3

A New Ant Colony Optimization Approach for the Degree-Constrained Minimum Spanning Tree Problem Using Prüfer and Blob Codes Tree Coding

Yoon-Teck Bau, Chin-Kuan Ho and Hong-Tat Ewe
Faculty of Information Technology, Multimedia University
Malaysia

1. Introduction

This chapter describes a novel ACO algorithm for the degree-constrained minimum spanning tree (d-MST) problem. Instead of constructing the d-MST directly on the construction graph, ants construct the encoded d-MST. Two well-known tree codings are used: the Prüfer code, and the more recent Blob code (Picciotto, 1999). Both of these tree codings are bijective because they represent each spanning tree of the complete graph on $|V|$ labelled vertices as a code of $|V|$-2 vertex labels. Each spanning tree corresponds to a unique code, and each code corresponds to a unique spanning tree. Under the proposed approach, ants will select graph vertices and place them into the Prüfer code or Blob code being constructed. The use of tree codings such as Prüfer code or Blob code makes it easier for the proposed ACO to solve another variant of the d-MST problem with both lower and upper bound constraints on each vertex (lu-dMST). A general lu-dMST problem formulation is given. This general lu-dMST problem formulation could be used to denote d-MST problem formulation also. Subsequently, Prüfer code and Blob code tree encoding and decoding are presented and then followed by the design of two ACO approaches using these tree codings to solve d-MST and lu-dMST problems. Next, results from these ACO approaches are compared on structured hard (SHRD) graph data set for both d-MST and lu-dMST problems, and important findings are reported.

2. Problem Formulation

In this chapter, a special case of degree-constrained minimum spanning tree where the lower and upper bound of the number of edges is imposed on each vertex is considered. This similar to the problem being solved by Chou et al. (2001), and is named lu-dMST in this chapter. Chou et al. (2001) named this problem as DCMST. The d-MST problem is different since it has only the upper bound constraint. Chou et al. (2001) also proposed the following notation to be used for the lu-dMST problem formulation:
$G = (V, E)$ connected weighted undirected graph.
i, j = index of labelled vertices $i, j = 0, 1, 2, \ldots, |V - 1|$.

Source: Swarm Intelligence: Focus on Ant and Particle Swarm Optimization, Book edited by: Felix T. S. Chan and Manoj Kumar Tiwari, ISBN 978-3-902613-09-7, pp. 532, December 2007, Itech Education and Publishing, Vienna, Austria

$V = \{v_0, v_1, ..., v_{|V|-1}\}$ is a finite set of vertices in the G.
$E = \{e_{ij} \mid i \in V, j \in V, i \neq j\}$ is a finite set of edges in the G.
T = set of all spanning trees corresponding to the G.
$\mathbf{x}$ = a subgraph of G.
C_{ij} = nonnegative real number edge cost that connect vertex i and vertex j.
$L_d(i)$ is lower bound degree constraint on vertex i. Lower bound can vary from vertex to vertex.
$U_d(i)$ is upper bound degree constraint on vertex i. Upper bound can vary from vertex to vertex.

$$\min\left\{ z(\mathbf{x}) = \sum_{\substack{i,j \in V \\ i<j}} C_{ij} X_{ij}. \right\} \tag{1}$$

subject to:

$$\sum_{\substack{j \in V \\ i \neq j}} X_{ij} \geq L_d(i),\ i \in V. \tag{2}$$

$$\sum_{\substack{j \in V \\ i \neq j}} X_{ij} \leq U_d(i),\ i \in V. \tag{3}$$

$$\sum_{\substack{i,j \in N \\ i<j}} X_{ij} \leq |N| - 1, N \subset V. \tag{4}$$

$$\sum_{\substack{i,j \in V \\ i<j}} X_{ij} = |V| - 1. \tag{5}$$

$$X_{ij} = \begin{cases} 1, \text{ if edge } e_{ij} \text{ is part of the subgraph } \mathbf{x} \mid i, j \in V, \mathbf{x} \in T; \\ 0, \text{ otherwise.} \end{cases} \tag{6}$$

The objective function (1) seeks to minimize the total connection cost between vertices. The total cost could be distance, material cost, or customers' requirement cost. The subconstraint ($i < j$) shows that graph is symmetric because vertex i must be less than vertex j where $i, j \in V$. Constraints (2) and (3) specify the lower and upper bounds degree constraints on the number of edges connecting to a vertex. The lower and upper bounds can vary from vertex to vertex. In the d-MST problem, there is only a degree constraint on each vertex. This is given by a constant value d. For the d-MST problem, the lower bound equals 1 and the upper bound equals d on each vertex. Therefore the lu-dMST problem formulation is a generalization of d-MST. At the same time, the lu-dMST problem is also NP-hard because the lu-dMST problem is a general problem formulation that can be used to represent d-MST problem (Garey & Johnson, 1979; Sipser, 2006).
Constraint (4) is an anticycle constraint and constraint (5) indicates that the number of edges in a spanning tree is always equal to the number of vertices minus one. At the same time, the designed networks should not have self-loop, cycles and missing vertices. Equation (6) expresses the binary decision variable X_{ij} equals to one if the edge between vertices i to j is part of the subgraph $\mathbf{x}$, and $\mathbf{x}$ is a spanning tree in T; zero, otherwise. A subgraph $\mathbf{x}$ of G is said to be a spanning tree in T if $\mathbf{x}$:

a. contains all the vertices of G and the vertices can be in non-order form;
b. is connected and graph contains no cycles.

Note that in a complete graph having $|V|$ vertices, the number of edges, $|E|$, is $|V|(|V|-1)/2$, and the number of spanning trees is $|V|^{|V|-2}$.

3. Prüfer code and Blob code tree codings

The Prüfer code of spanning trees is based on Prüfer's constructive proof of Cayley's Formula. Cayley showed that the number of distinct spanning trees in a complete undirected graph on $|V|$ vertices is $|V|^{|V|-2}$ (Cayley, 1889; Gross & Yellen, 2006). Prüfer described a one-to-one mapping between these trees and strings of length $|V|-2$ over an integer of $|V|$ vertex labels (Prüfer, 1918; Gross & Yellen, 2006). Thus, a Prüfer code of length $|V|-2$ whose vertices are the labels {0, 1, …, $|V|-1$} from a spanning tree of the complete graph on $|V|$ vertices for $|V| \geq 2$ is any sequence of integers between 0 and $|V|-1$, with repetitions allowed. The following Fig. 1 shows Prüfer tree encoding algorithm that constructs a Prüfer code from a given standard labelled tree. It defines a encoding function $f_e : T_{|V|} \rightarrow C_{|V|-2}$ from the set $T_{|V|}$ of trees on $|V|$ labelled vertices to the set $C_{|V|-2}$ of Prüfer code of length $|V|-2$. For example, a Prüfer code (3, 3, 6, 4, 0) corresponds to a spanning tree on seven vertices graph in Fig. 2. The first position value for Prüfer code is 3 because the Prüfer encoding algorithm finds the neighbour of vertex v of degree 1 with the smallest label in the spanning tree T is 3 whereby $v = 1$. Then the vertex labelled $v = 1$ is removed from the spanning tree T. This process is repeated to find the second position value for Prüfer code until only two vertices are remained in the spanning tree T. Two vertices remained in the spanning tree T as in the example mentioned below are vertices labelled 0 and 6. Notice also for example in Fig. 2 that the degree of each vertex in the spanning tree can be easily checked because it is one more than the number of times its label appears in the Prüfer code.

