Journal of Heuristics, 7: 587–611, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

Comparison of Algorithms for the Degree Constrained Minimum Spanning Tree

MOHAN KRISHNAMOORTHY AND ANDREAS T. ERNST
CSIRO Mathematical and Information Sciences, Private Bag 10, Clayton South MDC, Clayton VIC 3169, Australia
email: Mohan.Krishnamoorthy@cmis.csiro.au; Andreas.Ernst@cmis.csiro.au

YAZID M. SHARAIHA
The Management School, Imperial College, 53 Prince's Gate, Exhibition Road, London SW7 2PG, UK
email: y.sharaiha@ic.ac.uk

Abstract

The Degree Constrained Minimum Spanning Tree (DCMST) on a graph is the problem of generating a minimum spanning tree with constraints on the number of arcs that can be incident to vertices of the graph. In this paper we develop three heuristics for the DCMST, including simulated annealing, a genetic algorithm and a method based on problem space search. We propose alternative tree representations to facilitate the neighbourhood searches for the genetic algorithm. The tree representation that we use for the genetic algorithm can be generalised to other tree optimisation problems as well. We compare the computational performance of all of these approaches against the performance of an exact solution approach in the literature. In addition, we also develop a new exact solution approach based on the combinatorial structure of the problem. We test all of these approaches using standard problems taken from the literature and some new test problems that we generate.

Key Words: degree constrained minimum spanning tree, heuristics, tree representation

1. Introduction

Consider an undirected complete graph $G = (N, A)$, where $N = \{1, \dots, n\}$ is the set of n *nodes* (or *vertices*) and $A = \{1, \dots, m\}$ is the set of m *arcs* (or *edges*) with given costs c_l for each arc $l \in A$. The *degree constrained minimum spanning tree* (DCMST) problem on G is to find a spanning tree of minimum total cost, such that the *degree* of each node is at most a given value b_i. Here the *degree* of node i is denoted as d_i and is defined as the number of arcs incident at i. We define an arc that is in the solution (to either the DCMST or a DCST) as a *branch*. All other arcs are denoted as *chords*.

By reducing it to an equivalent symmetric TSP, Garey and Johnson (1979) show that the DCMST is NP-hard. Thus it is likely that any exact solution approach will be inefficient as the size of problem instance increases. Ausiello et al. (1999) show that the Minimum Geometric 3-Degree Spanning Tree (a DCMST with degree $b_i = 3, \forall i$) admits a polynomial time approximate solution (see Arora, 1996). Ausiello et al. (1999) also state that the 4-degree spanning tree problem in 2-dimensional Euclidean space also admits a polynomial time approximation scheme, but the NP-hardness question of this problem is open (Arora, 1996). However, the 5-degree problem is solvable in polynomial-time. In general

d-dimensional Euclidean space, the 3-degree spanning tree problem is approximable within 5/3 (see Khuller, Raghavachari, and Young (1996)). For more information on complexity, performance of approximation algorithms and worst case performance guarantees see Cresenzi and Kann (1995).

In this paper, we develop a few novel solution algorithms for the DCMST. As opposed to a DCMST, a degree constrained spanning tree (DCST) on G, satisfies the degree constraints at each vertex, but need not necessarily be a minimal spanning of the nodes of G. In this paper, we develop a few heuristic methods that yield good DCST's. If there are no degree constraints, the DCMST reduces to the MST.

The problem of finding a DCMST arises frequently in the design of telecommunication and energy networks. It is also used as a subproblem in the design of networks for computer communication, transportation, sewage and plumbing. Typically, the DCMST can be applied in cases where n terminals (or nodes) need to be connected with a minimum length of an underlying transportation mode (wiring or canals or pipes). However, the handling-capacity of each of the terminals imposes a restriction on the number of wires (or arcs) that can be connected to any terminal (see Narula and Ho (1980) for additional examples). The DCMST is used by Gavish (1985) as a subproblem in the design of a centralised computer network.The DCMST may be used in the design of a road system which has to serve a collection of suburbs and has an additional restriction that no more than four roads (say) may meet at any crossing (see Savelsbergh and Volgenant (1985)). It is also possible to impose a degree constraint on vertices in communication networks, as a means of limiting the vulnerability of the network in case of vertex failure. Gavish (1982) provides other examples of practical instances of the DCMST. Gavish (1992) also provides several examples of the types of optimisation problems that are faced in the process of designing computer communication networks.

Yamamoto (1978) develops an approach for the DCMST by finding the minimum common basis of two matroids. Narula and Ho (1980) describe two greedy heuristics, and an exact, branch and bound approach (based on the lagrangean relaxation approach for the TSP that was employed by Held and Karp (1970, 1971)). Gabow (1978) presents an algorithm based on edge-exchanges on a reduced graph for the DCMST. Gavish (1982) uses a subgradient optimisation approach for deriving lower bounds that are based on a lagrangean relaxation. Savelsbergh and Volgenant (1985) develop a branch and bound method based on a lagrangean relaxation, used in conjunction with a heuristic approach and an edge elimination idea that was applied to the TSP (see Volgenant and Jonker (1983)). To date the best approach appears to be by Volgenant (1989) who described a branch and bound procedure, based on lagrangean relaxation and edge exchanges. Kawatra (1997) presents a similar approach based on lagrangean relaxation and a branch-exchange procedure, however the performance appears to be worse than that of Volgenant's algorithm. Craig, Krishnamoorthy, and Palaniswami (1996) develop several heuristics, including a neural network approach, for solving the DCMST.

Zhou and Gen (1997a, 1997b) present an approach to solve DCMSTs using a genetic algorithm. Their method uses the concept of Prüfer numbers, which is also used in this paper. However, the specifics of the algorithm are somewhat different. For example, they opted for a repairing strategy in their GA. Unfortunately their papers do not provide enough detail to completely duplicate their algorithm, nor are there sufficient computational results to get a good understanding of the performance of their algorithm.

Boldon, Deo, and Kumar (1996) develop four heuristics for solving the DCMST and implement them on a massively parallel SIMD machine. Deo and Kumar (1997) study 29 constrained spanning trees including the degree constrained spanning tree and obtain results using a massively parallel computer. Their algorithm is based on repeatedly solving Minimum Spanning Tree (MST) problems with increasing penalties for arcs involved in degree violations.

In this paper, we develop and discuss a few novel approaches, including a few heuristics as well as an exact enumeration approach for solving the DCMST. We also provide a relative comparison of the performance of these approaches and discuss the merits of each of them. We compare the performance of all of these approaches with the exact solution approach of Volgenant (1989).

In recent times, modern heuristic methods like simulated annealing, tabu search, genetic algorithms, neural network approaches, local search approaches and greedy random search approaches have been used extensively to solve complex optimisation problems. See, for example, Reeves (1993) for an elucidation of some of these approaches. A comprehensive bibliography of methods and applications of these heuristics also appears in Osman and Laporte (1996).

In Section 2, we develop a method for representing trees, which is used in some of our heuristic methods. A similar representation has been suggested by Gen, Zhou, and Takayama (1998). In Section 3.1, we describe an approach for solving the DCMST that is based on a genetic algorithm (GA). We also employ a problem space search (PSS), as described in Section 3.2, for solving the DCMST. In Section 3.3, we describe a simulated annealing (SA) approach that is different to the one presented for the DCMST by Craig, Krishnamoorthy, and Palaniswami (1996). In this paper, in addition to the GA, PSS and SA heuristics, we also develop a novel enumeration-based exact solution approach that is described in Section 4. In Section 5, we provide extensive computational analysis and discuss the performance of the algorithms that we developed. Finally we present some conclusions in Section 6.

2. Efficient representation of trees

In this paper, we develop a few heuristic algorithms for the DCMST. In all of these algorithms, it is necessary for us to manipulate trees on a continual basis. Commonly used heuristic approaches to optimisation problems like, for example, the travelling salesman problem (TSP) or the Quadratic Assignment Problem (QAP) work with a vector that represents a possible solution to the problem. This solution could be, for example, a permutation of the nodes of the underlying graph.

A large majority of heuristic approaches for problems like the TSP and QAP assume that one can move from one solution (vector) to another one in its *neighbourhood* by merely altering, swapping or moving elements in and out of the solution vector. For example, consider a node permutation sequence as a representation of a solution to the TSP. By randomly exchanging the positions of two nodes in this sequence, we are assured that the resulting vector still represents a possible solution to the TSP. Therefore, it is sufficient if we just evaluate the quality (or cost) of the representation.

However, such representations may not work well for optimisation problems on trees. The main criteria that are critical in designing efficient encodings (or representations) of trees for use in a neighbourhood search heuristic approach are: (1) The representation should lend itself to the neighbourhood structure and the neighbourhood operators. (2) The encoding should ensure that the result of these neighbourhood operators is also a tree, which is a connected *graph* without *cycles*. (3) The encoding should be efficient with respect to the computations that are required to traverse back and forth between an encoding and the tree that it represents. (4) The encoding should be unbiased, in the sense that it should not prefer any one region of the solution search space over another. Some efficient data structures for use in on-line updating of trees are provided in Frederickson (1983, 1985).

