D. Rosenkrantz, and H. Hunt, III, "Many birds with one stone: Multiobjective approximation algorithms," in *Proc. 25th Annu. ACM Symp. Theory Comput.*, 1993, pp. 438–447.
[7] G. Craig, M. Krishnamoorthy, and M. Palaniswami, "Comparison of heuristic algorithms for the degree constrained minimum spanning tree," in *Metaheuristics: Theory and Applications*. Boston, MA: Kluwer, 1996.
[8] M. Krishnamoorthy, A. T. Ernst, and Y. M. Sharaiha, "Comparison of algorithms for the degree constrained minimum spanning tree," *J. Heuristics*, vol. 7, no. 6, pp. 587–611, 2001.
[9] B. Boldon, N. Deo, and N. Kumar, "Minimum-weight degree-constrained spanning tree problem: Heuristics and implementation on an SIMD parallel machine," *Parallel Comput.*, vol. 22, no. 3, pp. 369–382, 1996.
[10] G. Zhou and M. Gen, "A note on genetic algorithms for degree-constrained spanning tree problems," *Networks*, vol. 30, no. 2, pp. 91–95, 1998.
[11] G. Raidl and B. Julstrom, "A weighted coding in a genetic algorithm for the degree-constrained minimum spanning tree problem," in *Proc. ACM Symp. Appl. Comput.*, vol. 1. 2000, pp. 440–445.
[12] G. Raidl and B. Julstrom, "Edge sets: An effective evolutionary coding of spanning trees," *IEEE Trans. Evol. Comput.*, vol. 7, no. 3, pp. 225–239, Jun. 2003.
[13] G. Raidl, "An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem," in *Proc. IEEE CEC*, vol. 1. Jul. 2000, pp. 104–111.
[14] S.-M. Soak, D. Corne, and B.-H. Ahn, "A powerful new encoding for tree-based combinatorial optimization problems," in *Proc. PPSN*, LNCS 3242. 2004, pp. 430–439.
[15] T. N. Bui, T. H. Nguyen, and J. R. Rizzo, Jr., "Parallel shared memory strategies for ant-based optimization algorithms," in *Proc. Genet. Evol. Comput.*, 2009, pp. 1–8.
[16] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Trans. Syst. Man Cybern. Part B*, vol. 26, no. 1, pp. 29–41, Feb. 1996.
[17] M. Dorigo and G. Di Caro, "Ant colony optimization: A new meta-heuristic," in *Proc. IEEE CEC*, Jul. 1999, pp. 1470–1477.
[18] M. Dorigo, G. Di Caro, and L. M. Gambardella, "Ant algorithms for discrete optimization," *Artif. Life*, vol. 5, no. 2, pp. 137–172, 1999.
[19] A. Volgenant, "A Lagrangian approach to the degree-constrained minimum spanning tree problem," *Eur. J. Oper. Res.*, vol. 39, no. 3, pp. 325–331, 1989.
[20] Y.-T. Bau, C. K. Ho, and H. T. Ewe, "An ant colony optimization approach to the degree-constrained minimum spanning tree problem," in *Proc. CIS*, LNCS 3801. 2005, pp. 657–662.
[21] T. N. Bui and C. M. Zrncic, "An ant-based algorithm for finding degree-constrained minimum spanning tree," in *Proc. Genet. Evol. Comput.*, 2006, pp. 11–18.
[22] M. Doan, "An effective ant-based algorithm for the degree-constrained minimum spanning tree problem," in *Proc. IEEE CEC*, Sep. 2007, pp. 485–491.
[23] H. Binh and T. Nguyen, "New particle swarm optimization algorithm for solving degree constrained minimum spanning tree problem," in *Proc. Pacific Rim Int. Conf. Artif. Intell. Trends Artif. Intell.*, 2008, pp. 1077–1085.
[24] A. T. Ernst, "A hybrid Lagrangian particle swarm optimization algorithm for the degree-constrained minimum spanning tree problem," in *Proc. IEEE CEC*, Jul. 2010, pp. 1–8.
[25] T. N. Bui and B. Moon, "Genetic algorithm and graph partitioning," *IEEE Trans. Comput.*, vol. 45, no. 7, pp. 841–855, Jul. 1996.
[26] T. Stützle and H. Hoos, "MAX-MIN ant system," *Future Generation Comput. Syst.*, vol. 16, no. 8, pp. 889–914, 2000.
[27] A. Delbem, A. de Carvalho, C. Policastro, A. Pinto, K. Honda, and A. Garcia, "Node-depth encoding for evolutionary algorithms applied

to network design," in *Proc. Genet. Evol. Comput.*, LNCS 3102. 2004, pp. 678–687.

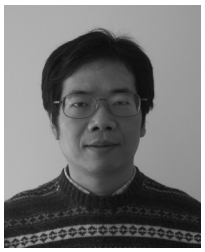
[28] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Modeling Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[29] S.-M. Soak, D. W. Corne, and B.-H. Ahn, "The edge-window-decoder representation for tree-based problems," *IEEE Trans. Evol. Comput.*, vol. 10, no. 2, pp. 124–144, Apr. 2006.

[30] R. V. Hogg and E. Tanis, *Probability and Statistical Inference*, 8th ed. Englewood Cliffs, NJ: Prentice-Hall, 2009.

Thang N. Bui received the B.S. degrees (both with honors) in mathematics and electrical engineering from Carnegie Mellon University, Pittsburgh, PA, in 1980, and the S.M. and Ph.D. degrees in computer science from the Massachusetts Institute of Technology, Cambridge, in 1983 and 1986, respectively.

He is currently an Associate Professor with the Department of Mathematics and Computer Science, Pennsylvania State University at Harrisburg, Middletown, PA. His current research interests include optimization algorithms and evolutionary computation.



Xianghua Deng received the M.S. and Ph.D. degrees in computer science from Kansas State University, Manhattan, KS, in 2002 and 2007, respectively.

He is currently a Software Engineer with Google, Inc., Mountain View, CA. His current research interests include strong formal methods and evolutionary computation.

Catherine M. Zrncic received the B.S. degree in computer science from Millersville University, Millersville, PA, in 1998, and the M.S. degree in computer science from Pennsylvania State University, Harrisburg, PA in 2006.

She has been an Adjunct Professor with the Department of Mathematics and Computer Science, Pennsylvania State University at Harrisburg, Middletown, PA, and Dickinson College, Carlisle, PA, since 2009 and 2010, respectively. Her other areas of experience include software development and project management. Her current research interests include parallel programming and ant-based algorithms.