Fig. 3 shows the Prüfer decoding algorithm that maps a given Prüfer code to a standard labelled tree. The Prüfer decoding algorithm defines a function $f_d : C_{|V|-2} \rightarrow T_{|V|}$ from the set of Prüfer code of length $|V|-2$ to the set of labelled trees on the $|V|$ vertices. For example, the Prüfer decoding algorithm identifies the tree's edges in this order: (1, 3), (2, 3), (3, 6), (5, 4), (4, 0), and (0, 6) as in Fig. 2. The Prüfer code's integers appear as the second vertices in the tree's first five edges. The last edge (0, 6) is joined by remaining two integers in list L (line 12 of Fig. 3) to produce the spanning tree with its vertex-labelling. Notice that the tree obtained in Fig. 2 by Prüfer decoding of the sequence (3, 3, 6, 4, 0) is the same as the tree that used by Prüfer-encoded sequence of (3, 3, 6, 4, 0) at the beginning. This inverse relationship between the encoding and decoding functions asserts that the decoding function $f_d : C_{|V|-2} \rightarrow T_{|V|}$ is the inverse of the encoding function $f_e : T_{|V|} \rightarrow C_{|V|-2}$.

```
1  labelledTreeToPruferCode(T = (V, E), Cij)
2  code ← ( )
3  Initialise T to be the given tree.
4  for i = 1 to |V|-2 do
5     Let v be the vertex of degree 1 with the smallest label in T.
6     Let code[i-1] be the label of the only neighbour of v.
7     T ← T – {v}
8  return code
```

Figure 1. The pseudocode of Prüfer encoding from the labelled tree to its Prüfer code

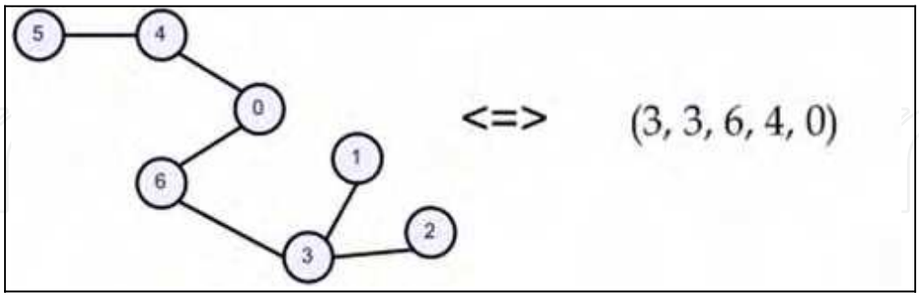


Figure 2. A Prüfer code and the spanning tree on seven vertices that it represents and vice versa via Prüfer encoding and decoding algorithms

1	**pruferCodeToLabelledTree**(*code*)
2	Initialise *code* as the Prufer input sequence of length $\|V\|$-2.
3	Initialise forest F as $\|V\|$ isolated vertices, labelled from 0 to $\|V\|$-1.
4	$L \leftarrow \{0, 1, \ldots, \|V\|\text{-}1\}$
5	$E_T \leftarrow \{\}$
6	**for** $i = 1$ **to** $\|V\|$-2 **do**
7	Let k be the smallest integer in list L that is not in the *code*.
8	Let j be the first integer in the *code*.
9	$E_T \leftarrow E_T \cup \{(k, j)\}$
10	$L \leftarrow L - \{k\}$
11	Remove the first occurrence of j from the *code*.
12	Add an edge joining the vertices labelled with the two remaining integers in list L.
13	**return** E_T

Figure 3. The pseudocode of Prüfer decoding from the Prüfer code to its labelled tree

There are many other mappings from integers of $|V|$-2 vertex labels to spanning trees. Picciotto (1999) has described three tree codings, different from Prüfer code. One of the tree codings is called the Blob code. In Picciotto's presentation, Prüfer codes decoded as Blob codes represent directed spanning trees rooted at vertex 0. In such a tree, there is a directed path from every vertex to vertex 0, and only vertex 0 has no out-edge. Ignoring the edges's direction yields an undirected spanning tree.

Figure 4 shows the Blob encoding algorithm for finding Blob code for a spanning tree. A *blob* is an aggregation of one or more vertices. This algorithm is progressively identifying vertices, starting at $|V|$-1 and ending with a blob-vertex consisting of all the vertices from 1 to $|V|$-1. As the *blob* grows, so does the code; meanwhile, the number of directed edges shrinks. At first, an undirected spanning tree is temporarily regarded as a directed spanning tree rooted at vertex labelled 0 to determine the successor $succ(v)$ of every vertex $v \in [1, |V|\text{-}1]$ where $succ(v)$ is the first vertex on the unique path from vertex v to vertex 0 in a spanning tree. The Blob encoding algorithm uses this directed spanning tree rooted at vertex labelled 0 as a set of directed edges whose vertices are the labels {0, 1, ..., $|V|$-1} as its input. The algorithm uses two functions: $succ(v)$ returns the first vertex on the unique path from vertex v to vertex 0 in a spanning tree, and $(path(v) \cap blob)$ returns TRUE if the directed path (an ordered list of vertices) using those directed edges from vertex v toward vertex 0 intersects the *blob*, FALSE otherwise.

```
1   labelledTreeToBlobCode(T = (V, E), C_ij)
2   blob ← {|V|-1}
3   blobCode ← ( ) //an array of length |V|-2
4   for i = 1 to |V|-2 do
5     if path(|V|-1–i) ∩ blob ≠ Ø then
6       blobCode[|V|-2-i] ← succ(|V|-1-i)
7       E_T ← E_T – {(|V|-1-i → succ(|V|-1-i))}
8       blob ← blob ∪ {|V|-1-i}
9     else
10      blobCode[|V|-2-i] ← succ(blob)
11      E_T ← E_T – {(blob → succ(blob))}
12      E_T ← E_T ∪ {(blob → succ(|V|-1-i))}
13      E_T ← E_T – {(|V|-1-i → succ(|V|-1-i))}
14      blob ← blob ∪ {|V|-1-i}
15  return blobCode
```

Figure 4. The pseudocode of Blob encoding from the labelled tree to its Blob code

An example of a Blob code corresponds to a directed spanning tree on seven vertices graph is given in Fig. 5. The successor *succ*(*v*) information for this directed spanning tree is shown in Table 1. Once this table has been constructed, the Blob code corresponding to this directed spanning tree on seven vertices graph is equal to (3, 3, 6, 4, 0). Initially on line 2 of Fig. 4, a *blob* containing a single vertex 6 is created, the vertex 6 is the largest label and *blobCode* = ( ). The *blobCode* is an array of length $|V|$-2. The Blob encoding algorithm's first iteration (*path*($|V|$-1-*i*) ∩ *blob*) = (*path*(5) ∩ *blob*) is FALSE on line 5 of Fig. 4. So the **else** block is followed whereby *blobCode*[4] = 0; delete (*blob* → 0) edge; add an edge from *blob* → *succ*(5) which is 4; delete the edge (5 → 4) and put 5 into the *blob*. The second iteration (*path*(4) ∩ *blob*) is also FALSE. So the **else** block is followed whereby *blobCode*[3] = 4; delete (*blob* → 4) edge; add an edge from *blob* → *succ*(4) which is 0; delete the edge (4 → 0) and put 4 into the *blob*. The third iteration (*path*(3) ∩ *blob*) is TRUE. The **then** block is followed in the algorithm whereby *blobCode*[2] = 6 which is *succ*(3); delete the edge (3 → 6) and put 3 in the blob. This process continues through two more iterations which *blobCode*[1] = 3 and *blobCode*[0] = 3 are obtained, and hence the Blob code of length $|V|$-2 is equal to *blobCode* = (3, 3, 6, 4, 0) is determined. It happened that this Blob code is the same with Prüfer code as in Fig. 2 using the same spanning tree as an example.