We now provide a description of some possible tree encodings and representations before describing the one we choose for this paper.

2.1. Arc list representation

Consider a set $V = \{1, \ldots, n-1\}$ as one possible representation of a tree in which the ith element of V is the arc index of the ith arc that is in the tree. Thus, $\bar{V}$ represents the set of $\{A \backslash V\}$ arcs of G that are not in the tree.

Consider a swap-based neighbourhood search heuristic in which an element of V is swapped with an element of $\bar{V}$. While efficient in terms of memory requirements and while it lends itself nicely to a swap-based neighbourhood search heuristic, such a procedure can not guarantee that the resulting set will continue to remain a tree, as this swap may easily have introduced a cycle.

2.2. Arc vector representation

Consider another simple representation in which a vector V of length $|A| = m$ represents a tree. An arc from A is said to be in the tree if it has a value of 1 in V. If it has a value of 0, it is not in the tree. Clearly, $n-1$ elements of V need to have a value of 1.

Now consider a search algorithm which produces a new solution in which k (where $k = 1, 2, \ldots$) elements of V which have the value of 0 are changed to have a value of 1, while k other elements which have the value of 1 now acquire a value of 0.

As in the earlier example, it is not assured that this new solution will be a feasible tree. Indeed, according to Palmer and Kershenbaum (1994), there is only a very small probability of $2^{-[n(\frac{n}{2}-\log 2(n))]}$ that a random representation of this type will be a tree.

2.3. Predecessor representation

It is also possible to designate a node r as the *root node* of the tree and then encode the tree by maintaining, for each node i, $p(i)$, which is the immediate predecessor in the path r to i. Consider a neighbourhood operator that applies a random alteration (say) to the value of $p(i)$ for a certain (randomly chosen) i. Again, it is not necessary that such a change will result in a feasible tree.

2.4. *Oriented tree representation*

In their paper which develops a tabu search procedure for the Capacitated Minimum Spanning Tree (CMST), Sharaiha et al. (1997), use a different representation of trees to the ones described above. They consider the CMST as a *oriented* tree rooted at a node r.

Associated with each node, i, are two predecessor pointers, $LSon(i)$ and $RBrth(i)$. These store, respectively, the left most son of and the right most brother of node i. A leaf node has an $LSon(\cdot)$ value of the negative of itself. Similarly, if a particular node is the last node in one level of this structured tree, its $RBroth(\cdot)$ points to the negative of its parent (predecessor) in the rooted tree. Such a representation works efficiently for the CMST, where, along with the requirement that all intermediate solutions must be trees, it is also necessary for all subtrees to honour a capacity constraint. Thus, the tree encoding must lend itself to easy traversal and on-line updating.

However, even if we use this representation for the DCMST, the traversal operations are computationally expensive. Furthermore, it is necessary to carry out such traversal operations to check the degree constraints and the tree connectivity constraints. Such traversal operations are also necessary to evaluate the cost of the tree. Moreover, unlike the CMST (for which this representation was used), the degree constraints in the DCMST are easy to maintain and do not require complex traversal operations and dynamic on-line updating. We note that this tree representation was used by Volgenant (1989) in his algorithm for the DCMST.

2.5. *Cayley's representation*

One efficient method for representing, encoding and decoding trees that lends itself *extremely well* to neighbourhood-search-type heuristic algorithms is based on *Cayley's Theorem*. The resulting encoding of the tree is called its *Prufer Number*. See Moon (1967) for a full description of the properties and the Cayley's method of representing trees. Here, we summarise the scheme and provide examples of coding and decoding a tree using this method.

Given a tree, T, its Prufer number, P, is an encoding with $n - 2$ node indices. This encoding of T is constructed using the following algorithm:

Algorithm 1: Encode

```
Step-1: $P$ is initially empty. All $n-1$ arcs in the tree $T$ are
    labelled as temporary.
Step-2: Using *only* the temporarily labelled arcs of $T$, construct a
    set $D_1 \subset N$ whose elements are nodes $i \in N$ that have degree,
    $d_i = 1$. In other words, the elements of $D_1$ are leaf nodes of
    the temporarily labelled arcs of $T$.
```

```
  Step-3: Choose node $k$, such that $k$ is the least index in $D_1$.
  Step-4: Consider arc $(j, k) \in T$. Add the value $j$ as the next element
       in $P$.
  Step-5: Give arc $(j, k) \in T$ a permanent label.
  Step-6: If there are only 2 remaining nodes with a temporarylable,
       Stop. Else, go to Step-2.
end algorithm 1.
```

We use the diagram presented in figure 1 to describe how this encoding works.

The tree in figure 1 can be represented as $P = [9817817]$. This encoding is obtained as follows. Initially, all arcs in T are temporarily labelled and we construct the set $D_1 = \{4, 6, 3, 5, 2\}$. Since node 2 is the lowest numbered node index in D_1 with degree 1 and since node 2 is connected to node 9 in T, we first add the node index 9 to P. We label arc $(2, 9)$ as permanent and update $D_1 = \{4, 6, 3, 5, 9\}$. Since node 3 is now the lowest numbered index with degree 1 in D_1 and since node 3 is connected to node 8 in T, we add node 8 as the next digit of P. We permanently label arc $(3, 8)$, update $D_1 = \{4, 6, 5, 9\}$. We proceed in this manner to encode the tree.

Given a Prufer number P, we can decode it using the following algorithm.

Algorithm 2: Decode

```
  Step-1: Define $\bar{P}$ as the temporary Prufer number. Initialise
      $\bar{P} = P$. Let $S$ represent the set of nodes that have been
      already considered. At the start, $S$ contains no elements.
  Step-2: Let $k$ be the smallest node index that is *not* present in
      $\bar{P} \cup S$.
  Step-3: Let $j$ be the first element of $\bar{P}$. Create an arc $(k, j)$ in
       the tree that is being constructed. Add $k$ to the set $S$.
```

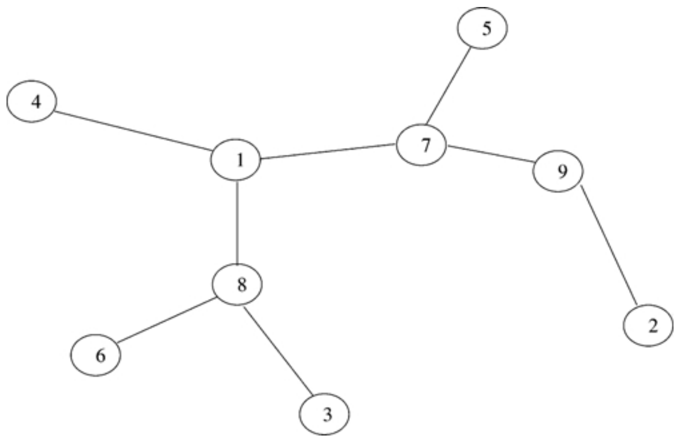

Figure 1. An example of an encoding.

```
  Step-4: Remove the digit $j$ from $\bar{P}$.
  Step-5: If $\bar{P}$ contains more digits, go to Step-2. If $\bar{P}$ no longer
      contains any digits, connect $j$ to the smallest node index in
      $S \cup j$ that is not already present in the tree that is being
      constructed.
end algorithm 2.
```

As an example, consider the decoding of $P = [2463215]$. To commence, we initialise $\bar{P} = P$ and S as null. Initially, node 7 is the smallest element that is *not* in $\bar{P} \cup S$. So, we add $(7, 2)$ to T, update $\bar{P} = [463215]$ and $S = \{7\}$. Now the smallest index that is not present in $\bar{P} \cup S$ is node 8. We add $(8, 4)$ to T and $S = \{7, 8\}$. In this manner, we continue to decode P till, in the end, we have $\bar{P} = [5]$ and $S = \{7, 8, 4, 6, 3, 2\}$. Node 1 is the smallest index that is not present in $\bar{P} \cup S$. We add $(1, 5)$ to T and now, $S = \{7, 8, 4, 6, 3, 2, 1\}$. Since $\bar{P}$ is empty, we add arc $(9, 5)$ to T, since $j = 5$ and since node 9 is the least index that is not present in S. Thus, we obtain the tree that is presented in figure 2.

This encoding has the following suitable properties: The degree of each node in T can be directly evaluated as one more than the number of times it appears in P. Given *any* random permutation P of $n - 2$ digits, we are *always* assured that it can be decoded into a feasible tree T. With the help of appropriate data structures, the encoding and decoding of P can be carried out efficiently, requiring $\mathcal{O}(n \log n)$ operations. Moreover each tree has a unique Prufer number and hence a unique encoding. We use this property (along with a hashing function) to identify and eliminate duplicates in our genetic algorithm implementation.

