An Ant-Based Algorithm for Finding Degree-Constrained Minimum Spanning Tree

Thang N. Bui and Catherine M. Zrncic
Computer Science Program
The Pennsylvania State University at Harrisburg
Middletown, PA 17057
{tbui, cmg212}@psu.edu

ABSTRACT

A spanning tree of a graph such that each vertex in the tree has degree at most d is called a degree-constrained spanning tree. The problem of finding the degree-constrained spanning tree of minimum cost in an edge weighted graph is well known to be NP-hard. In this paper we give an Ant-Based algorithm for finding low cost degree-constrained spanning trees. Ants are used to identify a set of candidate edges from which a degree-constrained spanning tree can be constructed. Extensive experimental results show that the algorithm performs very well against other algorithms on a set of 572 problem instances.

Categories and Subject Descriptors: G.2.2[Discrete Mathematics]:Graph Theory – Graph algorithms, Network problems, Trees; I.2.8[Artificial Intelligence]:Problem Solving, Control Methods, and Search – Heuristic methods

General Terms: Design, Algorithms

Keywords: Ant Algorithm, Degree Constrained Spanning Tree

1. INTRODUCTION

This paper describes an Ant-Based algorithm for the degree-constrained minimum spanning tree (DCMST) problem. This is an interesting, real-world problem that seems well suited to an ant algorithm approach. Numerous genetic algorithms have been used to solve the DCMST problem, but to our knowledge this is the first ant algorithm for this problem.

The DCMST problem entails finding a spanning tree of minimum cost such that no vertex in the tree exceeds a given degree constraint. This concept is useful in designing networks for everything from computer and telephone communications to transportation and sewage system [12] [18]. For instance, switches in an actual communication network will each have a limited number of connections available. Transportation systems must place a limit on the number of roads meeting in one place. Also, limiting the degree of each node limits the potential impact if a node fails [10].

While the unconstrained minimum spanning tree (MST) problem can be solved easily in polynomial time, the DCMST problem is NP-hard [9]. In fact, even approximating optimal DCMST solutions within a constant factor is NP-hard [1] [10]. Therefore, heuristics are often used to find good solutions in a reasonable amount of time. Recent work on this problem has focused on genetic algorithms (GAs) and finding more efficient representations to use with those algorithms.

One type of heuristic that has not been previously used to solve the DCMST problem is the ant algorithm. Ant algorithms have been used to solve many optimization problems including other types of constrained MST problems such as the capacitated MST [17], generalized MST [19], and k-cardinality tree [4] problems.

There are several different types of ant algorithms. The most common is Ant Colony Optimization (ACO), while our approach is an Ant-Based (AB) algorithm. In both cases, artificial ants maneuver based on local information and deposit pheromones as they travel. Our algorithm then uses cumulative pheromone levels to determine candidate sets of edges from which degree-constrained spanning trees are built. Extensive experimental results show that our Ant-Based algorithm finds results that are generally better than results produced by existing GAs. The addition of a local optimization step, which is not implemented in our current algorithm, could further enhance performance, particularly for graphs designed to mislead greedy algorithms.

The rest of the paper is organized as follows. In Section 2 we give a formal definition of the problem and describe previous works. Our Ant-Based algorithm is described in Section 3. Section 4 compares the performance of our algorithm against existing algorithm on a set of benchmark problems. The conclusion is given in Section 5.

2. PRELIMINARIES

A *spanning tree* of a graph G is a subgraph of G that is a tree and has the same number of vertices as G. An undirected graph G is said to be *complete* if there is an edge between any pair of distinct vertices, i.e., no self-loop is allowed. The *degree constrained minimum spanning tree* (DCMST) problem can be formally defined as follows.

Input: An undirected complete graph $G = (V, E)$ with non-negative weights on the edges and an integer degree constraint $d \geq 2$.

Output: A spanning tree of G with minimum total weight such that the degree of each vertex in the tree is $\leq d$.

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 [9]. 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 [16].

Early heuristics for the DCMST problem used a branch and bound approach [12][18]. Other heuristics developed since then have included hill-climbing, simulated annealing, neural networks, and additional branch and bound algorithms [1][11]. Genetic algorithms (GA) were first applied to the DCMST problem in the mid-1990's [24], and that has been the focus of work on the DCMST problem to date.

Zhou and Gen [24] used Prüfer codes to encode spanning trees for their GA. In 2000, Knowles and Corne [10] used a different representation – storing a spanning tree in an $n \times (d-1)$ array, where n is the number of vertices and d is the degree constraint. They used this representation in a GA as well as hill-climbing and simulated annealing algorithms. Of their three algorithms, the GA found the best solutions and compared favorably to earlier algorithms. Also in 2000, GAs using weighted [14] and edge-set [13] encodings were introduced. These representations stored the edges and edge weights instead of a permutation of vertices. Testing showed that the weighted encoding performed better than previous algorithms, and the edge-set encoding in turn produced even better results [13][14].

In 2001, Krishnamoorthy, Earnst, and Sharaiha [11] developed a new set of heuristic algorithms for the DCMST problem. These algorithms included a GA using a Prüfer encoding, Problem Space Search (PSS – another GA using a constructive heuristic to create the initial population and a vertex-permutation encoding), and a simulated annealing algorithm. They also introduced an exact algorithm, which worked well for small graphs and was used to generate heuristic solutions for larger graphs. These algorithms were compared against each other and the optimal solutions, if known. Of the four, the PSS algorithm performed best overall [11].