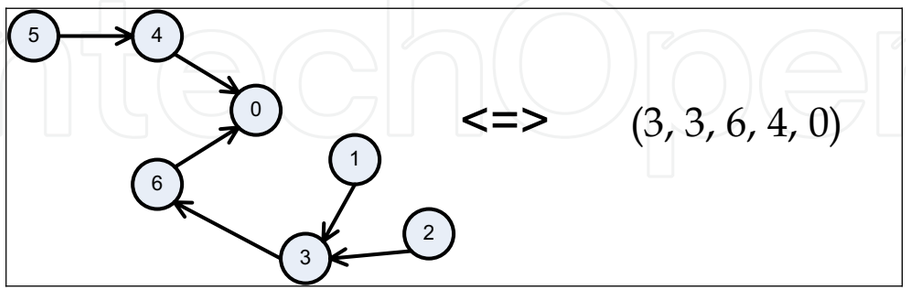


Figure 5. A Blob code and a rooted directed spanning tree on seven vertices that it represents and vice versa via Blob encoding and decoding algorithms

v	$succ(v)$
1	3
2	3
3	6
4	0
5	4
6	0

Table 1. The successor $succ(v)$ information of every vertex $v \in [1, |V|\text{-}1]$

Nevertheless Blob code is already proven to be a different coding system from the Prüfer code by Picciotto (1999) in his PhD thesis even tough Blob code contains the same number of times of its vertex label as it appears in the Prüfer code for the same spanning tree representation. The reason for this is the sequences of both Blob code and Prüfer code can have distinct vertex label for each of their sequence position to represent the same spanning tree. An example suffices to prove that Blob code = (2, 4, 4, 6, 2, 4) is different from the Prüfer code = (6, 2, 4, 2, 4, 4) even though these codes are used to represent the same spanning tree.

To identify the directed spanning tree that a Blob code represents, the Blob decoding algorithm begins with a single directed edge from a *blob* to vertex 0. This *blob* contains all the other vertices except vertex labelled 0, and as the algorithm proceeds, it always contains vertices numbered i, i+1, …, $|V|$-2 as i moves from 1 to $|V|$-2. The algorithm scans the code and adjusts the developing spanning tree depending on whether or not the directed path from each vertex in Blob code toward vertex 0 intersects the *blob*, which shrinks by one vertex on each iteration. The following Fig. 6 summarizes the Blob decoding algorithm, which also uses the same two functions as Blob encoding algorithm: $succ(v)$ and $(path(v) \cap blob)$. The edges directions are ignored to obtain the undirected spanning tree that the Blob code represents.

```
1   blobCodeToLabelledTree(blobCode)
2   blob ← {1, 2, …, |V|-1}
3   E_T ← {(blob → 0)}
4   for i = 1 to |V|-2 do
5     blob ← blob – {i}
6     if path(blobCode[i–1]) ∩ blob ≠ ∅ then
7       E_T ← E_T ∪ {(i → blobCode[i–1])}
8     else
9       E_T ← E_T ∪ {(i → succ(blob))}
10      E_T ← E_T – {(blob → succ(blob))}
11      E_T ← E_T ∪ {(blob → blobCode[i–1])}
12  // now blob is a vertex labelled |V|-1
13  blob ← {|V|-1} in any edges where the blob appears.
14  return E_T
```

Figure 6. The pseudocode of Blob decoding from the Blob code to its labelled tree

Figure 5 shows the Blob code (3, 3, 6, 4, 0) and the spanning tree to which it decodes via the Blob decoding algorithm. The algorithm identifies the tree's (directed) edges in this order: (1, 3), (2, 3), (3, 6), (4, 0), (5, 4), and (6, 0). Initially on line 2 of Fig. 6, the *blob* contains vertices 1 through 6 and the tree consists of the single edge (*blob* → 0). The algorithm's first iteration removes vertex 1 from the *blob*. The *blobCode*[0] = 3 and the *blob* contains vertex 3, so that (*path*(3) ∩ *blob*) is TRUE and the edge (1 → 3) is added to the tree. The second iteration removes vertex 2 from the *blob*. The *blobCode*[1] = 3, (*path*(3) ∩ *blob*) is also TRUE, and the edge (2 → 3) is added to the tree. The third iteration removes vertex 3 from the *blob*. The *blobCode*[2] = 6, (*path*(6) ∩ *blob*) is TRUE, and the edge (3 → 6) is added to the tree. The fourth iteration removes vertex 4 from the *blob*. The *blobCode*[3] = 4, (*path*(4) ∩ *blob*) is FALSE. So the **else** block is followed whereby *succ*(*blob*) which is 0; an edge (4 → 0) is added to the tree; delete the edge (*blob* → 0) and add an edge (*blob* → 4). This process continues through one more iteration, each of which increases the number of the tree's edges by one. Then, the *blob* itself is replaced by vertex 6 as on line 13 of Fig. 6. The Blob code's integers appear as the destination vertices of the first five edges. An efficient implementation of the algorithm represents the directed edges in an array that is indexed by the vertex labels. If (*i* → *j*) is an edge, then the array entry indexed *i* holds *j*. As in Prüfer codes, the degree of each vertex in the spanning tree is one more than the number of times its label appears in the Blob code decoded by the Blob decoding algorithm. This is the same directed spanning tree that was encoded by the Blob encoding algorithm as shown as example above. So, the Blob decoding algorithm has indeed reversed the Blob encoding algorithm.

4. An ACO algorithm using Prüfer code and Blob code tree codings for d-MST problem

In the design of an ACO algorithm, it has been customary to have the ants work directly on the construction graph. For pheromones associated with the graph edges, a common difficulty is the number of pheromone updates is in the order of $O(|V|^2)$, V being the set of vertices of the construction graph. A new ACO algorithm for the d-MST problem is proposed that can address this challenge in a novel way. Instead of constructing the d-MST directly on the construction graph, ants construct the encoded d-MST as solution components. Two well-known tree codings either by using the Prüfer code or the more recent Blob code is used. Under the proposed approach, ants will select graph vertices and place them into the Prüfer code or Blob code being constructed. The advantages of using tree codings as ACO solution components are it reduces the complexity of the number of pheromone update operations to $O(|V|-2)$ attributed to the length of the Prüfer or Blob codes, capable of representing all possible spanning trees from these tree codings, capable of representing only graph spanning trees, and the degree of each vertex in the decoded spanning tree is easily determined whether it's satisfied the degree constraint, d or not. The degree of each vertex in the spanning tree is one more than the number of times its label appears in the Prüfer or Blob codes.

The pseudocode of the proposed ACO approach for d-MST problem is given in Fig. 7. Both of the Prüfer coding and the Blob coding can be applied using this pseudocode. This ACO approach uses local search procedure. The pseudocode of the local search procedure using exchange mutation is given in Fig. 8. Two separate experiments are conducted for the ACO approach in Fig. 7. The first experiment uses the Prüfer encoding and decoding algorithms.