We can, therefore, immediately think of constructing neighbourhood search algorithms in which the elements or P are randomly altered or swapped to produce new trees. These new trees may or may not be feasible with respect to the degree constraints. Alternatively, we can think of a genetic algorithm that uses P as a chromosome representation. Two random

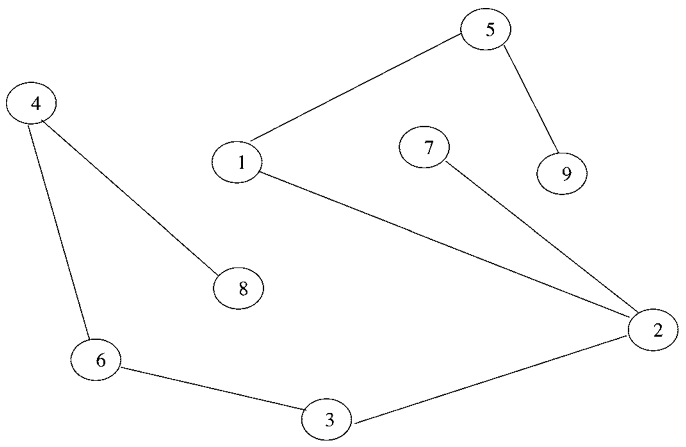

Figure 2. An example of a decoding.

parents P_1 and P_2 can be crossed to immediately produce two (feasible) children trees that may or may not be feasible with respect to the degree constraints.

Consider a move-operator that changes an index in P_1 to yield a new representation P_2. Although the resulting *representation*, P_2 may be considered to be in the neighbourhood of P_1, the resulting *tree*, $T(P_2)$ is often not in the immediate neighbourhood of the tree $T(P_1)$. For example, consider a graph with 7 nodes and a change from $P_1 = [37261]$ and $P_2 = [37262]$. Although the representations are in the neighbourhoods of each other (since only one digit is changed), the resulting trees have *only* two arcs in common. We alleviate this problem in our GA implementation by adopting a special crossover mechanism called *path crossover*. By using this crossover method, it is still possible to obtain child trees that are not in the immediate neighbourhood of the parent(s). However, the likelihood of obtaining trees in the neighbourhood of the parent(s) is greatly increased with the use of this method, rather than a random crossover method. We note that the path crossover scheme has a strong similarity to other forms of recombination in the *Scatter Search* methods proposed by Glover (1994). We do, however, define and use a new representation (based on the above representation) for use in our SA algorithm and describe this in Section 3.3.

3. Heuristic algorithms for the DCMST

3.1. Genetic algorithm

Genetic algorithms (GAs) have been effective and have been employed for solving a variety of difficult optimisation problems. Much of the basic ground work in implementing and adapting GAs has been laid by Holland (1975). Since then, a large number of papers have appeared in the literature, proposing variations to the basic algorithm or describing different applications. A discussion of some of these as well as definitions and some of the basic GA terminology that is used in this section can be found in Davis (1991), Goldberg (1989) and Osman and Laporte (1996). A large number of successful applications of genetic algorithms have been presented in Osman and Laporte (1996).

One of the main issues that needs to be addressed when trying to implement a GA for a problem is the definition of an encoding of solutions. This encoding, or solution representation, is also referred to as the *chromosome*. The encoding must be constructed in such a way that, when a crossover is performed, the resulting chromosome again represents a solution of the problem. Fortunately, as described in Section 2.5, Cayley's representation of trees makes it easy to preserve the spanning tree structure even when crossovers are performed. Hence, dealing with the degree constraint is the only problem that we could potentially face in applying a standard GA for solving the DCMST.

In our implementation, we use, P, the Prufer number of a tree (as described in Section 2.5) as our representation of a *chromosome* in the GA. Using this problem representation, we then tried several variants of the standard GA application, including:

Infeasibility: By using the Cayley's method to represent our trees, we ensure that there are no structural infeasibilities in our GA. In other words, all representations that result from crossovers or mutations in the GA are always trees. However, these could be infeasible with respect to the degree constraints. One way to deal with the degree constraint is to ensure that, in each generation, we only retain individuals that are feasible with respect to node degrees. It is easy to generate an initial population of feasible solutions. Under such a regime, each new offspring can be checked for feasibility. In fact, as has been described in Section 2.5, the degree constraint feasibility check is embedded in the Cayley's encoding and can, hence, be completed without decoding the new chromosomes. Infeasible chromosomes may be rejected without even being evaluated for their fitness. We refer to this variant of the GA algorithm as GA-F.

Alternatively the degree constraint may be relaxed. In other words, in our GA implementation, it is possible for us to accept child solutions that do not satisfy the degree constraints on the nodes. However, we need to ensure that a higher preference is given to solutions that are feasible with respect to the degree constraints. We ensure this feature by adding the degree violations to the objective via a penalty multiplier. Hence the objective function is:

$$\sum_{l \in T} c_l + p \sum_{i \in N} \max\{0, d_i - b_i\},$$

where, T is the set of edges in the tree, b_i is the maximum allowable degree of i and p is a penalty parameter. In our implementation, the penalty parameter was dynamically updated as follows: We commence with $p = 1$. Whenever there has been no improvement in the number of feasible solutions in the population for 5 generations (or 95% of the population is still infeasible), we set $p \leftarrow 1.10\,p$. Similarly whenever there has been an increase in the number of feasible solutions (or 95% of the population is feasible), we set $p \leftarrow \frac{1}{3}p$. This dynamic, penalty-alteration method was employed by Gendreau, Hertz, and Laporte (1994) in a tabu search procedure for solving the vehicle routing problem. We refer to this variant of the GA algorithm as GA-P.

Crossover methods: We use a standard single-point crossover method in GA-F, where two children are produced, which between them, carry all of the genes from the two parents. GA-P on the other hand uses a path-crossover scheme as proposed by Ahuja, Orlin and Tiwari (1995). For a set of parents, P_1 and P_2, we consider every possible single-point crossover, thus creating $2\,(n-2)$ offspring (since there are $n-1$ genes). The best offspring with more genes from P_1 than P_2 replaces P_1 if it is fitter than P_1. Similarly, the best offspring with more genes from P_2 than P_1 replaces P_2 if it is fitter than P_2. In other words, a 'path' of descendants is created by starting with P_1 and replacing each gene in turn by the gene in the corresponding position from P_2. The two best offspring on the paths from P_1 to P_2 and from P_2 to P_1 replace their closest parent, provided they are fitter. For each of these two crossover types, a multi-point crossover variant was also tried. However computational experiments suggest that these provide no significant advantage over the simpler single point crossover. Hence no results are provided for the multi-point crossover variants.

Breeding: For GA-F, all of the individuals in the population are paired in a random manner and produce two offspring each. The best of these offspring then replaces the weakest of the parents (if the offspring is fitter). In GA-P on the other hand, the path-crossover produces a relatively large number of offspring that have to be evaluated, hence only one pair of randomly chosen parents are allowed to breed in each generation. The population size of GA-F is adjusted to ensure that roughly the same number of new chromosomes are evaluated in each generation. In both methods however the fittest individual in the population is crossed with each of the other individuals once in every 10 generations.

Termination: GA-F terminates after $50n/\bar{b}$ iterations, where $\bar{b}$ is the average maximum degree over all nodes. The reasoning is that larger problems require more iterations, on the other hand having a larger degree constraint makes the problem significantly easier. GA-P terminates when there has been no improvement in the best solution found for a 100 iterations.

Mutation: Mutation occurs in both versions of the GA with a probability of 0.05. Mutation involves a randomly chosen gene of a child chromosome being changed to a new randomly chosen value.

Initial Population: A half of the initial population is created purely at random by simply picking the value of each gene from a uniform distribution (but disallowing degree violations for GA-F). The other half of the initial population is generated using Algorithm 3 (description provided below) with randomisation rank $r = 3$. The only difference between GA-F and GA-P being that the latter allows the degree constraint to be violated in Algorithm 3.

Population Size: The population size of GA-P is always set at 100. For GA-F the population is based on the number of nodes in such a way that the number of offspring generated in each generation roughly matches those generated using the path-crossover scheme of GA-P.

Algorithm 3: Randomised Minimum Spanning Tree

```
    Sort arcs in complete graph by cost
    Let $r \geq 1$ be the randomisation rank
    Pick a random initial node $i$
    Set $T = \{i\}$
    while $T \neq N$ do
    Let $\mathcal{C} = \{(t,i) \mid t \in T;\ \ i \notin T;\ \ d_i, d_t < d\}$
    Choose one of the $r$ cheapest arcs $(t,i)$ in $\mathcal{C}$ at random
    Set $T \leftarrow T \cup \{i\}$
  end while
end Algorithm 3.
```

3.2. Problem space search

Problem Space Search (PSS) is meta-heuristic which combines a simple constructive heuristic with a genetic algorithm (or possibly some other heuristic search algorithm). This type of heuristic has been successfully used in scheduling problems as well as other combinatorial problems (Storer, Flanders, and Wu, 1996; Storer, Wu, and Vaccari, 1992).

The main idea of PSS is as follows: instead of searching through the space of feasible solutions directly, a base heuristic is used as a mapping function between a space of perturbations and the solution space. Thus it is sufficient to use the base heuristic and some method of searching through the perturbation space, to find a good solution to the original problem. Although proposed initially as *Problem Space Search*, this method could also be called an "Instance Space Search" since perturbations are performed in the space of problem instances.