A few years later, Raidl and Julstrom [15] introduced an updated version of the edge-set encoding. A GA using this representation was compared against several previous GAs, including the one from Knowles and Corne [10] and the PSS algorithm [11]. The GA using the edge-set encoding, which utilized edge-weight knowledge when constructing potential solutions, consistently performed the best and scaled up to larger problems (up to 500 vertices were tested) [15].

In 2004, another GA based on a Prüfer encoding was introduced [6]. It showed great improvement over the original DCMST GA, but it was not compared to any contemporary GAs. That algorithm was tested on graphs up to 1000 vertices. Also in 2004, Soak, Corne, and Ahn [20] introduced a new encoding for trees and compared it to the edge-set encoding from [15]. The two algorithms produced the same results for graphs of 30 and 40 vertices, while the new encoding performed better on 50 vertex graphs [20].

Our Ant-Based algorithm was tested using the same data sets as many of these algorithms. Our solutions were compared to previously published results from each of the algorithms listed above, except 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 [20]. This final algorithm was not used because the specific graphs tested in that paper were not available for comparison. The other algorithms were excluded because more recent algorithms have produced better results. Those comparisons are detailed in Section 4.

3. ALGORITHM

Ant algorithms are usually based on the observed behavior of a colony of ants to collectively solve problems such as nest building or food foraging. In traditional ant systems such as Ant Colony Optimization, each ant solves the entire problem and then leaves pheromone on the solution configuration that it found as a guide for the next ant [8]. In contrast, each ant in our Ant-Based algorithm finds only part of the solution. A complete solution to the problem is then obtained from all the parts discovered by the ants. This difference allows ants in AB algorithms to consider smaller, more localized sections of the problem, making AB algorithms well-suited to distributed or parallel programming.

Our Ant-Based algorithm for the DCMST problem, called AB-DCST, consists of a number of cycles. Each cycle has two stages: **exploration** and **construction**. In the exploration stage of each cycle ants are used to discover a set of candidate edges from which a degree-constrained spanning tree is constructed. In the construction stage we build such a tree. Thus, each cycle produces a degree constrained spanning tree. The smallest cost tree returned by all cycles is returned by the algorithm.

In each cycle, each ant explores a local section of the graph. The edges an ant travels across are chosen such that edges with a higher level of pheromone have a higher chance of being selected. The pheromone on an edge is increased when an ant travels across that edge, leaving an increasingly appealing trail for other ants to follow. It should be noted that for an edge (i, j), the initial amount of pheromone distributed at (i, j) is the same as the amount of pheromone an ant leaves on (i, j) when the ant traverses it. Furthermore, the lower the edge cost is the higher this pheromone amount is. Effectively, ants will have a tendency to travel along lower cost edges as well as edges that have been traveled more often by other ants.

After all ants have explored their local section of the graph, meaning each ant has traveled across a set number of edges, the entire graph is examined. The edges with the highest pheromone levels, which are the edges ants traveled across most often, are selected as a candidate set of edges. A degree-constrained spanning tree is then built from these candidate edges in a greedy manner. Figure 1 shows the full algorithm. In the following subsections we will describe the algorithm in more detail.

3.1 Initialization

Initially one ant is assigned to each vertex of the graph. As the algorithm progresses, ants move about and each vertex may have zero or more ants. The next step in the initialization process is to calculate the initial pheromone level of each edge. Let (i, j) be an edge in the graph. The initial pheromone level assigned to (i, j), denoted by $IP(i, j)$, is

```
AB-DCST($G = (V, E), w, d$)
// $w$ is the edge cost function
// $d$ is the degree constraint
//Initialization State
  assign one ant to each vertex of the graph
  initialize pheromone level of each edge
  $B \longleftarrow \emptyset$    // best tree
  $cost(B) \longleftarrow \infty$
  while stopping criteria not met
    // Exploration Stage
    for $step = 1$ to $s$
      if $step = s/3$ or $step = 2s/3$
        update pheromone levels for all edges
      end-if
      for each ant $a$
        move $a$ along one edge
      end-for
    end-for
    update pheromone levels for all edges
    // Tree Construction Stage
    while $|T| \neq n - 1$
      identify a set $C$ of candidate edges
        using pheromone levels
      construct degree-constrained spanning tree $T$ from $C$
    end-while
    if $cost(T) < cost(B)$
      $B \longleftarrow T$
    end-if
    enhance pheromone levels for edges in the best tree $B$
    if no improvement in 100 cycles
      evaporate pheromone from edges of the best tree $B$
    end-if
  end-while
  return the best tree found $B$
```

Figure 1: The AB-DCST algorithm

defined as follows.

$$IP(i, j) = (M - c(i, j)) + (M - m)/3,$$

where $c(i, j)$ is the cost of the edge (i, j), and M and m are the maximum and minimum edge costs in the graph, respectively. Note that edges that have smaller cost will have a higher initial pheromone level. As defined, the highest initial pheromone level is no larger than 4 times the lowest initial pheromone level. This scaling method is similar to that of [2]. Scaling methods such as this prevent extremely large gaps in the initial pheromone levels even when such large gaps exist in the edge costs. Generally, it is useful to have a large enough difference between the pheromone levels so that good and bad features identified by pheromone can be differentiated. However, extremely large differences in pheromone levels can cause unwanted bias easily.