The second experiment uses the Blob encoding and decoding algorithms. Lines from 2 to 4 of Fig. 7 set several parameters for the ACO approach. The parameters are:

- τ_0 is the initial pheromone,
- maximum edge weight cost for SHRD graph is set to $20*|V|$,
- pheromone trails τ_{vr} be the desirability of assigning vertex v to a tree code of array index r is initially set to a small value as $\tau_0 = |V|^2*20*|V|$, where $v \in [0, |V|-1]$ and $r \in [0, |V|-3]$. Note that, $[0, |V|-1]$ is the vertex labels in the spanning tree from 0 to $|V|-1$ and $[0, |V|-3]$ is the array indices of the tree code (in array structure) of length $|V|-2$ from 0 to $|V|-3$,
- *mAnts* is the number of ants,
- *antDeg*[*k*][*v*] is the array of ant *k* degree for each vertex *v* in the spanning tree where $k \in [1, mAnts]$ and $v \in [0, |V|-1]$,
- *ant*[*k*].*avlVtx* is the list of ant *k* available vertices to be selected from the spanning tree vertices where $k \in [1, mAnts]$,
- *antTreeCode*[*k*][*r*] is the array of ant *k* tree code of length $|V|-2$ where $k \in [1, mAnts]$ and $r \in [0, |V|-3]$,
- *d-PrimCode*[*r*] is the tree code of d-Prim degree-constrained spanning tree (d-ST) of length $|V|-2$ where $r \in [0, |V|-3]$. The d-Prim algorithm as described in (Narula & Ho, 1980; Knowles & Corne, 2000) is a greedy algorithm and might not always find the globally optimal solution. It is based upon alterations or additions to Prim's algorithm (Prim, 1957) for finding a MST. The *d-PrimCode* is encoded from its d-Prim d-ST by using tree encoding algorithms,
- *d-ST*^{gb}*Code*[*r*] is the tree code of global-best degree-constrained spanning tree of length $|V|-2$ where $r \in [0, |V|-3]$. Initially *d-ST*^{gb}*Code* ← *d-PrimCode*,
- *ant_d-STCost*[*k*] is the total weight cost of d-ST^k of *antTreeCode*[*k*] where $k \in [1, mAnts]$. The *antTreeCode*[*k*] cost is computed from its d-ST by using tree decoding algorithm,
- *d-PrimCost* is the total weight cost of d-Prim d-ST. The d-Prim d-ST is determined by using d-Prim algorithm,
- L^{gb} is the total weight cost of d-ST^{gb}. Initially L^{gb} ← *d-PrimCost*,
- a positive integer which governs the influences of pheromone trails α,
- evaporation rate ρ,
- a positive integer Q, and
- *termination_cond* is the termination condition where it can be either a predefined number of iterations has been reached or a satisfactory solution has been found.

The ACO algorithm starts by initialising *d-ST*^{gb}*Code* of length $|V|-2$ to be equal to the *d-PrimCode* as on line 3 of Fig. 7. d-Prim d-ST is encoded by using tree encoding algorithm to obtain its *d-PrimCode*. Then, the ants start to construct their tree code solutions. Initially, *antDeg*[*k*][*v*] is set to 1 where $k \in [1, mAnts]$ and $v \in [0, |V|-1]$ as on line 8 of Fig. 7. The reason for this is the degree of each vertex in the spanning tree is one more than the number of times label of vertices appears in the Prüfer or Blob codes and initially for each ant their *antTreeCode*[*k*] is emptied. Next, for each ant their *ant*[*k*].*avlVtx* is initially set to {0, 1, …, $|V|-1$} where the spanning tree vertex labels start from 0 to $|V|-1$ (line 9 of Fig. 7). Line 12 of Fig. 7 the ants start to construct their first (index 0) tree code solutions by selecting a vertex *v* from *ant*[*k*].*avlVtx* randomly. A particular vertex *v* will be removed from *ant*[*k*].*avlVtx* so that the vertex *v* won't be available anymore if ($antDeg[k][v] = U_d(v)$). The reason for this is to ensure that degree constraint is not violated. For the remaining tree code

position value that is starting from its second position (index 1) to its last position (index $|V|$-3) as lines from 16 to 32 of Fig. 7, every ant will select a vertex v among the available vertices from $ant[k].avlVtx$ probabilistically by applying the roulette wheel selection (Goldberg, 1989; Michaelwicz, 1996; Dorigo & Stützle, 2004) method. According to the probability on line 25 of Fig. 7, only the pheromone trail τ_{vr} indicates the desirability of assigning vertex v to a tree code at array index r is being used. Notice also that this probability formula does not use any visibility measure because the pheromone trail τ_{vr} does not means that an edge cost connecting from vertex v (the ant k tree code array value) to vertex r (the ant k tree code array index) always exists.

After every ant k has completed their $antTreeCode[k]$ of length $|V|$-2, then the $ant\_d\text{-}STCost[k]$ is determined from their $antTreeCode[k]$ where $antTreeCode[k]$ is decoded by using the tree decoding algorithm to obtain the ant k d-ST^k. If the $ant\_d\text{-}STCost[k]$ is less costly than the current L^{gb} as on line 36 of Fig. 7, then the current $d\text{-}ST^{gb}Code$ will be replaced to be equal to $antTreeCode[k]$. Next, the local search procedure by using exchange mutation is applied as on line 39 of Fig. 7. The new mutated tree code will always produce a new feasible d-ST. The detail of exchange mutation is given in Fig. 8. The exchange mutation used here takes the current $d\text{-}ST^{gb}Code$ and the current L^{gb} as its inputs. Then, two different positions from $d\text{-}ST^{gb}Code$ are being selected randomly so that both of the position values can be exchanged. As on line 10 of Fig. 8, the number of times for the exchange mutation procedure that takes the mutated code as its input to be repeated is equal to $|V|/2$ if $|V|$ is an even number, else $(|V|+1)/2$. Notice that, lines from 14 to 30 of Fig. 8, the exchange mutation will be stopped even if the number of repetition has not been completed; if the current new mutated d-ST code is less costly than the current d-ST^{gb} code. Then, the current d-ST^{gb} code will be replaced by the better mutated d-ST code. If the mutated d-ST code is not better than the current d-ST^{gb} code, the current d-ST^{gb} code will be remained without any changes made to its spanning tree code as implied on line 31 of Fig. 8.

```
1   procedure ACO for d-MST
2     Set parameters.
3     d-ST^gbCode ← d-PrimCode //d-Prim d-ST is encoded by using tree encoding algorithm
4     L^gb ← d-PrimCost
5     while termination_cond = false do
6       for k = 1 to mAnts do
7         for v = 0 to |V|-1 do
8           antDeg[k][v] = 1 //each ant k spanning tree vertices initial degree is set to 1
9         ant[k].avlVtx ← {0, 1, …, |V|-1}
10      for k = 1 to mAnts do
11        v ← select from ant[k].avlVtx randomly
12        antTreeCode[k][0] = v
13        antDeg[k][v] = antDeg[k][v] + 1
14        if antDeg[k][v] = U_d(v) then
15          ant[k].avlVtx ← ant[k].avlVtx – {v}
16      r ← 0
17      while (r < |V|-2) do
18        r ← r + 1
19        for k = 1 to mAnts do
```

20 The ant k tree code of array index r of iteration t will select a vertex v
21 among the list of available vertices from $ant[k].avlVtx$, according to
22 probability:
23
24
25
26
27
28
$$p_v^k(t_r) = \begin{cases} \dfrac{[\tau_{vr}(t)]^\alpha}{\sum_{l \in ant[k].avlVtx} [\tau_{lr}(t)]^\alpha}, & \text{if } \forall l \in ant[k].avlVtx; \\ 0, & \text{if } \forall l \notin ant[k].avlVtx. \end{cases}$$

29 $antTreeCode[k][r] = v$
30 $antDeg[k][v] = antDeg[k][v] + 1$
31 **if** $antDeg[k][v] = U_d(v)$ **then**
32 $ant[k].avlVtx \leftarrow ant[k].avlVtx - \{v\}$
33 **for** k = 1 to $mAnts$ **do**
34 $ant\_d\text{-}STCost[k] \leftarrow$ compute the d-ST^k cost from $antTreeCode[k]$ by using
35 tree decoding algorithm
36 **if** $ant\_d\text{-}STCost[k] < L^{gb}$ **then**
37 $L^{gb} \leftarrow ant\_d\text{-}STCost[k]$
38 $d\text{-}ST^{gb}Code \leftarrow antTreeCode[k]$
39 $d\text{-}ST^{gb}Code \leftarrow$ Local search by using exchange mutation($d\text{-}ST^{gb}Code$, L^{gb}) //Fig. 8
40 The pheromone trails are updated:
41
42
43
$$\tau_{vr}(t+1) = (1 - \rho)\ \tau_{vr}(t) + \sum_{k=1}^{mAnts} \Delta\tau_{vr}^k,$$