In implementing PSS, we need to chose a subset of the full set of input data to perturb. Often it is advantageous not to perturb all of the input data. For example, for the DCMST it is probably advisable not to perturb b_i, the input parameter which represents the maximum degree for each node. For PSS to be effective, it requires the base heuristic to be computationally efficient. Furthermore the base heuristic must allow any solution to be generated provided a suitable perturbation is used. For the DCMST, we apply the perturbations to the arc costs.

In the PSS that we develop in this paper, for the DCMST, we use Algorithm 3 as our base heuristic but without randomisation. In other words, we set $r = 1$. Different perturbations to the arc costs lead to a different sorting of the edges and hence, different solutions (although there may be many perturbation vectors that give rise to the same solution). The base heuristic here has a computational complexity of $\mathcal{O}(n \log n)$. Hence it can be applied repeatedly to different perturbations of the input data. We then use a GA to operate in the perturbation space. Given two perturbation vectors (node weights, in our case), we use a simple one-point crossover GA to obtain a new perturbation vector.

There are two different ways that such a perturbation vector can be used in our PSS implementation. The most obvious one would be to have a perturbation vector of length $\frac{n}{2}(n-1)$, one element for each undirected arc in the complete graph, $G = (N, A)$. After implementing and testing this idea, we abandoned it for a slightly different method. This is because the perturbation space was found to be unnecessarily large, making it difficult to find good solutions.

We use a perturbation vector π of length n. Instead of operating directly on the arcs, our PSS perturbation vector represents a weight on each *node*. Then c'_{ij}, the updated cost of an arc (i, j) is given by, $c'_{ij} = c_{ij} + \pi_i + \pi_j$. By adopting such a perturbation vector, we only need to search a much smaller parameter space. Moreover, since the base heuristic does not effectively take care of the degree constraint on the nodes, we can use the perturbation vector to, effectively, encourage or discourage the use of each node, depending on their under-utilisation or attractiveness, respectively.

An alternate way to describe this PSS algorithm is in terms of lagrange multipliers. In effect, we are searching the space of lagrangean multipliers corresponding to the degree constraints on the nodes. Note that, while the optimal set of lagrange multipliers would give the optimal solution in the base heuristic, the algorithm still produces feasible solutions even if the lagrange multipliers are far away from optimality.

In our implementation, we use a population of size 75. In other words, we always consider a population of 75 perturbation vectors. Experimentation with different population sizes indicates that no significant gains can be made by changing this parameter. Each gene in the original population as well as each mutated gene is taken from a normal distribution with a mean of zero and a standard deviation of 0.15 $(max - avg)$ where *max* is the maximum arc cost and *avg* is the average arc cost.

We employ a GA to operate in the perturbation space. In each generation of the GA, two chromosomes (perturbation vectors, each of length n) are selected and a single point crossover is performed. Each gene in the new chromosome then has a probability of 0.01% of being mutated. The GA terminates when no improvements have been made in 150 generations.

3.3. Simulated annealing

Simulated Annealing is another popular search heuristic often applied to combinatorial problems. The basic concepts of this method and some of the common variations are described in Collins, Eglese, and Golden (1988) and Rutenbar (1989). A large number of successful applications of simulated annealing have been presented in Osman and Laporte (1996).

In this paper we chose to use a *cut-and-paste* neighbourhood transformation. More precisely, a new spanning tree is formed by randomly picking a node i and deleting the arc from node i to its parent node, j. The deletion of $(i, j) \in T$ creates two disconnected components. The two components are then re-connected by randomly choosing another node to connect to i.

In order to implement this cut-and-paste operation efficiently, we need to store more than just the Prufer number of the tree, in order to avoid having to decode a number in each iteration. In addition to the Prufer number, P, that is associated with a tree, T, we also store B, a permutation of the nodes.

We use both P and B in the representation of solutions in our SA implementation. We describe the use of these through examples.

Consider again the tree shown in figure 1. The Prufer number of this tree is $P = [9817817]$. There are several possible values of B that could be used in conjunction with this representation. Consider two such representations of B as $B = [234568179]$ and $B' = [264938571]$. The first of these representations arises naturally out of Step-3 in Algorithm 1. Applying Algorithm 2 again, but with the node ordering as given in B (or B') again produces the original tree. In each case the arcs that are created in Step-3 of Algorithm 2 are given by the corresponding elements of B (or B') and P. Considering each of these arcs as directed and pointing towards the element of P, one can see that the arcs all point to the root node where for B the root node is node 9 and for B' node 1.

There are three different ways to interpret how the elements that are stored in P and B define the corresponding tree, T:

1. P is the Prufer number with respect to permutation B. In other words when decoding the Prufer number to obtain the tree, the operation 'smallest node index' (in Step-2 of Algorithm 2) finds the first suitable index in B.
2. The set of pairs $\{\{P_i, B_i\} \mid i = 1, \ldots, n-2\} + \{B_{n-1}, B_n\}$ give the edges of the tree. In other words $B_1, \ldots, B_{n-2}$ is the decoded string obtained at the end of Algorithm 2 when decoding P with respect to the ordering in B. This view of the representation makes it trivial to calculate the cost of a solution.

3. For each B_i ($i \leq n-2$), the corresponding P_i is the parent node in a rooted representation of the tree. Here B_{n-1} has parent node B_n which is the root of the tree. Furthermore all of the descendant of a node i (*ie* all of the nodes of the subtree rooted at i), are to be found to the left of i in B. Using this view of the representation of the tree allows the cut set to be identified quickly when an arc is removed from the tree.

To find a neighbourhood move we simply pick a number $j \in \{1, \ldots, n-2\}$ at random. The arc (B_j, P_j) is to be removed. In order to find a replacement arc we first label the subtree rooted at B_j. This can be done in a single pass of $B_j, \ldots, B_1$. An unlabelled node k is then picked at random as the new parent of B_j. The change in cost is simply the difference in cost between the new and the old arc. If a new solution is accepted, the update operation requires two passes through B. In the first pass, all of the labelled nodes in B are added to the new permutation B' (and the corresponding elements of P to P'). Next the arc (B_j, k) is added to B' and P' respectively, and finally all the unlabelled elements of B are added to B'.

Below, we give an example of how the neighbourhood move operator works. Consider the tree in figure 1. It is represented by $P = [9817817]$ and $B = [234568179]$. Let $j = 6$. Hence the arc (8, 1) is to be removed. The subgraph consisting of nodes {3, 6, 8} is labelled as disconnected. We need an arc from 8 to the unlabelled part of the graph. Say (8,2) is chosen at random. The new representation is $B' = [368245179]$ with $P' = [8829177]$.

Using this representation, the basic operations of finding a possible neighbourhood move as well as the update to create the neighbouring solution can be performed in $\mathcal{O}(n)$ operations. The evaluation of a move requires $\mathcal{O}(1)$ operations.

It is possible for any SA to operate either in the feasible search space or in a search space that also allows infeasible solutions. We conducted some preliminary tests of an implementation in which only feasible solutions are considered in the SA's search of the solution space. These initial tests enabled us to conclude that an SA that admits infeasible solutions is more effective. This is not entirely surprising because, for the SA method to be effective, the neighbourhood moves must allow the search space to be traversed relatively easily. In an SA implementation that operates only in feasible space, the degree constraints often restrict the number of choices for re-connecting the tree after the removal of an edge.

Thus, in our implementation of the SA, it is possible for intermediate solutions to violate the degree constraint. However, we impose a penalty of $\sum_{i \in N} \rho \max\{0, d_i - b_i\}$, for violating the degree constraints. The initial value of the degree penalty parameter is set as a multiple of the maximum arc cost. During the course of the SA algorithm, the penalty is set to $\rho \leftarrow 1.10\rho$ if $\frac{n}{2}$ consecutive iterations have produced infeasible solutions. Conversely, if $\frac{n}{2}$ consecutive iterations produce only feasible solutions, then the penalty is reduced to $\rho \leftarrow \frac{\rho}{1.10}$. This dynamic, penalty-alteration method, which is similar to our implementation in GA-P, was employed by Gendreau, Hertz and Laporte (1994) in a tabu search procedure for solving the vehicle routing problem.

We use a fairly standard cooling regime in our SA implementation. The initial temperature is set at 1% of the standard deviation in cost over a random walk of length $20n$ in which all neighbouring solutions are accepted. The Markov chain length is set at $100n$. This choice of Markov chain length reflects the fact that the number of neighbourhood moves that can be made increases for a particular solution increases linearly with n. The temperature is reduced by 2% after each iteration until either 3 successive Markov chains produce the same result (indicating that the SA is stuck in a local minimum) or when 300 temperature decrements have been performed.

4. Exact algorithms for the DCMST

In this section, we provide a brief description of the exact approach of Volgenant (1989) for solving the DCMST. We also describe a simpler exact solution method based on branch and bound.

4.1. Lagrangean relaxation

A lagrangean relaxation based approach to DCMST has been described by Volgenant (1989). The algorithm relaxes the degree constraint and uses a simple ascent method for obtaining lower bounds. The author has kindly made his implementation available to us so that we could use it for accurate comparisons in this paper. In our computational results tables, we refer to this algorithm as Volg.