We also use $IP(i, j)$ as the amount of pheromone that an ant adds to the edge (i, j) whenever it traverses (i, j). Thus, the lower the cost of an edge is, the more the pheromone level is increased every time that edge is traversed.

3.2 Exploration

In this section we describe how ants move. The objective of the ants is to help identify a set of edges from which we can construct a good degree-constrained spanning tree. The main ideas here are: (i) ants should move along edges that have low cost and (ii) ants should utilize information from other ants in deciding on where to go to next, i.e., which edge to traverse. These ideas can be accomplished by letting each ant lay a certain amount of pheromone whenever it moves across an edge. The amount of pheromone laid on an edge should depend on the cost of that edge so that the lower the edge cost is the higher the amount pheromone will be. An ant can then select an edge to traverse by examining the edges that are incident to the vertex that it is currently on and selecting one of those edges based on pheromone levels. Specifically, the ant selects one of these edges at random such that edges with higher level of pheromone have a higher probability of being selected.

This method of proportional selection ensures that ants favor high-pheromone (i.e., low cost) edges while still allowing low-pheromone edges to be selected occasionally. This is important because a good solution for the overall graph may need to include an edge that appears to be a bad choice from a local perspective. If the selected edge leads to a vertex the ant has already visited during the current cycle, the ant selects a different edge. If the ant tries 5 times and all 5 vertices have been visited, the ant does not move for that step and the algorithm continues. This limit is used as a balance between allowing an ant to find a new vertex and preventing the ant from spending too much time on that search, and is less expensive than maintaining lists of feasible edges for each ant. This process of selecting one edge to traverse is called a **step**, and is summarized in Figure 2

```
Move($a, i$)       // ant $a$ is at vertex $i$
$nAttemps \longleftarrow 0$
while $nAttempts < 5$
  select an edge $(i, j)$ at random and
    proportional to its pheromone level
  if vertex $j$ is unvisited
    mark edge $(i, j)$ for pheromone update
    move $a$ to vertex $j$
    mark $j$ visited
    break
  else
    $nAttempts++$
  end-if
end-while
```

Figure 2: One step in the ant movement algorithm

After an ant traverses an edge, that edge is marked for pheromone update. The actual update is done at certain points in the algorithm as shown in Figure 1. This update is done periodically, instead of after every move, in order to provide ants with sufficiently accurate information without sacrificing running time efficiency. The pheromone level of an edge is modified by two factors. First, the current pheromone level of the edge is decreased through evaporation. This prevents ants from relying on old information for too long a time, as the pheromone levels on edges that have not been selected recently will evaporate to insignificant levels. Second, the pheromone level of the edge is increased by an amount equal to the initial level of pheromone assigned to that edge multiplied by the number of ants that traversed the edge since the last pheromone update. Some other types of ant algorithms, such as Ant Colony System [8], also use a local pheromone update.

Before specifying how pheromone levels are updated, we define the maximum and minimum allowable levels of pheromone. These are denoted by *maxP* and *minP*, respectively.

$$maxP = 1000((M-m)+(M-m)/3) \quad \text{and}$$
$$minP = (M-m)/3,$$

where M and m denote the maximum and minimum edge costs in the graph. A similar strategy is used in the Max-Min Ant System [21]. As shown below, we do not reset pheromone levels exactly to these boundaries. Rather, we offset the adjusted pheromone level of an edge by the initial pheromone for that edge. This helps ants distinguish good edges from bad, even if the pheromone levels for a number of edges are approaching *maxP* or *minP*.

Let $P(i,j)$ denote the pheromone level of the edge (i,j) and $u(i,j)$ be the number of delayed pheromone updates to be applied to edge (i,j). Then $P(i,j)$ is updated as follows.

1. $P(i,j) = (1-\eta)P(i,j) + u(i,j)IP(i,j)$
2. If $P(i,j) > maxP$ then $P(i,j) = maxP - IP(i,j)$, else if $P(i,j) < minP$ then $P(i,j) = minP + IP(i,j)$

where η is the evaporation factor and $IP(i,j)$ is given in Section 3.1. The evaporation factor η is assigned an initial value of 0.5. It is then gradually decreased as the algorithm progresses. This is so that the algorithm has more chance to explore in the beginning and will be able to converge later on. It is clear from the definition of $IP(i,j)$ and $P(i,j)$ that they both depend on the cost of the edge (i,j).

3.3 Tree Construction

After the ants have completed their movements and the pheromone levels of all the edges have been updated, we are ready to identify the edges from which to construct a degree-constrained spanning tree. To identify a set of candidate edges, we first sort the edges in the graph in the order of decreasing pheromone level. The top *nCandidates* edges from this sorted list are selected to form a candidate set C. We next sort the edges in C in increasing order of edge cost. A degree-constrained spanning tree is then constructed using a version of the Kruskal's algorithm [5], modified to enforce the degree constraint. The algorithm is given in Figure 3.