44 where
45
46
47
$$\Delta\tau_{vr}^k = \begin{cases} Q/L^{gb}; \\ 0, \text{ otherwise.} \end{cases} \text{ as global update only.}$$

48 where L^{gb} is the total weight cost of decoded $d\text{-}ST^{gb}Code$ by using tree decoding
49 algorithms to obtain its d-ST^{gb} cost.
50 **end while**
51 **end procedure**

Figure 7. The pseudocode of the proposed ACO approach for d-MST problem. Both tree codings can be applied using this pseudocode

Back to the last step in an iteration of ACO on line 40 of Fig. 7 is the pheromone update. Only global pheromone update procedure is applied. The global update pheromone procedure decreases the value of the pheromone trails on τ_{vr} by a constant factor ρ and at the same time also deposit pheromone of an amount Q/L^{gb}. The v and r of τ_{vr} is corresponding to be the desirability of assigning vertex v in d-ST^{gb} code of length $|V|$-2 at array index r where $v \in [0, |V|\text{-}1]$ and $r \in [0, |V|\text{-}3]$. Q is a positive integer and L^{gb} is the total weight cost of decoded d-ST^{gb} tree code of the current iteration by using tree decoding algorithm to obtain its d-ST^{gb} cost.

```
1  procedure Local search by using exchange mutation(d-ST^sbCode, L^sb)
2    mutatedCode ← d-ST^sbCode //tree code of length |V|-2
3    indexFirst ← random[0, |V|-3]
4    do {
5      indexSecond ← random[0, |V|-3]
6    } while (indexFirst = indexSecond)
7    tempInteger ← mutatedCode[indexSecond]
8    mutatedCode[indexSecond] ← mutatedCode[indexFirst]
9    mutatedCode[indexFirst] ← tempInteger
10   if (|V|%2 = 0) then //if |V| is an even integer
11     numberOfTimes ← |V|/2
12   else
13     numberOfTimes ← (|V|+1)/2
14   count ← 0
15   do {
16     count ← count + 1
17     mutatedCodeCost ← compute the mutated tree code length from its
18                       mutatedCode by using tree decoding algorithm
19     if (mutatedCodeCost < L^sb) then
20       L^sb = mutatedCodeCost
21       return mutatedCode
22     else
23       indexFirst ← random[0, |V|-3]
24       do {
25         indexSecond ← random[0, |V|-3]
26       } while (indexFirst = indexSecond)
27       tempInteger ← mutatedCode[indexSecond]
28       mutatedCode[indexSecond] ← mutatedCode[indexFirst]
29       mutatedCode[indexFirst] ← tempInteger
30   } while (count < numberOfTimes)
31 return d-ST^sbCode
```

Figure 8. The pseudocode of the local search procedure by using exchange mutation

5. An ACO algorithm using Prüfer code and Blob code tree codings for lu-dMST problem

Four modifications have been made to the algorithm mentioned in section 4 to solve another variant of the d-MST problem with both lower and upper bound constraints on each vertex. The pseudocode of the proposed ACO approach for lu-dMST problem is given in Fig. 9. The Prüfer coding and Blob coding can be applied using this pseudocode. Again, two separate experiments are conducted. The first experiment is using the Prüfer coding and the second experiment is using the Blob coding. The use of tree codings such as Prüfer and Blob codes have made it easier to solve lu-dMST problem because the degree of the spanning tree is equal to one more of the number of times label of vertices appears in the Prüfer or Blob codes. It is also easy to determine if both the lower and upper bound constraints on each vertex are satisfied.

The first modification is to add new parameter *ant*[*k*].*lwrBndList* for each ant. The *ant*[*k*].*lwrBndList* parameter is the ant *k* lower bound list where $k \in [1, mAnts]$. The intention is that each ant will populate the vertices from *ant*[*k*].*lwrBndList* into *antTreeCode*[*k*] before selecting vertices from *ant*[*k*].*avlVtx*. This *ant*[*k*].*lwrBndList* parameter is needed for the ants to meet their lower bound degree constraint requirement. Each ant initialises their *ant*[*k*].*lwrBndList* as *ant*[*k*].*lwrBndList* ← *ant*[*k*].*lwrBndList* ∪ {*v*} if $L_d(v) > 1$ for each *v* in *V*. Because the $U_d(v)$ can be vary from vertex to vertex and be equal to one, the ant *k* also need to initialises their *ant*[*k*].*avlVtx* as *ant*[*k*].*avlVtx* ← *ant*[*k*].*avlVtx* ∪ {*v*} if $U_d(v) \neq 1$ for each *v* in *V*.

The second modification (line 4 of Fig. 9) is that the *d-PrimCode* is used to initialise the pheromone trails instead of being used as the starting solution for d-ST^{gb} code as in Fig. 7. The reason for this is most of the time, the d-Prim algorithm generates spanning tree that does not satisfy the lower bound degree constraint requirement for lu-dMST problem. The degree constraint for d-Prim is set to the maximum value of $U_d(i)$ where $i \in V$. The d-Prim d-ST is encoded to *d-PrimCode* by using tree encoding algorithm.

The third modification (lines 25 to 45 of Fig. 9) is the ants' tree code solution construction process to obtain their *antTreeCode*[*k*]. According to probability on line 35 of Fig. 9, the ant *k* will select a vertex *v* from *ant*[*k*].*lwrBndList* if *ant*[*k*].*lwrBndList* ≠ { } before the ant *k* can select a vertex *v* from *ant*[*k*].*avlVtx* for their *antTreeCode*[*k*]. The reason for this is to do away with repair function. If the repair option is used extensively it may be computationally expensive to repair infeasible ants' tree code solutions instead of the computation time could be better used for the ants to explore a better solution. A particular vertex *v* will be removed from *ant*[*k*].*lwrBndList* if (*antDeg*[*k*][*v*] = $L_d(v)$) and at the same time the vertex *v* will also be removed from *ant*[*k*].*avlVtx* if (*antDeg*[*k*][*v*] = $U_d(v)$). This is to ensure that both the lower and upper bound degree constraints during the ants' solutions construction process are adhered to. The objective function returns the cost of the lower and upper degree-constrained spanning tree (lu-dST). After every ant has completed their *antTreeCode*[*k*] of length $|V|$-2, then the best *antTreeCode*[*k*] will become the *lu-dST^{gb}Code*. The cost of the best *antTreeCode*[*k*] is determined by using tree decoding algorithms. Then, the same local search procedure by using exchange mutation as for d-MST problem is applied. This local search procedure is already given in Fig. 8.

The final modification is to add an extra pheromone update. The pheromone trails τ_{vr} are updated by using the *v* and *r* of *d-PrimCode* as follows:

$$\tau_{vr}(t+1) = (1 - \rho)\,\tau_{vr}(t) + Q/d\text{-}PrimCost. \quad (7)$$

where the d-Prim degree constraint is set randomly between two and the maximum value of $U_d(i)$ where $i \in V$. The d-Prim d-ST is encoded to *d-PrimCode* by using tree encoding algorithm. This *d-PrimCode* can be differed from the *d-PrimCode* as on line 4 of Fig. 9. This additional pheromone update idea is to enable the ants to consider others possible vertex *v* for their tree code solutions.