4.2. Branch and bound

This algorithm is essentially a very simple depth first, branch and bound algorithm that uses the unconstrained minimum spanning tree as a lower bound in each iteration. For branching, the node i with the greatest degree violation $(d_i - b_i)$ is selected. A new branch of the enumeration search is then created for every possible choice of b_i of these arcs, in which all the arcs that are not chosen are deleted. For each branch the new minimum spanning tree is computed.

The basic algorithm, which we refer to (in the results section) as 'Branch & Bound', has been refined through the following features:

1. The sorted list of remaining edges is stored for each node of the branch and bound list. This speeds up the identification of a new minimum spanning tree after some edges have been deleted or eliminated. It also provides a convenient way of keeping track of all arcs that still need to be considered in the current part of the branch and bound tree.

   While this could require excessive amounts of memory particularly for large problems, in practice we encountered no difficulties, at least in part due to point 2, discussed below.
2. Expensive edges are eliminated from consideration based on the lower and upper bounds. Elimination is done by calculating the cost of replacing the most expensive edge in the lower bound solution with a given edge (i, j)— irrespective of whether or not the addition of the resulting arc would even yield a tree. If this cost exceeds the upper bound, the edge can be eliminated from further consideration.

   Empirically this method tends to eliminate about 80% of all edges at the root node for problems with $b_i \geq 3$.
3. The branch and bound algorithm usually generates several descendants at each node. These are searched in order of least lowest bound to greatest (but still using a depth first

methodology). Using this search order improves the chances of finding a better upper bound early in the search.

4. The lower bound can be tightened slightly by considering all of the nodes that violate the degree constraint. For each node i with $d_i > b_i$ the cost increase δ_{ij} is calculated for replacing each arc (i, j) in the MST with the next most expensive arc in the arc list. The $d_i - b_i$ smallest such increases can be added to the lower bound.
5. Consider a node i of the spanning tree which has to be branched on (i.e., $d_i > b_i$). The normal branching process creates $\binom{d_i}{b_i}$ descendants in each of which $d_i - b_i$ of the arcs that are adjacent to node i have been deleted. However there may also be arcs left in the arc list that join i to other nodes. In this case we also delete an unused arc (i, k) if the adjacent node of i which is closest to k is being deleted. In other words we not only prohibit a node j adjacent to i from connecting to i but also all nodes in the same 'cluster' as j.

   For example, consider a problem with 11 nodes in which the root node solution (the unconstrained MST), consists of arcs $(1, i)$ and $(i, i + 5)$ for all $i = 2, \ldots, 6$. In other words the tree looks like a star with 5 chains of length 2 radiating out from node 1. Let $b_1 = 3$. In this case one of the branches of the branch & bound tree would be formed by deleting arcs (1, 5) and (1, 6), in addition arcs (1, 10) and (1, 11) would be disallowed from occurring subsequently in the branch and bound tree.
6. For problems in which many instances with $d_i \gg b_i$ occurs, it is possible to generate a large number of descendants in each node of the branch and bound tree. However it is often necessary to traverse to a very deep level in the tree search to find good upper bounds relatively quickly. Hence we artificially limit the number of descendants generated for each node of the branch and bound tree to 100. The program used to generate the results in this paper does this by keeping β_i out of the d_i arcs where β_i is chosen maximally such that $\binom{d_i}{\beta_i} < 100$. A more sophisticated alternative (which was not implemented) would be to heuristically group the d_i adjacent nodes into δ_i clusters such that $\binom{\delta_i}{b_i} < 100$, and only keeping arcs to the chosen clusters for each descendant of the branch and bound tree.

5. Computational results

In this section, we discuss the computational performance of all the algorithms that have been described in this paper.

5.1. Data sets

Several data sets were used to test the algorithms described above. Following the convention in earlier papers in the literature, all coordinates and distances are set to integer units:

CRD These are 2 dimensional euclidean problems. Points are generated randomly with a uniform distribution in a square. This type of problem was used by Narula and Ho (1980), Savelsbergh and Volgenant (1985), and by Volgenant (1989). These problems tend to be very easy as there are never any nodes with a degree of more than 4 in the unconstrained MST.

SYM These higher dimensional euclidean problems are analogous to the CRD problems but with points generated in a higher dimensional euclidean space. This makes the problems slightly harder as there is a higher probability of nodes having a degree violation and a node may have a degree of up to 2 $\times$ dimension. These data sets also come from Volgenant (1989).

STR In an effort to produce more difficult data sets to push the limits of the algorithms, we generated structured sets of points. For these types of problems points are distributed in a higher dimensional space in clusters. One node is located at the origin, all others are split into clusters of about the same number of nodes. One cluster is located 500 units along each of the major axis of the euclidean space, within the cluster nodes are distributed in a hypercube of size 50 units. The number of dimensions D was varied between 3 and 7.

SHRD Since even the STR problems are still quite easy to solve optimally for a degree of 3 or more, we set out to construct an artificial example that is particularly hard. A difficult problem of any size can be constructed by using non-euclidean distances as follows. The first node is connected to all other nodes by an edge of length ℓ, the second node is connected to all nodes bar the first by an edge of length 2ℓ and so on. We randomised this slightly by adding a uniformly distributed perturbation between 1 and 18 where $\ell = 20$. This reduces the likelihood of a large number of optimal solutions existing but doesn't change the underlying complexity of the problem.

All of the above data sets are available from the authors on request. These can also be obtained from OR-Library (Beasley, 1990). Note that, in *all* of the problems in all of the four data sets mentioned above, we set $b_i = b$, for all $i \in N$. The respective values of b are presented in the tables below.

5.2. Computational comparisons

All of the algorithms that we describe in this paper were coded in C (except the lagrangean relaxation which was converted from Pascal to C using an automatic translator). The results that we present are obtained by running these algorithms on a 200 MHz DEC alpha processor.

In Table 1, we present the results for the CRD and the SYM problems. There are 10 CRD problems for each $n = 30, 50, 70, 100$, thus giving 40 CRD problems in all. There are also 10 SYM problems for each $n = 30, 50, 70$, thus giving 30 SYM problems in all. In Table 1, we do not provide computational results for *all* of the 70 CRD and SYM problems. Instead, we provide average results for each problem type. For example, the first row of this table represents average computational behaviour over all the CRD problems with $n = 30$. Table 1 shows the average behaviour of GA-F, GA-P, PSS, SA, Branch & Bound and Volg on all of the CRD and SYM problems. It is not necessary to consider a degree constraint of value greater than $b = 3$ for the CRD problems. For all of these problems, the MST is optimal with respect to degree constraints if we consider $b > 3$. For the SYM problems, however, it is also interesting to consider $b = 3, 4$ in all cases, and for $n = 50, 70$ instances where $d_i > 5$ occur in the unconstrained MST.

It is clear that, for the CRD and SYM problems, Volg, the exact algorithm of Volgenant (1989), performs exceedingly well. It requires, on average, 0.45 seconds per problem.

Table 1. Comparison of easy problems.

			GA-F		GA-P		PSS		SA		Branch & Bound			Volg
Problem	n	b	Gap	CPU	Gap	CPU	Gap	CPU	GAP	CPU	Nodes	Gap	CPU	CPU
crd	30	3	0.04	4.18	0.50	4.60	0.04	5.80	0.04	4.40	3	0.70	0.00	0.00
crd	50	3	0.07	23.84	0.07	14.22	0.04	18.74	0.00	22.71	2	0.01	0.01	0.00
crd	70	3	0.06	73.03	1.20	34.09	0.00	40.06	0.03	72.75	2	0.01	0.02	0.01
crd	100	3	0.07	248.13	0.03	83.60	0.00	87.33	0.11	196.92	3	0.04	0.04	0.02
sym	30	3	0.77	5.13	3.67	5.41	0.04	9.34	0.42	9.87	84	4.26	0.02	0.01
sym	30	4	0.08	4.15	0.74	5.74	0.00	9.53	0.01	8.75	3	0.16	0.01	0.01
sym	50	3	2.26	28.27	3.95	18.19	0.30	45.01	0.74	45.27	591	4.78	0.12	0.28
sym	50	4	0.79	23.19	1.20	18.27	0.24	48.98	0.22	44.40	19	0.95	0.02	0.01
sym	50	5	0.31	19.16	0.11	20.24	0.05	50.44	0.00	37.73	4	0.10	0.01	0.01
sym	70	3	4.35	88.75	6.67	42.01	2.53	99.34	1.00	117.24	25013	5.60	5.71	$*5.03$
sym	70	4	0.83	72.24	1.82	41.85	1.23	119.38	0.48	113.44	208	1.03	0.07	0.10
sym	70	5	0.14	60.99	0.65	41.77	0.83	116.55	0.15	102.16	4	0.11	0.03	0.01

However there is one exception to this: Volg does not solve SYM700 (with $b = 3$) to optimality within 10 minutes of CPU time. Hence, the asterisk against this entry in Table 1. It is also interesting to note that our simple enumeration-based Branch & Bound algorithm performs reasonably well. We present three sets of results for the Branch & Bound algorithm. The first reflects the number of tree nodes that are explored. The second presents the initial lower Gap, which is the ratio of the lower bound and the optimal solution, expressed as a percentage. The last column provides the CPU time in seconds. The Branch & Bound algorithm uses the upper bound from the PSS heuristic as a starting point, however the running time for that heuristic is not included in the CPU time. We observe from the last column for the Branch & Bound algorithm that the average CPU time consumed by this algorithm, over all problems is 0.50 seconds. If we exclude the CPU time for the problem SYM700 (the problem which Volg did not complete), then the average CPU time for the Branch & Bound drops to just 0.19 seconds. This compares very favourably with the performance of Volg.