Occasionally, the spanning tree cannot be completed using only the candidate edges. In that case, additional edges are selected until the full tree has been built. This is done by selecting the next *nCandidates* edges from the pheromone-sorted list and repeating the process. This continues until the spanning tree is complete.

```
ConstructTree($G = (V, E), w, d$)
sort all edges by pheromone level in descending order
$C \longleftarrow$ top nCandidates edges (highest pheromone levels)
sort $C$ by cost into ascending order
$T \longleftarrow \emptyset$
while $|T| \neq n-1$
  if $C \neq \emptyset$
    let $(i,j)$ be the next edge in order if increasing cost
    remove $(i,j)$ from $C$
    if $i$ and $j$ are not connected in $T$
    and adding $(i,j)$ would not violate the degree constraint
      add $(i,j)$ to $T$
    end-if
  else // $C$ is empty
    add to $C$ the next nCandidates edges
      in the order of decreasing pheromone level
    sort $C$ into increasing order of edge cost
  end-if
end-while
return $T$
```

Figure 3: Tree Construction

3.4 Pheromone Enhancement

The total cost of the spanning tree returned by the ConstructTree algorithm is compared to the best solution found up to that point. If the new cost is lower, that tree is saved as the new best solution. Then the pheromone levels for the best edges, i.e., those in the lowest-cost tree found so far, are enhanced to promote exploitation of those good edges. This is related to the global pheromone update found in other types of ant algorithms [7][8][21]. Specifically, the pheromone levels of edges in the current best tree are updated as follows. Let (i,j) be an edge in the best tree found so far.

1. $P(i,j) = \gamma P(i,j)$
2. If $P(i,j) > maxP$ then $P(i,j) = maxP - IP(i,j)$, else if $P(i,j) < minP$ then $P(i,j) = minP + IP(i,j)$

where $IP(i,j)$ is given in Section 3.1, and γ is the enhancement factor. Initially, γ is assigned a value of 1.5 and it is gradually increased as the algorithm progresses. The intuition for this gradual growth of the enhancement factor is that spanning trees found in the beginning are not expected to be good ones, thus we should not enhance too much the pheromone levels of edges in such trees. On the other hand, as the algorithm progresses it is expected to find better trees, and hence their edges should be enhanced more.

To prevent convergence on a local optimum, this enhancement is reversed if there is no improvement after a set number of iterations. This escape is done by evaporating the pheromone from the best edges to bring them back into balance so other edges have a better chance of being selected. The pheromone level of edges in the current best tree is updated as above, with γ replaced by an evaporation factor ζ, which is chosen at random from the interval $[0.1, 0.3]$.

3.5 Stopping Criteria

The algorithm stops if one of the following two conditions is satisfied: (i) there is no improvement found in 2,500 consecutive cycles, or (ii) it has run for 10,000 cycles. When the algorithm stops, the current best tree is returned.

3.6 Parameters

Table 1 shows the parameters used when running this algorithm. Two of the values, the factors for evaporation and enhancement of pheromone levels, change over time. Initially, evaporation is high and enhancement is low. As explained above, this allows for exploration of the graph in early cycles. As the algorithm progresses, the evaporation factor is decreased while the enhancement factor is increased. This encourages exploitation as the solutions begin to converge on good trees. The remaining parameters remain fixed for the duration of the algorithm.

These parameters were not tuned to any specific graph or class of graphs. They were determined based on comparative testing using a set of three graphs that were chosen randomly to cover a range of graph types and sizes. This test set of graphs was not included in the reported results.

Table 1: AB-DCST Algorithm Parameters

Parameter	Value	Comments
maxCycles	10,000	Maximum allowed cycles
s	75	Steps: number of edges an ant traverses each cycle
nCandiates	$5n$	Candidate set size
a	n	Number of ants
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
updateCycles	500	Number of cycles between updating η and γ
updateSteps	$s/3$	Number of steps between applying pheromone updates
escapeCycles	100	Number of cycles without improvement before escaping
stopCycles	2,500	Number of cycles without improvement before stopping

4. EXPERIMENTAL RESULTS

4.1 Data Set

Our algorithm was run on a set of 143 complete graphs ranging from 15 to 1000 vertices, with degree constraints of 2 through 5, giving a total of 572 problem instances. Seven different types of graphs were used and are summarized in Table 2. We compared our results with previously published results for these graphs. For each problem instance in our test set, we used the best result found by any previous algorithm for our comparison. It should be noted that no single paper had data for all of the graphs used here.

These graphs fall into two categories, Euclidean and non-Euclidean graphs. Euclidean graphs represent coordinate points with edge costs being the Euclidean distances between two points. Three types of Euclidean graphs were tested: CRD, SYM, and STR graphs. The CRD graphs contain points in a basic 2-dimensional plane. The points in the SYM graphs come from higher dimensional space, increasing the chance that nodes in the unconstrained MST will violate the degree constraint. The points in the structured (STR) graphs, which are also from higher dimensional space, are grouped in clusters. The CRD and SYM graphs were used in several papers including [23] and [11], where the STR graphs were introduced.

The remaining graphs do not correspond to Euclidean points and distances. The first type in this category is the purely random graph. Vertices simply have ID numbers (vertex 1, 2, etc.) and the edge weights are chosen randomly from the interval $[1, n]$ [6]. However, like Euclidean graphs, these random graphs rarely have unconstrained MSTs with high-degree vertices [1].