1	**procedure** ACO for lu-dMST
2	Set parameters.
3	Set Ls^{gb} to the maximum real number.
4	The pheromone trails τ_{vr} are initialised by using *v* and *r* of *d-PrimCode* as follows:
5	$\tau_{vr}(t+1) = (1 - \rho)\,\tau_{vr}(t) + Q/d\text{-}PrimCost$

where d-Prim degree constraint is set to the maximum value of $U_d(i)$ where $i \in V$. The d-Prim d-ST is encoded to *d-PrimCode* by using tree encoding algorithm.

while *termination_cond* = *false* **do**
  for k = 1 to *mAnts* **do**
    for v = 0 to $|V|$-1 **do**
      $antDeg[k][v] = 1$ //each ant k spanning tree vertices initial degree is set to 1
      if $L_d(v) > 1$ **then**
        $ant[k].lwrBndList \leftarrow ant[k].lwrBndList \cup \{v\}$
      if $U_d(v) \neq 1$ **then**
        $ant[k].avlVtx \leftarrow ant[k].avlVtx \cup \{v\}$
  for k = 1 to *mAnts* **do**
    $v \leftarrow$ select from $ant[k].avlVtx$ randomly
    $antTreeCode[k][0] = v$
    $antDeg[k][v] = antDeg[k][v] + 1$
    if $antDeg[k][v] = L_d(v)$ **then**
      $ant[k].lwrBndList \leftarrow ant[k].lwrBndList - \{v\}$
    if $antDeg[k][v] = U_d(v)$ **then**
      $ant[k].avlVtx \leftarrow ant[k].avlVtx - \{v\}$
  $r \leftarrow 0$
  while ($r < |V|$-2) **do**
    $r \leftarrow r + 1$
    for k = 1 to *mAnts* **do**
      The ant k tree code of array index r of iteration t will select a vertex v from the spanning tree vertices, according to probability:

$$p_v^k(t_r) = \left\{ \left\{ \begin{array}{ll} \dfrac{[\tau_{vr}(t)]^\alpha}{\sum_{l \in ant[k].lwrBndList} [\tau_{lr}(t)]^\alpha}, & \text{if } \forall l \in ant[k].lwrBndList; \\ 0, & \text{if } \forall l \notin ant[k].lwrBndList. \end{array} \right\}, \text{if } ant[k].lwrBndList \neq \{\}; \right.$$
$$\left. \left\{ \begin{array}{ll} \dfrac{[\tau_{vr}(t)]^\alpha}{\sum_{l \in ant[k].avlVtx} [\tau_{lr}(t)]^\alpha}, & \text{if } \forall l \in ant[k].avlVtx; \\ 0, & \text{if } \forall l \notin ant[k].avlVtx. \end{array} \right\}, \text{otherwise.} \right.$$

      $antTreeCode[k][r] = v$
      $antDeg[k][v] = antDeg[k][v] + 1$
      if $antDeg[k][v] = L_d(v)$ **then**
        $ant[k].lwrBndList \leftarrow ant[k].lwrBndList - \{v\}$
      if $antDeg[k][v] = U_d(v)$ **then**
        $ant[k].avlVtx \leftarrow ant[k].avlVtx - \{v\}$
  for k = 1 to *mAnts* **do**
    $ant\_lu\text{-}dSTCost[k] \leftarrow$ compute the lu-dST^k cost from its $antTreeCode[k]$ by using tree decoding algorithm
    if $ant\_lu\text{-}dSTCost[k] < L_{sb}$ **then**
      $L_{sb} \leftarrow ant\_lu\text{-}dSTCost[k]$
      $lu\text{-}dST_{gb}Code \leftarrow antTreeCode[k]$

lu-dST^{gb}Code ← Local search by using exchange mutation(*lu-dST^{gb}Code*, L^{gb}) //Fig. 8
Then, the pheromone trails are updated as global update as follows:

$$\tau_{vr}(t+1) = (1 - \rho)\ \tau_{vr}(t) + \sum_{k=1}^{mAnts} \Delta\tau_{vr}^{k},$$

where

$$\Delta\tau_{vr}^{k} = \left\{ \begin{array}{l} Q/L^{gb}; \\ 0, \text{ otherwise.} \end{array} \right\} \text{as global update.}$$

where L^{gb} is the total weight cost of decoded *lu-dST^{gb}Code* by using tree decoding algorithms to obtain its lu-dST^{gb} cost.
The pheromone trails τ_{vr} are updated by using v and r of *d-PrimCode* as follows:
$\tau_{vr}(t+1) = (1 - \rho)\tau_{vr}(t) + Q/d\text{-}PrimCost$
where the d-Prim degree constraint is set randomly between two and the maximum of $U_d(i)$ where $i \in V$. The d-Prim d-ST is encoded to *d-PrimCode* by using tree encoding algorithm.
end while
end procedure

Figure 9. The pseudocode of the proposed ACO approach for lu-dMST problem. Both tree codings can be applied using this pseudocode

6. Performance comparisons of Prüfer ACO and Blob ACO on structured hard (SHRD) graph data set for d-MST problem

The Prüfer-coded ACO and Blob-coded ACO are tested on structured hard (SHRD) graphs as used in (Raidl, 2000; Mohan et al., 2001; Bui & Zrncic, 2006) for the d-MST problem. The SHRD graphs are constructed by using non-Euclidean distance as follows. The first vertex is connected to all other vertices by an edge of length l; the second vertex is connected to all vertices bar the first by an edge of length $2l$ and so on. Then SHRD is randomised slightly by adding a uniformly distributed perturbation between 1 and 18 where l = 20. The details to generate a SHRD graph is given in Fig. 10. This reduces the likelihood of a large number of optimal solutions existing but doesn't change the underlying complexity of the problem. These are difficult to solve optimally compared to other data sets such as Euclidean data sets of degree 3 or more (Mohan et al., 2001). The MST for SHRD is a star graph where one vertex has degree $|V|$-1 and the all other vertices have degree 1. The SHRD graphs are complete graphs with undirected non-negative weighted edges.
The parameter ρ for Prüfer-coded ACO and Blob-coded ACO is tuned from 0.0 to 0.9. For each ρ, average solution costs over 50 independent runs are recorded. Each run terminates after 274 ($50 * \sqrt{|V|}$) iterations. The setting that produced the lowest average solution cost will be the Prüfer-coded ACO and Blob-coded ACO parameter value used for SHRD data set. Table 2 shows the parameter tuning results for Prüfer-coded ACO and Blob-coded ACO approaches on SHRD data set. The lowest ρ values for Prüfer-coded ACO and Blob-coded ACO from Table 2 are in bold print. Separate parameter values are used for Prüfer-coded ACO and Blob-coded ACO on the SHRD problem instances. The parameter value of ρ = 0.1 is chosen for Prüfer-coded ACO while value of ρ = 0.9 is chosen for Blob-coded ACO. There

is so much difference between Prüfer-coded ACO and Blob-coded ACO parameter value of ρ. One of the probable reasons is the Blob code exhibits higher locality under mutation of one symbol compares to Prüfer code. On average only about two edges for a spanning tree is changed after changing one symbol in a Blob code to be decoded by the Blob decoding algorithm (Julstrom, 2001). Table 3 shows the values of the ACO parameters. All results are obtained using a PC with Pentium 4 processor with 512 megabytes of memory, running at 3.0 GHz under Windows XP Professional.

```
1   procedure generateSHRDgraph
2      Let total number of vertices as |V|
3      Let graph edges as edge[|V|][|V|]
4      for i = 0 to |V|-1 do
5         for j = 0 to i do
6            if i = j then
7               edge[i][j] = 1000000000.000000
8            else
9               edge[i][j] = 20*j + random[1, 18]
10              edge[j][i] = edge[i][j]
11     // Print lower left SHRD triangular graph matrix only
12     for i = 1 to |V|-1 do
13        for j = 0 to i-1 do
14           Print edge[i][j] and " ".
15        Print newline.
16  end procedure
```