Looking at the average behaviour over the CRD and SYM data sets, none of the heuristics that we develop are comparable to the excellent performance of the exact approaches, in terms of both CPU time as well Gap. Here, *Gap* is defined in the normal manner, as the ratio of the heuristic solution and the optimal solution, expressed as a percentage.

We can, however roughly compare the performance of the GAs that we have developed in this paper with previous approaches in the literature. A rough comparison can be made using the results in the above table with the GA results presented in Zhou and Gen (1997a) and again in Zhou and Gen (1997b). Zhou and Gen test their GA using a data set with only 9 nodes introduced by Savelsbergh and Volgenant (1985). Their results indicate an average Gap of 0.7% which is worse than either of the GA implementation presented in Table 1, even on the much larger CRD problems with 30 nodes and more.

Notwithstanding the overall weak performance of the heuristics on the CRD and SYM data sets, we can conclude that both PSS and SA perform reasonably well in terms of providing tight upper Gaps. Over all problems, PSS provides an average Gap of 0.44%, while SA provides an average upper Gap of 0.27%, while the GA implementations provide average Gaps of 0.81% and 1.71% respectively. Despite the slightly larger computational effort required by SA and PSS (when compared to the GA implementations) we can conclude that the former two heuristics are marginally more suited for solving the CRD and SYM problems.

It appears that the main reason for the superior performance of the exact methods is that for these data sets the unconstrained minimum spanning tree violates very few of the degree constraints. Hence, a method that systematically eliminates these violations will generally do very well. On the other hand the GA approaches in particular (even the GA-P approach that can start with the unconstrained minimum spanning tree in its initial population) have a neighbourhood structure that makes it difficult to make these simple repairs in order to obtain the optimal DCMST.

For the sake of comparison with Deo and Kumar (1997) we also tested the algorithms on a few TSP data sets (which are in the same category as the CRD data sets). These were obtained from TSPLIB (http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html). Table 2 presents the results. The CPU time required to obtain exact solutions (using either Volg or our Branch & Bound method) was less than 10 seconds in each case. We note that for the same problems, (Deo and Kumar, 1997) report CPU times of between 0.1 and 0.5 seconds. However, it is not meaningful to make comparisons of CPU times on these problems with those reported in Deo and Kumar (1997) since the algorithms were executed on very different machines; (Deo and Kumar, 1997) uses a massively computer with 8192 processors!

For pr264, a 264 node problem, the minimum spanning tree lower bound is almost the same as the optimal degree 3 DCMST. All of the heuristics except for GA-P and SA (when

Table 2. Results for selected TSP data sets.

Algorithm	pr264	att532	rat575
Best Known TSP soln	49135	27686[a]	6773
MST Lower Bound	41142	75872	6246/6248
DET (Deo and Kumar, 1997)	41143	NA	6265
RAND (Deo and Kumar, 1997)	41143	NA	6265
GA-F	41143	75981*	6250
GA-P	44344	75981	6397
PSS	41143	75981	6250
SA	43438	79046	6393
Branch & Bound	41143	75912	6250
Volg	41143	75912	6250

[a]The att532 TSP result uses integer pseudo-Euclidean distances, so edge distances differ approximately by a factor of $\sqrt{10}$.

used with default parameters) obtained the optimal solution. For att532, which was not tested by Deo and Kumar (1997), the results are similar, except that GA-F did not finish running within the alloted CPU time. None of the heuristics converged to the optimal solution but the Gaps (apart from SA) are just 0.09%. For rat575, a 575 node problem, we obtain a minimum spanning tree solution which differs marginally from that reported in Deo and Kumar (1997). We suspect that this is due to rounding errors when calculating integer distances between points with integer coordinates. In Table 2 we report both of these lower bounds. These results are indicative of the ability of our heuristic algorithms to solve even these larger problems.

However, there still exists the issue that it is hard to justify the use of these heuristics when it is possible to obtain exact solutions by expending a *fraction* of the computational effort! In fact if we do not start with any heuristic upper bound in the Branch & Bound method for the most difficult of the SYM problems (SYM700, $b = 3$) the computational time increased by only 1.09 seconds (2.8%).

However, based on our extensive computational experiments with the CRD and SYM data sets, we conclude that these are essentially very easy problems. A lot of them are solved to optimality even by Algorithm 3, which is a simple randomised Dijkstra's algorithm for the MST. Thus, in spite of the weak performance of our heuristics on the CRD and SYM problems, we proceeded to develop these as effectively as possible to test their performance on some harder DCMST data sets that we developed. We place two new DCMST data sets in the literature.

The first of these new and possibly hard data sets is the STR data set, which is topologically structured to ensure that they will not be easy. The results of our experiments with this data set are presented in Table 3.

In Table 3, the first column reflects the size of the problem, the second reflects the uniform degree constraint on all nodes and the third reflects D, the *dimension* that is chosen in developing the data set. The number of dimensions was varied from 3 to 7. We do not present results for $b = 3$ for problems of size $n = 30, 50, 70$ as these are not particularly interesting or instructive.

The first observation that we make is that there are no results for Volg in Table 3. This is because this algorithm is *not* able to solve *any of these problems* in a stable manner. Volg is either unable to solve these problems to optimality in a reasonable amount of computational time, or produces infeasible solutions (due to a bug which we were unable to resolve). However we note that in one instance with $n = 50, b = 4$ and $D = 5$, where Volg does produce the correct solution, it required 68.57 seconds of CPU time, compared with just 2.03 seconds for the Branch & Bound method (even without using the PSS upper bound).

However, our Branch & Bound algorithm is able to solve *most* of the problems in this set to optimality in a reasonable amount of computational time. Even this approach is not able to cope with the most difficult problems in this data set, where $D = 7$. We terminated the algorithm when it was unable to obtain optimal solutions within a fixed amount of CPU time (10 min). Hence the asterisks against these problems. In these cases the best solution found within the branch and bound tree is used to calculate the Gaps shown in the table. It is clear that for these problems, which have been specifically designed to try to defeat our exact approach, the Branch & Bound algorithm has to work much harder.

Table 3. Comparison of structured problems.

Problem			GA-F		GA-P		PSS		SA		Branch & Bound		
n	*b*	*D*	Gap	CPU	Gap	CPU	Gap	CPU	GAP	CPU	Nodes	Gap	CPU
30	4	3	0.07	4.17	1.90	5.41	0.14	16.65	0.42	17.74	16	8.11	0.01
30	4	4	0.00	4.10	1.40	5.43	0.23	14.04	0.17	37.61	71	11.68	0.03
30	4	5	0.00	3.81	1.16	5.45	0.15	15.45	0.24	26.18	3196	14.27	1.00
30	4	6	0.08	4.26	0.62	7.82	0.10	17.52	0.06	25.89	210937	16.05	59.94
30	4	7	0.11	4.03	0.43	7.80	0.05	16.93	0.30	13.39	–	17.13	***
30	5	3	0.00	3.40	1.81	7.81	0.67	13.55	0.22	15.34	7	4.18	0.01
30	5	4	0.12	3.35	1.60	5.43	0.20	12.54	0.07	21.08	57	8.86	0.02
30	5	5	0.00	3.33	1.73	5.67	0.30	13.25	0.15	24.27	2566	11.67	0.76
30	5	6	0.09	3.45	0.81	5.42	0.17	15.31	0.07	20.20	169357	13.95	46.74
30	5	7	0.17	3.46	0.44	5.70	0.10	14.68	0.27	16.83	–	15.34	***
50	4	3	0.00	24.34	0.01	18.29	0.70	47.13	0.26	87.73	16	7.62	0.02
50	4	4	0.00	23.04	0.76	18.17	0.69	58.25	0.62	85.77	71	10.47	0.06
50	4	5	0.72	24.11	0.74	18.19	0.35	67.13	0.43	105.33	3196	12.89	2.04
50	4	6	0.15	22.57	0.18	18.16	0.64	50.33	0.10	119.45	210947	14.86	124.82
50	4	7	0.57	23.66	0.61	18.13	0.37	50.96	0.15	113.61	–	15.72	***
50	5	3	0.03	19.87	0.03	18.16	0.75	73.85	1.68	58.83	7	3.94	0.01
50	5	4	0.00	19.60	0.10	18.13	0.39	83.34	0.29	86.94	57	7.96	0.05
50	5	5	0.54	20.00	0.70	18.24	0.47	51.90	0.12	73.05	2566	10.50	1.58
50	5	6	0.07	19.48	0.07	18.07	0.42	68.91	0.08	110.98	169357	12.91	97.88
50	5	7	0.32	19.70	0.56	18.09	0.27	59.93	0.10	115.80	–	14.07	***
70	4	3	0.07	70.23	1.33	41.91	1.09	151.67	1.54	187.25	16	6.90	0.04
70	4	4	0.58	68.68	1.59	41.86	0.82	116.92	0.97	168.55	71	10.00	0.12
70	4	5	0.68	65.28	1.28	42.46	1.17	103.22	1.13	232.69	3308	12.04	3.54
70	4	6	0.22	69.48	1.56	76.19	0.85	96.86	0.83	149.36	210945	13.57	220.57
70	4	7	0.56	68.53	1.56	42.78	0.52	148.72	0.55	140.00	–	14.92	***
70	5	3	0.00	59.67	1.26	42.05	1.21	162.65	2.15	145.86	7	3.56	0.03
70	5	4	0.12	59.89	1.61	41.89	1.31	94.14	3.42	128.03	57	7.61	0.10
70	5	5	0.57	61.00	1.38	42.34	0.83	124.28	4.61	121.18	2566	9.78	2.76
70	5	6	0.09	59.83	1.50	42.58	0.68	120.44	2.06	133.96	169357	11.75	172.51
70	5	7	0.52	57.97	1.68	46.57	0.52	131.59	2.60	131.94	–	13.35	***
100	3	3	0.17	257.55	1.40	107.11	2.40	291.00	2.79	418.42	141	9.38	0.14
100	3	4	1.10	271.85	0.86	107.38	1.58	291.83	2.55	334.96	285	11.78	0.25
100	3	5	0.07	314.29	0.69	107.22	1.29	307.40	1.21	316.85	141	13.42	0.31
100	3	6	0.58	203.95	0.54	107.62	1.20	246.50	2.70	315.16	597	14.44	0.67
100	3	7	0.58	233.01	0.60	107.18	1.02	223.01	1.89	300.68	4633	15.21	2.51
100	4	3	0.00	245.80	0.15	106.84	2.46	325.74	1.43	310.56	16	6.27	0.06