The last three sets of test graphs were structured to pose a greater challenge to DCMST algorithms. The structured-hard (SHRD) graphs were created by assigning non-Euclidean costs in such a way that the number of optimal solutions is limited [11]. The random-hard (R) and misleading-hard (M) graphs were created by building the unconstrained MST to contain star patterns with high degrees. Additionally, the M graphs contain edges specifically intended to mislead greedy algorithms. The procedure for creating R graphs was introduced in [1] and used by Knowles and Corne in [10], who developed the method for creating the M graphs.

Table 2: Data Set

Type	# Used	Vertex Range
CRD	40	30 to 100
SYM	29	30 to 70
STR	35	30 to 100
SHRD	7	15 to 30
Random	11	15 to 1000
R	9	50 to 200
M	12	50 to 500
TOTAL	143	15 to 1000

4.2 Results

Our Ant-Based algorithm shows a definite improvement over previous algorithms. Over the entire data set, the average gain in solution quality is 16%. Only small gains are possible for the smallest graphs, for which earlier algorithms already find very good solutions. For the middle range where the data set contains the most graphs of each size ($n = 30$ to $n = 100$), the average gain is 5%.

Our algorithm was implemented in C++ and run on a 3GHz Pentium 4 PC with 2GB of RAM running the Linux operating system. To obtain our results, the algorithm was run 50 times on each instance, with the best (lowest) cost for each instance being recorded.

Tables 3 through 9 show our results for each type of graph. Results are not given for each individual graph due to space constraints. Instead, the average results for each graph size are given for each type of graph. For all of the tables below, n is the number of vertices and d is the degree constraint. *(Prev. Best)* is the average of the best costs available from previous heuristic algorithms for each graph type and size. *AB-DCST* shows the average of our best costs for each category (type/size/constraint). *Gain(%)* shows the difference between the previous results and our results as a percentage of *(Prev. Best)*. *Avg. Cost* is the average result obtained from all runs of our AB-DCST algorithm and *Std. Dev.* is the standard deviation. Finally, *Time* is the average running time in seconds. For times that are more than 10 seconds, the values are rounded to the nearest integer.

Our results for the CRD, SYM, and STR graphs were compared against those found by the heuristic algorithms in [11]. For some categories, heuristic results were not given. In those cases our results were compared against the best known costs provided by the authors of [11]. Some of those costs have been proven to be optimal and may have been obtained using exact algorithms. Those categories are indicated with an * after the best known cost.

One issue with the CRD graphs was rounding. The graphs provided contained coordinate points and the distances had to be calculated. These distances were rounded to integer values to meet the standard indicated in [11], but small differences could have been introduced. For the SYM and STR graphs, the integer costs were provided.

The average gains for the CRD, SYM, and STR graphs were 0.07%, 1.48%, and 0.32%, respectively. These averages include categories that had to be compared against exact optimal solutions (with a potentially lower gain than would be seen when compared to heuristic results), and also include many cases where no gain was possible because both our algorithm and previous heuristics obtained the optimal solutions. The overall average gain for these three graph types was 0.55%. See Tables 3, 4, and 5.

Table 3: Data for Coordinate (CRD) Graphs

n	*d*	**(Prev. Best) AB-DCST**	**Gain** (%)	**Avg. Cost**	**Std. Dev.**	**Time** (sec)
30	2	(4107*) 4172	−1.58	4222	40.52	4.65
30	3	(3761) 3759	0.00	3769	0.00	2.85
30	4	(3765*) 3765	0.00	3765	0.00	3.92
30	5	(3765*) 3765	0.00	3765	0.00	3.87
50	2	(5380*) 5431	−0.95	5581	89.46	16
50	3	(4834) 4834	0.00	4838	15.81	15
50	4	(4819*) 4819	0.00	4830	30.18	15
50	5	(4819*) 4819	0.00	4825	24.07	15
70	2	(6622*) 6593	0.44	6853	206.87	34
70	3	(5851) 5851	0.00	5878	62.19	31
70	4	(5849*) 5849	0.00	5875	71.33	32
70	5	(5849*) 5849	0.00	5872	60.48	28
100	2	(7860*) 7607	3.22	7940	171.42	77
100	3	(6686) 6686	0.00	6702	0.66	67
100	4	(6698*) 6698	0.00	6698	0.00	67
100	5	(6698*) 6698	0.00	6698	0.00	67

Our results for the random graphs were compared against those found by the GA in [6], whose authors supplied this data set. Only small gains were possible for the smaller graphs in this set (up to 30 vertices) because the DCMSTs are very close in cost to the unconstrained MSTs, and so the previous algorithm performed well. Our average gain for random graphs with 50 to 1000 vertices was 80% (51% for all graphs in this set). See Table 6.