Figure 10. The pseudocode to generate a SHRD graph

	Prüfer-coded ACO	**Blob-coded ACO**
ρ = 0.0	1554.74	1532.66
0.1	**1551.96**	1533.74
0.2	1554.14	1532.22
0.3	1556.68	1532.04
0.4	1553.48	1533.66
0.5	1554.92	1533.66
0.6	1553.94	1535.20
0.7	1553.90	1533.52
0.8	1552.70	1535.26
0.9	1552.00	**1530.34**

Table 2. Parameter ρ tuning for Prüfer-coded ACO and Blob-coded ACO average results, problem shrd305, d = 5, $|V|$ = 30, number of iterations = 50 * $\sqrt{|V|}$ = 274, number of runs = 50

	Prüfer-coded ACO	Blob-coded ACO
ρ	0.1	0.9
mAnts	$\|V\|$	$\|V\|$
Q	1.0	1.0
α	1	1
SHRD maxEdgeCost	$20*\|V\|$	$20*\|V\|$
τ_0	$\|V\|^2*20*\|V\|$	$\|V\|^2*20*\|V\|$
iterations	$50*\sqrt{\|V\|}$	$50*\sqrt{\|V\|}$

Table 3. The ACO parameters and their values for artificial ant *k* using Prüfer code and Blob code tree codings on SHRD problem instances

Problem	Prüfer ACO avg.	Prüfer ACO best	Prüfer ACO time	Blob ACO avg.	Blob ACO best	Blob ACO time	Enhanced k-ACO avg.	Enhanced k-ACO best	Enhanced k-ACO time
SHRD153	16.94	18.89	15	17.95	19.84	21	20.26	21.19	120
SHRD154	9.94	12.01	15	12.25	14.37	21	12.29	15.35	120
SHRD155	8.04	9.60	15	7.87	9.60	21	8.95	9.60	120
SHRD203	8.74	10.74	38	10.22	11.39	52	11.72	12.12	180
SHRD204	6.45	7.79	38	7.05	8.47	52	9.22	9.48	180
SHRD205	5.70	7.15	38	6.46	8.03	52	8.17	8.47	180
SHRD253	16.26	18.67	87	17.82	19.13	118	19.81	20.40	360
SHRD254	3.36	4.82	87	4.40	5.55	118	6.41	6.72	360
SHRD255	5.45	7.19	87	7.39	8.29	118	8.91	9.02	360
SHRD303	9.14	10.71	145	9.88	11.52	197	12.30	12.46	660
SHRD304	8.20	10.04	145	9.63	11.06	197	11.61	11.80	660
SHRD305	3.59	5.40	145	4.68	5.96	197	6.37	6.58	660
Total Average:	8.48	10.25	855	9.63	11.10	1164	11.34	11.93	3960

Table 4. Average and best results (quality gains over d-Prim in %), and total times (in seconds) on SHRD problem instances. Label SHRD153 means SHRD graph 15-vertex with degree constraint, *d*=3 and so on

Table 4 summarises the results of Prüfer-coded ACO and Blob-coded ACO on SHRD data set. The Prüfer-coded ACO and Blob-coded ACO were run 50 independent times on each problem instance. Each run is terminated after $50 * \sqrt{|V|}$ iterations. The number of vertices are in the range 15, 20, 25, and 30. The maximum degree was set to 3, 4 and 5. The results for the enhanced kruskal-ACO are adopted from (Bau et al., 2007). At this time, the enhanced kruskal-ACO is used as a performance benchmark. It is one of the best approaches for the d-MST problem on the SHRD graphs (Bau et al., 2007). Besides average gains, the gains of the best run and total times (in seconds) that are required for 50 runs are reported in Table 4.

The total times in seconds for 50 runs is recorded so that the time required between ACO without tree coding and ACO using tree coding can be compared. The enhanced kruskal-ACO is referred to as Enhanced k-ACO but the Prüfer-coded ACO and Blob-coded ACO are referred to as Prüfer ACO and Blob ACO. Between Prüfer ACO and Blob ACO, the highest average gains are underlined.

It can be concluded that Enhanced k-ACO has higher total average results compared to Blob ACO. In turn, Blob ACO has higher total average results compared to Prüfer ACO. Between Prüfer ACO and Blob ACO, Blob ACO almost always identifies trees of higher gains except on a SHRD155 d=5 problem instance. Between Blob ACO and Enhanced k-ACO, Blob ACO achieves results very close to Enhanced k-ACO. On all the problem instances, the maximum average result difference between them is only by 2.42 on a SHRD303 d=3. When all three ACO approaches are compared, the Enhanced k-ACO attains the highest total average compared to the Prufer ACO and Blob ACO. The reason for this is probably due to the fact that Enhanced k-ACO uses visibility measure during the ants' solution construction. However on all problem instances, the Prüfer ACO and Blob ACO performed faster in terms of computation time compared to the Enhanced k-ACO. The Prüfer ACO requires only about 22% and Blob ACO requires only about 29% as much time as does the Enhanced k-ACO.

7. Performance comparisons of Prüfer ACO and Blob ACO on structured hard (SHRD) graph data set for lu-dMST problem

Four networks of varying sizes based on SHRD graphs are generated. The number of vertices are 20, 40, 60, and 80, similar to those used in (Chou et al., 2001). The SHRD 20-vertex problem instance set is labelled as SHRD20, the SHRD 40-vertex problem instance set is labelled as SHRD40 and so on. For each vertex, an integer from a range of one to four is randomly generated for the lower bound degree constraint, $L_d(i)$, and one to eight is randomly generated for the upper bound degree constraint, $U_d(i)$ where $i \in V$. This means that the maximum value for the upper bound degree constraint is eight. The minimum value of $L_d(i)$ and $U_d(i)$ is always equal to 1, and $U_d(i)$ is always greater than or equal to $L_d(i)$. In order to ensure that the network forms at least one feasible solution, the sum for each vertex of lower bound degree constraint is set between $|V|$ and $2(|V|\text{-}1)$ as follows:

$$|V| \leq \sum_{i=0}^{|V|-1} L_d(i) \leq 2(|V|-1). \tag{8}$$

And, the sum for each vertex of upper bound degree constraint is set between $2(|V|\text{-}1)$ and $|V|(|V|\text{-}1)$ as follows:

$$2(|V|-1) \leq \sum_{i=0}^{|V|-1} U_d(i) \leq |V|(|V|-1). \tag{9}$$

The reason for this is that a spanning tree always consists of $|V|$-1 edges, an edge consists of exactly two distinct vertices, and the total number of the degrees of an edge is two. Therefore, the sum over the degrees $deg(i)$ of a spanning tree on each vertex i in V as given in (Gross & Yellen, 2006) can be calculated as follows:

$$\sum_{i=0}^{|V|-1} \deg(i) = 2(|V|-1) . \quad (10)$$

Table 5 summarises the results of these Prüfer-coded ACO and Blob-coded ACO approaches on SHRD graph. The Prüfer-coded ACO and Blob-coded ACO approaches were each run 50 independent times on each problem instance. Each run is terminated after $50 * \sqrt{|V|}$ iterations. The numbers of vertices are in the range 20, 40, 60, and 80. For each $i \in V$, the $1 \leq L_d(i) \leq 4$, $1 \leq U_d(i) \leq 8$, and $U_d(i) \geq L_d(i)$. Besides average solution cost, the solution cost of the best run and total times (in seconds) required for 50 independent runs are reported in Table 5. The solution cost is used here for performance comparison rather than the quality gain. The Prüfer-coded ACO and Blob-coded ACO approaches are referred to as Prüfer ACO and Blob ACO. The parameter values of Prüfer ACO and Blob ACO for lu-dMST problem are the same as the parameter values of Prüfer ACO and Blob ACO for d-MST problem.