(*Continued on next page.*)

Table 3. (*Continued*).

Problem			GA-F		GA-P		PSS		SA		Branch & Bound		
n	*b*	*D*	Gap	CPU	Gap	CPU	Gap	CPU	GAP	CPU	Nodes	Gap	CPU
100	4	4	0.38	240.35	0.77	107.07	1.77	267.31	2.21	506.33	71	9.03	0.19
100	4	5	0.02	254.46	0.51	106.72	1.03	442.97	1.71	355.71	3196	11.33	5.15
100	4	6	0.51	233.18	0.52	107.15	0.92	295.33	1.90	494.40	215137	12.68	325.11
100	4	7	0.51	232.48	0.58	106.34	0.86	242.35	1.94	272.15	–	13.62	***
100	5	3	0.00	220.06	0.02	106.81	1.39	563.59	0.75	322.57	7	3.23	0.04
100	5	4	0.05	213.79	0.03	106.84	2.01	291.62	1.65	294.67	57	6.90	0.14
100	5	5	0.00	219.51	0.52	106.71	1.49	209.52	3.57	284.68	2566	9.24	4.07
100	5	6	0.46	206.02	0.49	107.13	1.13	268.19	5.67	265.44	169357	10.94	254.84
100	5	7	0.48	207.28	0.54	106.44	0.88	277.81	4.00	229.29	–	12.10	***

We observe that the computational effort expended by the heuristic algorithm seems to be dependent only on n and not on D. In other words, for a given value of n, the CPU time required by a heuristic is fairly uniform, irrespective of D. This is not the case for the Branch & Bound algorithm, for which the CPU effort increases dramatically with D for a fixed n. Although the Branch & Bound algorithm still finds optimal solutions fairly rapidly for most problems with $D < 6$, we find that the heuristics start to out-perform the exact approach for the higher dimensioned problems (in terms of computational effort). It is here that the usefulness of our heuristics is highlighted. Also, rather surprisingly, the size of the upper Gap produced by the heuristic appears to decrease slightly as D increases.

In terms of relative performance, in terms of both Gap as well as CPU time, it is surprising to note that GA-F performs consistently better than GA-P, PSS or SA for the STR problems. On average, over all problems in this data set, the Gaps produced by GA-F, GA-P, PSS and SA are 0.25%, 0.86%, 0.83% and 1.36% respectively. It is not clear why the GA approaches should overtake the other two heuristics for these types of problems.

We then consider the SHRD data set which represents a few really hard problems. The results of our experiments with these problems are presented in Table 4. The artificial problems in Table 4 are obviously much harder than any of the others presented so far.

Here again, Volg was not able to solve *any* of the problems to optimality in a reasonable amount of computational time. Despite the small number of nodes, in these problems, the Branch & Bound algorithm was not able to solve *any* of these within the 10 minute limit on CPU time. Hence for the Branch & Bound algorithm, we only present the initial lower Gap and the final upper Gap. The CPU times required to obtain exact solutions for each of these problems using our B&B algorithm exceeds 600.0 seconds. The initial upper bound for the Branch & Bound algorithm is obtained from the PSS approach. We note that, in many cases, the final upper Gap reported by Branch & Bound is an improvement on the PSS Gap. We do however, note that this final solution may not represent the optimal solution. In some cases, the enumeration algorithm is unable to improve on the PSS upper bound.

Table 4. Comparison of hard problems.

Problem			GA-F		GA-P		PSS		SA		Branch & Bound	
n	type	b	Gap	CPU	Gap	CPU	Gap	CPU	GAP	CPU	L Gap	U Gap%
15	1	3	5.33	0.55	3.61	3.76	1.72	2.00	3.78	2.04	71.82	0.00
15	2	3	2.01	0.58	4.68	4.44	0.67	2.32	0.67	2.01	80.77	0.00
15	1	4	4.62	0.41	16.86	1.20	2.08	1.92	3.70	3.97	62.12	0.00
15	2	4	6.28	0.41	10.47	1.21	0.70	2.18	1.16	0.87	73.26	0.00
15	1	5	6.19	0.36	11.80	1.20	0.00	2.96	0.59	1.75	51.62	0.00
15	2	5	5.12	0.36	10.54	1.23	0.90	3.15	2.41	5.23	65.36	0.00
20	1	3	0.00	1.44	6.70	2.23	0.45	5.70	1.00	10.39	85.79	0.45
20	2	3	4.94	0.98	12.30	2.22	0.09	5.23	0.00	7.01	82.41	0.09
20	1	4	0.24	0.73	6.72	2.21	0.00	3.90	1.59	7.95	80.81	0.00
20	2	4	5.82	1.10	3.09	2.21	0.00	4.85	1.11	11.53	75.74	0.00
20	1	5	1.58	0.88	6.80	2.20	0.47	4.83	0.00	7.01	75.16	0.47
20	2	5	5.87	0.88	8.25	2.20	0.00	10.12	0.95	10.53	68.89	0.00
25	1	3	2.74	2.57	2.35	3.73	0.00	13.35	0.78	15.08	88.13	0.00
25	2	3	3.57	2.37	7.53	3.71	0.00	12.80	2.51	10.44	87.56	0.00
25	1	4	2.09	1.83	5.58	3.68	0.00	12.43	1.32	11.93	83.57	0.00
25	2	4	3.59	2.03	3.97	3.70	0.00	13.02	2.44	11.31	82.96	0.00
25	1	5	3.18	1.74	2.58	3.64	1.69	8.18	3.38	16.99	78.95	0.00
25	2	5	2.05	1.76	1.37	3.66	1.07	10.09	2.63	14.61	78.24	0.00
30	1	3	5.37	2.77	5.37	5.60	0.00	18.46	1.97	20.21	89.64	0.00
30	2	3	2.82	3.33	3.97	5.55	0.00	17.13	2.17	68.13	90.88	0.00
30	1	4	3.67	3.62	7.59	5.52	0.00	20.31	2.37	15.21	85.85	0.00
30	2	4	1.99	3.34	1.62	5.50	0.00	32.98	2.41	68.81	87.49	0.00
30	1	5	2.69	2.93	4.78	5.51	0.00	18.82	2.36	27.11	82.04	0.00
30	2	5	0.34	3.34	0.94	9.68	0.34	20.47	3.30	18.02	83.92	0.00

The average upper Gaps produced by the GA-F, GA-P, PSS and SA are 3.22%, 6.22%, 0.42% and 1.86% respectively. It is clear that the GA heuristics are not so efficient for these harder problems. Even the SA, which performed favourably (when compared to PSS) on the other data sets, is relatively outperformed by PSS in terms of Gaps as well as CPU time.

In Table 4, a figure of 0.00 in each row in the corresponding (upper) Gap column reflects that this is the best available solution for the problem. The GA-F heuristic finds the best solution for only one problem with $n = 20$, type$=1$, $b = 3$. For this particular problem, none of the other methods were able to produce this best solution, although PSS and Branch & Bound produce a solution with a Gap of only 0.45%. GA-P does not find the best solution in any of the problems. The SA algorithm identifies the best solution in 2 of the 24 problems with $n = 20$, type$=2$, $b = 3$ and $n = 20$, type$=1$, $b = 5$ respectively. Again, for these two problems, the SA algorithm is the only one to identify this best solution, although (for both of these problems) the PSS identifies solutions with a very small upper Gap.