Previous results for the SHRD graphs were available from two recent genetic algorithms for $d = 3$ through $d = 5$ [11] [15]. Only the best results from either source were used to find the *Best Known* average. Our results for $d = 2$ were compared against the best known results provided by the authors of [11]. Our algorithm found lower-cost trees in all categories for which heuristic results were available,

Table 4: Data for Symmetric (SYM) Graphs

n	*d*	**(Prev. Best) AB-DCST**	**Gain** (%)	**Avg. Cost**	**Std. Dev.**	**Time** (sec)
30	2	(1793*) 1890	−5.41	2037	63.16	2.63
30	3	(1291) 1290	0.08	1295	4.53	2.24
30	4	(1218) 1218	0.00	1218	1.05	1.93
30	5	(1216*) 1216	0.00	1216	0.00	1.91
50	2	(2342*) 2075	11.40	2287	112.98	11
50	3	(1303) 1274	2.23	1300	21.78	7.24
50	4	(1242) 1242	0.00	1251	7.22	7.17
50	5	(1215) 1215	0.00	1221	7.28	6.91
70	2	(2546*) 2304	9.51	2593	176.25	25
70	3	(1292) 1296	−0.31	1348	47.73	20
70	4	(1228) 1227	0.08	1244	37.91	18
70	5	(1210) 1208	0.17	1218	22.54	18

Table 5: Data for Structured (STR) Graphs

n	*d*	**(Prev. Best) AB-DCST**	**Gain** (%)	**Avg. Cost**	**Std. Dev.**	**Time** (sec)
30	2	(7090*) 7102	−0.17	7109	4.29	3.40
30	3	(6775*) 6775	0.00	6778	2.04	3.22
30	4	(6577) 6575	0.03	6576	1.03	3.11
30	5	(6544) 6541	0.05	6542	0.82	3.02
50	2	(7683*) 7664	0.25	7693	14.64	16
50	3	(7636*) 7485	1.98	7489	1.97	13
50	4	(7445) 7436	0.12	7667	3.11	15
50	5	(6916) 6912	0.06	7093	1.98	15
70	2	(8135*) 8116	0.23	8159	24.00	37
70	3	(7699*) 7701	−0.02	7706	2.66	32
70	4	(7537) 7507	0.40	7508	1.24	32
70	5	(7350) 7331	0.26	7331	0.04	31
100	2	(8946*) 8845	1.13	8906	41.28	71
100	3	(8404) 8371	0.26	8375	2.70	68
100	4	(8193) 8172	0.26	8173	0.19	68
100	5	(8012) 7998	0.17	7998	0.45	68

and in all but one of the categories that were compared against overall best-known results. For all SHRD graphs, our average gain was 2%. See Table 7.

Our results for the random-hard graphs (the R graphs) were better in all cases when compared to the best solutions from [10] and [14]. The average gain for this type of graph was 7%. The greatest improvement was made for $d = 5$, with an average of 10% compared to an average gain of 4% for $d = 4$. No previous results were available for $d = 2$ or $d = 3$. See Table 8.

Our algorithm did not perform as well on the M graphs, which were designed specifically to mislead greedy algorithms. On average, the trees returned by our algorithm had 22% higher costs than those previously reported in [10] and [15]. This is not completely surprising considering that the tree construction algorithm in AB-DCST is essentially greedy, with some randomization. We expect that the addition of a local optimization step to our algorithm will improve the results in this area. No previous results were available for $d = 2, 3,$ or 4. See Table 9.

The time needed for our algorithm to complete was highly

Table 6: Data for Random Graphs

n	d	(Prev. Best) AB-DCST	Gain (%)	Avg. Cost	Std. Dev.	Time (sec)
15	2	(–) 24	–	24	0.00	0.34
15	3	(23) 23	0.0	23	0.00	0.34
15	4	(23) 23	0.0	23	0.00	0.34
15	5	(23) 23	0.0	23	0.00	0.34
20	2	(–) 49	–	50	0.75	0.88
20	3	(36) 37	−2.78	37	0.00	0.82
20	4	(36) 35	2.78	35	0.00	0.66
20	5	(35) 35	0.0	35	0.00	0.65
25	2	(–) 57	–	57	0.20	1.89
25	3	(42) 43	−3.61	43	0.00	1.61
25	4	(42) 41	1.44	41	0.00	1.48
25	5	(41) 41	0.73	41	0.00	1.61
30	2	(–) 63	–	67	1.75	5.03
30	3	(52) 52	−0.58	52	0.14	2.96
30	4	(53) 50	5.66	50	0.00	2.63
30	5	(54) 50	6.89	50	0.00	2.91
50	2	(–) 126	–	139	9.44	19
50	3	(108) 87	19.14	88	1.74	14
50	4	(112) 85	24.24	85	0.66	15
50	5	(112) 85	24.31	86	1.26	15
100	2	(–) 271	–	308	31.20	70
100	3	(477) 174	63.53	179	8.46	73
100	4	(496) 168	66.09	169	4.79	69
100	5	(509) 168	66.99	169	4.38	67
200	2	(–) 681	–	798	137.62	357
200	3	(3006) 379	87.39	382	2.24	259
200	4	(2838) 364	87.17	365	0.48	251
200	5	(2776) 360	87.03	361	0.23	244
300	2	(–) 1095	–	1272	104.03	810
300	3	(9216) 530	94.25	783	719.65	566
300	4	(9394) 500	94.68	669	556.71	490
300	5	(9407) 497	94.72	638	504.26	487
400	2	(–) 1489	–	1832	154.53	1770
400	3	(21074) 732	96.53	4260	1816.56	963
400	4	(20802) 701	96.63	4190	913.08	870
400	5	(20820) 697	96.65	4010	792.07	848
500	2	(–) 2356	–	2932	357.52	3022
500	3	(37445) 901	97.59	6887	2161.49	1553
500	4	(37519) 848	97.74	6139	1217.62	1410
500	5	(37445) 841	97.75	5739	1260.08	1420
1000	2	(–) 11310	–	15014	7111.91	7899
1000	3	(247474) 16293	93.42	23462	3401.43	2078
1000	4	(245404) 13529	94.49	19405	2994.59	2130
1000	5	(247605) 12424	94.98	18318	2890.26	2070

dependent on graph size. The smallest graphs (15 vertices) ran in about half a second, 100-vertex graphs took about 70 seconds, and approximately one hour was needed for 1000-vertex graphs. While graph type did not have a noticeable effect on running time, the degree constraint did. For each graph, our algorithm took about the same amount of time for $d = 4$ and $d = 5$, with less than 1% difference in running times. Compared to $d = 5$, our algorithm ran 5% longer for $d = 3$ and almost 20% longer for $d = 2$. If $d = 2$ is not considered, the average running times are not changed for $n \leq 100$. For larger graphs, eliminating the results for $d = 2$ reduces average running times considerably, bringing the average for $n = 1000$ down to 35 minutes. Comparisons of running times with other algorithms were not feasible due to hardware differences and the lack of running time data from some previous papers.

Table 7: Data for Structured Hard (SHRD) Graphs

n	d	(Prev. Best) AB-DCST	Gain (%)	Avg. Cost	Std. Dev.	Time (sec)
15	2	(901*) 903	−0.22	909	3.56	0.43
15	3	(592) 591	0.17	592	0.57	0.46
15	4	(432) 430	0.46	432	1.53	0.58
15	5	(337) 336	0.30	339	1.28	0.38
20	2	(1841*) 1690	8.20	1701	10.91	0.97
20	3	(1097) 1093	0.36	1101	3.20	1.18
20	4	(805) 802	0.37	802	0.71	1.13
20	5	(631) 630	0.16	632	1.45	1.17
25	2	(2969*) 2715	8.36	2732	14.82	1.77
25	3	(1808) 1757	2.82	1777	9.33	3.31
25	4	(1294) 1284	0.77	1287	1.88	2.62
25	5	(1015) 1008	0.69	1011	1.79	2.70
30	2	(4560*) 4009	12.08	4051	51.86	2.66
30	3	(2592) 2594	−0.04	2619	21.37	5.72
30	4	(1905) 1905	0.00	1914	13.48	4.17
30	5	(1504) 1506	−0.13	1518	13.97	4.43

Table 8: Data for Random Hard (R) Graphs

n	d	(Prev. Best) AB-DCST	Gain (%)	Avg. Cost	Std. Dev.	Time (sec)
50	2	(–) 5.61	–	5.82	0.11	17
50	3	(–) 4.66	–	4.70	0.05	15
50	4	(4.46) 4.29	3.88	4.32	0.05	15
50	5	(4.48) 3.99	11.12	4.04	0.08	15
100	2	(–) 10.23	–	10.45	0.20	81
100	3	(–) 8.69	–	8.72	0.01	67
100	4	(8.41) 8.05	4.29	8.07	0.01	67
100	5	(8.43) 7.54	10.51	7.55	0.00	67
200	2	(–) 19.84	–	20.39	0.97	328
200	3	(–) 17.17	–	20.20	4.51	302
200	4	(16.23) 15.97	1.58	16.91	2.01	175
200	5	(16.29) 15.09	7.38	16.14	1.86	260

Table 9: Data for Misleading Hard (M) Graphs, $d = 5$

n	(Prev. Best) AB-DCST	Gain (%)	Avg. Cost	Std. Dev.	Time (sec)
50	(5.96) 6.07	−1.81	6.88	0.41	18
100	(10.87) 12.50	−15.00	14.92	1.16	74
200	(17.88) 20.83	−16.54	25.52	4.43	263
300	(40.71) 52.71	−29.47	54.34	0.91	293
400	(55.24) 75.40	−36.50	84.28	12.07	1144
500	(80.21) 104.51	−30.30	118.77	17.67	1886

5. CONCLUSION

Our experiments show that an ant algorithm can be successfully applied to the DCMST problem. These encouraging results lead to other questions that can be investigated in the future. For instance, the DCMST problem with a degree constraint of 2 is equivalent to the weighted Hamiltonian Path problem [1] and similar to the Traveling Salesman Problem (TSP) [18]. Therefore, results for $d = 2$ have not generally been published for previous algorithms. Our algorithm performed well compared to the best-known solutions available for $d = 2$, so it would be interesting to see if there was indeed an improvement over previous algorithms and also how it compares to algorithms specifically designed to solve Hamiltonian Path.

Further investigation into the difficulty of finding good solutions for different degree constraints could also be useful. Research on the DCMST problem has generally stopped at $d = 5$. This boundary is apparently based on the fact that the unconstrained MST for a single-dimension Euclidean graph has $d \leq 5$, but that does not apply to random or higher-dimensional Euclidean graphs. While the problem is known to be NP-Hard for any fixed constraint [9], there is at least one crossover point where the problem becomes easy, namely $d = n - 1$, which produces the unconstrained MST. It would be interesting to see if there is a lower crossover.

An important follow-up would be to add local optimization to our algorithm in an attempt to obtain better performance on graphs where the underlying MST is significantly different from the DCMST. In particular, we believe this will help improve the performance of our algorithm on the M graphs. Finally, the structure of our Ant-Based algorithm appears to be conducive to a parallel implementation. Using a parallel algorithm should reduce the running time, making it possible to find solutions for larger and larger graphs.