The PSS heuristic finds the best solution in 13 of the 24 problems, while we obtain the best solution in 21 of the 24 problems using the Branch & Bound algorithm (which is also a heuristic for this data set since we can not prove optimality). Note that since we are starting with the PSS upper bound, the upper Gap percentage for PSS really shows how much improvement the Branch & Bound method has made in the cases where it finds the best solution. However, in terms of time, the PSS heuristic is significantly faster that the Branch & Bound approach. Only very small improvements in the solution are made by the Branch & Bound algorithm during the 600 seconds of CPU time until it is terminated.

6. Conclusions

In this paper, we presented a suite of algorithms for solving the Degree Constrained Minimum Spanning Tree (DCMST) on a graph, which is the problem of generating a minimum spanning tree with constraints on the number of arcs that can be incident to vertices of the graph. We developed several heuristics for the DCMST, including simulated annealing (SA), two variants of a genetic algorithm (GA-F and GA-P) and also a method based on problem space search (PSS). We proposed alternative tree representations to facilitate the efficient search of the solution space. This was used in both our GA implementations and a novel modification of this was employed in the SA neighbourhood search. We also develop a new exact solution approach based on the combinatorial structure of the problem.

We compared the computational performance of the new heuristics and the new exact solution approach against the performance of an exact solution approach in the literature that is due to Volgenant (1989). We tested all of these approaches using standard problems taken from the literature. We also tested all of these approaches on some new (and difficult) test problems that we generated and placed in the literature.

In general, we conclude that the PSS algorithm works best over all the problems. However, for one of the data sets, we do note that GA-F performed significantly better. The SA algorithm, which used a novel tree representation, performs in an average manner.

While the exact algorithms performed very well on some of the easier problems, the usefulness of our heuristic methods was best illustrated in the tests that we carried out on the more difficult problems that we have placed in the literature.

The new tree representation that we employed in this paper (and its modification which was used in our SA algorithm) can be generalised to other tree optimisation problems as well. It will probably be effective in designing suitable heuristics for (say) the Capacitated Minimum Spanning Tree and other such problems.

Acknowledgments

The authors wish to acknowledge the three anonymous referees for their useful comments and suggestions. The third author, Yazid Sharaiha, undertook this research while he visited CSIRO Mathematical and Information Sciences (CMIS), Australia. He wishes to acknowledge the partial support from The Royal Society, provided to facilitate this visit.

References

Ahuja, R.K., J.B. Orlin, and A. Tiwari. (1995). “A Greedy Genetic Algorithm for the Quadratic Assignment Problem.” In *Proceedings, The Sixth International Conference on Manufacturing Engineering*. Melbourne, Australia, 29 Nov.–1 Dec. 1995, Vol. 2, pp. 567–575.

Arora, S. (1996). “Polynomial Time Approximation Scheme for Euclidean tsp and Other Geometric Problems.” In *Proceedings 37th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, pp. 2–11.

Ausiello, G., P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti Spaccamela, and M. Protasi. (1999). *Approximate Solution of NP-Hard Optimization Problems*. Berlin: Springer-Verlag.

Beasley, J.E. (1990). “Or-Library: Distributing Test Problems by Electronic Mail.” *Journal of the Operational Research Society* 41(11), 1069–1072.

Boldon, B., N. Deo, and N. Kumar. (1996). “Minimum-Weight Degree-Constrained Spanning Tree Problem: Heuristics and Implementation on an Simd Parallel Machine.” *Parallel Computing* 22(3), 369–382.

Collins, N., R. Eglese, and B. Golden. (1998) “Simulated Annealing: An Annotated Bibliography.” *American Journal of Mathematical and Management Sciences* 8(3–4), 209–307.

Craig. G., M. Krishnamoorthy, and M. Palaniswami. (1996). “Comparison of Heuristic Algorithms for the Degree Constrained Minimum Spanning Tree.” In I.H. Osman and J.P. Kelly (eds.), *Metaheuristics: Theory and Applications*. Boston: Kluwer.

Cresenzi, P. and V. Kann. (1995). “A Compendium of np Optimization Problems.” Technical report, Università di Roma “La Sapienza.” http://www.nada.kth.se/∼viggo/problemlist/compendium.html.

Davis, L. (1991). *Handbook of Genetic Algorithms*. New york: Van Nostrand.

Deo, N. and N. Kumar. (1997). “Computation of Constrained Spanning Trees: A Unified Approach.” *Network Optimization* 450, 194–220.

Frederickson, G.N. (1983). “Data Structures for On-Line Updating of Minimum Spanning Trees.” In *Proceedings of the 15th Annual ACM Symposium of the Theory of Computing*. Boston, MA.

Frederickson, G.N. (1985). “Data Structures for On-Line Updating of Minimum Spanning Trees.” *SIAM J. Comput.* 14(4), 781–798.

Gabow, H.N. (1978). “Good Algorithm for Smallest Spanning Trees with a Degree Constraint,” *Networks* 8(3), 201–208.

Garey, M.R. and D.S. Johnson. (1979). *Computers and Intractability, A Guide to the Theory of NP–Completeness*. San Francisco: Freeman.

Gavish, B. (1982). “Topological Design of Centralized Computer Networks—Formulations and Algorithms.” *Networks* 12, 355–377.

Gavish, B. (1985). “Augmented Lagrangean Based Algorithms for Centralized Network Design.” *IEEE Transactions on Communications* 33(12), 1247–1257.

Gavish, B. (1992). “Topological Design of Centralized Computer Networks—the Overall Design Problem.” *European Journal of Operations Research* 58, 149–172.

Gen, M., G. Zhou, and M. Takayama. (1998). “A Comparative Study of Tree Encodings on Spanning Tree Problems.” *IEEE Conference on Evolutionary Computation*.

Gendreau, M., A. Hertz, and G. Laporte. (1994). “A Tabu Search Heuristic for the Vehicle Routing Problem.” *Management Science* 40, 1276–1290.

Glover, F. (1994). “Genetic Algorithms and Scatter Search: Unsuspected Potentials.” *Statistics and Computing* 4, 131–140.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Held, M. and R.M. Karp. (1970). “The Travelling-Salesman Problem and Minimum Spanning Trees.” *Operations Research* 18, 1138–1162.

Held, M. and K.R.M. (1971). “The Travelling–Salesman Problem and Minimum Spanning Trees: Part II.” *Mathematical Programming* 1, 6–25.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press.

Kawatra, R. (1997). “Lagrangean Based Heuristic for the Degree Constrained Minimal Spanning Tree.” In *Proceedings of the Annual Meeting of the decision Sciences Institute*. Nov. 22–25, San Diego, USA.

Khuller, S., B. Raghavachari, and N. Young. (1996). “Low Degree Spanning Trees of Small Weight.” *SIAM Journal on Computing* 25, 355–368.

Moon, J.W. (1967). “Various Proofs of Cayley’s Formula for Counting Trees.” In F. Harary (ed.), *A Seminar on Graph Theory*. New York: Holt, Rinehart and Winston. pp. 70–78.

Narula, S.C. and C.A. Ho. (1980). “Degree Constrained Minimum Spanning Tree.” *Computers and Operations Research* 7, 239–249.

Osman, I.H. and G. Laporte. (1996). “Metaheuristcs: A Bibliography.” *Annals of Operations Research* 63, 513–623.

Palmer, C.E. and A. Kershenbaum. (1994). “Representing Trees in Genetic Algorithms.” Technical report, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, 1994.

Reeves, C.R. (ed.). (1993). *Modern Heuristic Techniques for Combinatorial Problems*. Oxford: Blackwell Scientific Publications.

Rutenbar, R.A. (1989). “Simulated Annealing Algorithms: An Overview.” *IEEE Circuits and Devices Magazine* 5(1), 19–26.

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

Sharaiha, Y.M., M. Gendreau, G. Laporte, and I.H. Osman. (1997). “A Tabu Search Algorithm for the Capacitated Shortest Spanning Tree.” *Networks* 3, 161–171.

Storer, R.H., S.W. Flanders, and S.D. Wu. (1996). “Problem Space Local Search for Number Partitioning.” *The Annals of Operations Research* 63, 465–487.

Storer, R.H., S.D. Wu, and R. Vaccari. (1992). “New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling.” *Management Science* 38(10), 1495–1509.

Volgenant, A. (1989). “A Langrangean Approach to the Degree Constrained Minimum Spanning Tree Problem.” *European Journal of Operational Research* 39, 325–331.

Volgenant, A. and R. Jonker. (1983). “The Symmetric Travelling Salesman Problem and Edge Exchanges in Minimum 1–Trees.” *European Journal of Operational Research* 12, 394–403.

Yamamoto, Y. (1978). “The Held–Karp Algorithm and Degree–Constrained Minimum 1–Trees.” *Mathematical Programming* 15, 228–231.

Zhou, G. and M. Gen. (1997). “Approach to the Degree-Constrained Minimum Spanning Tree Problem Using Genetic Algorithms.” *Engineering Design and Automation* 3(2), 156–165.

Zhou, G. and M. Gen. (1997). “A Note on Genetic Algorithms for Degree Constrained Minimum Spanning Tree Problems.” *Networks* 30, 105–109.