6. ACKNOWLEDGEMENTS

We would like to thank Joshua Knowles, Gunther Raidl, Andreas Ernst, and Karen Honda for providing their test graphs. We also would like to thank the anonymous reviewers for their helpful comments.

7. REFERENCES

[1] Boldon, B., N. Deo, and N. Kumar, "Minimum-Weight Degree-Constrained Spanning Tree Problem: Heuristics and Implementation on an SIMD Parallel Machine," *Parallel Computing*, 22, 1996, pp. 369–382.

[2] Bui, T. N. and B. R. Moon, "Genetic Algorithm and Graph Partitioning," *IEEE Trans. on Computers*, 45(7), July 1996, pp. 841–855.

[3] Bui, T. N. and J. R. Rizzo, Jr., "Finding Maximum Cliques with Distributed Ants," *GECCO 2004, Lecture Notes in Computer Science*, 3102, 2004, pp. 24–35.

[4] Bui, T. N. and G. Sundarraj, "Ant System for the k-Cardinality Tree Problem," *GECCO 2004, Lecture Notes in Computer Science*, 3102, 2004, pp. 36–47.

[5] Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, MIT Press, 2001.

[6] Delbem, C. B., A. de Carvalho, C. A. Policastro, A. K. O. Pinto, K. Honda, and A. C. Garcia, "Node-Depth Encoding for Evolutionary Algorithms Applied to Network Design," *GECCO 2004, Lecture Notes in Computer Science*, 3102, 2004, pp. 678–687.

[7] Dorigo, M., G. D. Caro, and L. M. Gambardella, "Ant Algorithms for Discrete Optimization," *Artificial Life*, 5, 1999, pp. 137–172.

[8] Dorigo, M., V. Maniezzo, and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Trans. on Systems, Man, and Cybernetics - Part B*, 26(1), Feb. 1996, pp. 29–41.

[9] Garey, M. R. and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979.

[10] Knowles, J. and D. Corne, "A New Evolutionary Approach to the Degree-Constrained Minimum Spanning Tree Problem," *IEEE Trans. On Evolutionary Computation*, 4(2), 2000, pp. 125–134.

[11] Krishnamoorthy, M., A. T. Ernst, and Y. M. Sharaiha, "Comparison of Algorithms for the Degree Constrained Minimum Spanning Tree," *Journal of Heuristics*, 7, 2001, pp. 587–611.

[12] Narula, S. C. and C. A. Ho, "Degree Constrained Minimum Spanning Tree," *Computers and Operations Research*, 7, 1980, pp. 239–249.

[13] Raidl, G. R., "An Efficient Evolutionary Algorithm for the Degree-Constrained Minimum Spanning Tree Problem," *Proc. IEEE CEC*, 2000, pp. 104–111.

[14] Raidl, G. R. and B. A. Julstrom, "A Weighted Coding in a Genetic Algorithm for the Degree-Constrained Minimum Spanning Tree Problem," *Proc. 2000 ACM Symposium on Applied Computing*, 2000, pp. 440–445.

[15] Raidl, G. R. and B. A. Julstrom, "Edge-Sets: An Effective Evolutionary Coding of Spanning Trees.," *IEEE Trans. On Evolutionary Computation*, 7(3), 2003, pp. 225–239.

[16] Ravi, R., M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III, "Many Birds with One Stone: Multi-Objective Approximation Algorithms," *Proc. of the 25th ACM Symposium on Computing (STOC)*, 1993, pp. 438–447.

[17] Reimann, M. and M. Laumanns, "A Hybrid ACO Algorithm for the Capacitated Minimum Spanning Tree Problem," *Hybrid Metaheurustucs*, 2004, pp. 1–10.

[18] Savelsbergh, M. and T. Volgenant, "Edge Exchanges in the Degree-Constrained Minimum Spanning Tree Problem," *Computers and Operations Research*, 12(4), 1985, pp. 341–348.

[19] Shyu, S. J., P. Y. Yin, B. M. T. Lin, and M. Haouari "Ant-Tree: An Ant Colony Optimization Approach to the Generalized Minimum Spanning Tree Problem," *Journal of Experimental & Theoretical Artificial Intelligence*, 15(1), 2003, pp. 103–112.

[20] Soak, S., D. Corne, and B. Ahn, "A Powerful New Encoding for Tree-Based Combinatorial Optimization Problems," *Lecture Notes in Computer Science*, 3242, 2004, pp. 430–439.

[21] Stützle, T. and H. H. Hoos, "MAX-MIN Ant System," *Future Generation Computer Systms*, 16, 2000, pp. 889–914.

[22] Tarasewich, P. and P. R. McMullen, "Swarm Intelligence: Power in Numbers," *Communications of the ACM*, 45(8), August 2002, pp. 62–67.

[23] Volgenant, A., "A Lagrangean Approach to the Degree-Constrained Minimum Spanning Tree Problem," *European Journal of Operational Research*, 39, 1989, pp. 325–331.

[24] Zhou, G. and M. Gen, "A Note on Genetic Algorithms for Degree-Constrained Spanning Tree Problems," *Networks*, 30, 1997, 91–95.