As shown in Table 5, it can be concluded that Blob ACO always has the better results compared to Prüfer ACO. The Blob ACO always identifies trees of lower solution cost for all the problem instances in the SHRD graphs. The overall effectiveness of the Blob ACO compared to Prüfer ACO is probably due to the fact that it uses better tree coding scheme. Prüfer code is a poor representation of spanning trees for EA (Gottlieb et al., 2001; Julstrom, 2001). Small changes in Prüfer code often cause large changes in the spanning trees they represent. However on all problem instances, the Prüfer ACO requires lesser time than the Blob ACO. This does not bring to a conclusion that Blob tree coding always requires more time compared to Prüfer tree coding. In a recent study of the Blob code spanning tree representations, Paulden and Smith (2006) have described linear-time encoding and decoding algorithms for the Blob code, which supersede the usual quadratic-time algorithms.

Problem	Prüfer ACO avg.	Prüfer ACO best	Prüfer ACO Time (secs)	Blob ACO avg.	Blob ACO best	Blob ACO Time (secs)
SHRD20	1429.28	1304	33	1391.56	1286	50
SHRD40	5573.04	4941	330	5034.32	4673	458
SHRD60	12167.48	11003	1345	11448.68	10625	1715
SHRD80	16683.12	14839	3483	15013.24	13844	4610
Total Average:	35852.92	32087	5191	32887.80	30428	6833

Table 5. Average solution cost on SHRD problem instances with both lower and upper bound degree constraints. Label SHRD20 means SHRD graph 20-vertex and so on

8. Conclusion

The design and implementation of Blob-coded ACO and Prüfer-coded ACO for d-MST and lu-dMST problems have been presented. This ACO approaches is different because it constructs the encoded of the solution and can speed up computation time. Performance studies have revealed that Blob-coded ACO is almost always better than Prüfer-coded ACO for both types of problems for the SHRD graphs. However for the d-MST problem, Blob-coded ACO does not perform better than the enhanced kruskal-ACO approach in any single

problem instance for SHRD graphs. Finally, the Blob code may be a useful coding of spanning trees for ants' solution construction in ACO algorithms for the d-MST and ludMST problems in terms of computation time. There may be other codings of spanning trees even more appropriate for ants' solution construction such as Happy code or Dandellion code as mentioned by Picciotto (1999) in his PhD thesis.

9. References

Bau, Yoon Teck; Ho, Chin Kuan, & Ewe, Hong Tat (2007). Ant Colony Optimization Approaches to the Degree-constrained Minimum Spanning Tree Problem. *Journal of Information Science and Engineering* (in press)

Bui, T. N. & Zrncic, C. M. (2006). An Ant-Based Algorithm for Finding Degree-Constrained Minimum Spanning Tree. *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation,* 11-18

Cayley (1889). A theorem on trees. *Quarterly Journal of Mathematics*, 23, 376-378

Dorigo, M. & Stützle T. (2004). *Ant Colony Optimization.* A Bradford Book, The MIT Press, Cambridge, MA, London, England

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

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

Gottlieb, J.; Julstrom, B. A., Raidl, G. R. & Rothlauf, F. (2001). Prüfer numbers: A poor representation of spanning trees for evolutionary search. In: *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-2001,* Lee, Spector, Goodman, E. D., Annie, Wu, Langdon, W. B., Hans-Michael, V., Mitsuo, Gen, Sandip, Sen, Dorigo, M., Pezeshk, S., Garzon, M. H. & Burke, E. (Eds.), (343-350), Morgan Kaufmann, San Francisco, California, USA

Gross, J. L. & Yellen, J. (2006). *Graph Theory and its Applications 2nd ed.,* CRC Press, Boca Raton, London, New York, Washington, D.C.

Hsinghua Chou; G. Premkumar & Chao-Hsien Chu (2001). Genetic Algorithms for Communications Networks Design – An Empirical Study of the Factors that Influence Performance. *IEEE Transactions on Evolutionary Computation*, Vol. 5, No. 3, 236-249

Julstrom, B. A. (2001). The Blob Code: A better string coding of spanning trees for evolutionary search. In: *2001 Genetic and Evolutionary Computation Conference Workshop Program*, Annie, S. Wu (Ed.), (256-261), San Francisco, CA

Knowles, J. & Corne, D. (2000). A new evolutionary approach to the degree-constrained minimum spanning tree problem. *IEEE Transactions on Evolutionary Computation,* 4(2), 125-134

Michalewicz Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs 3rd Rev. and Extended ed*., Springer Verlag

Mohan, Krishnamoorthy; Andreas, T. Ernst & Yazid, M. Sharaiha (2001). Comparison of Algorithms for the Degree-constrained Minimum Spanning Tree. *Journal of Heuristics,* 7(6), 587-611

Narula, S. C. & Ho, C. A. (1980). Degree-constrained minimum spanning tree. *Computer Operation Research,* 7(4), 239-249

Paulden, T. & Smith, D. K. (2006). Recent advances in the study of the Dandelion code, Happy code, and Blob code spanning tree representations, *In Proceeding IEEE Congress on Evolutionary Computation,* 2111- 2118

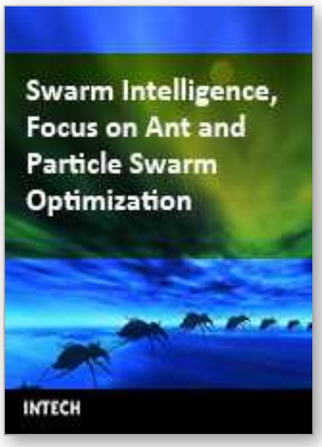
Picciotto, S. (1999). *How to encode a tree,* Ph.D. Thesis, University of California, San Diego

Prim, R. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal,* 36, 1389-1401

Prüfer, H. (1918). Neuer Beweis eines Satzes über Permutationen [New proof of a counting labelled tree sequence over permutations]. *Architecture Mathematics Physics,* 27, 742-744

Raidl, G. R. (2000). An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 1, 104-111

Sipser, M. (2006). I*ntroduction to the Theory of Computation 2nd ed*., Course Technology



Swarm Intelligence, Focus on Ant and Particle Swarm Optimization

Edited by FelixT.S.Chan and Manoj KumarTiwari

ISBN 978-3-902613-09-7
Hard cover, 532 pages
Publisher I-Tech Education and Publishing
Published online 01, December, 2007
Published in print edition December, 2007

In the era globalisation the emerging technologies are governing engineering industries to a multifaceted state. The escalating complexity has demanded researchers to find the possible ways of easing the solution of the problems. This has motivated the researchers to grasp ideas from the nature and implant it in the engineering sciences. This way of thinking led to emergence of many biologically inspired algorithms that have proven to be efficient in handling the computationally complex problems with competence such as Genetic Algorithm (GA), Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), etc. Motivated by the capability of the biologically inspired algorithms the present book on "Swarm Intelligence: Focus on Ant and Particle Swarm Optimization" aims to present recent developments and applications concerning optimization with swarm intelligence techniques. The papers selected for this book comprise a cross-section of topics that reflect a variety of perspectives and disciplinary backgrounds. In addition to the introduction of new concepts of swarm intelligence, this book also presented some selected representative case studies covering power plant maintenance scheduling; geotechnical engineering; design and machining tolerances; layout problems; manufacturing process plan; job-shop scheduling; structural design; environmental dispatching problems; wireless communication; water distribution systems; multi-plant supply chain; fault diagnosis of airplane engines; and process scheduling. I believe these 27 chapters presented in this book adequately reflect these topics.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Yoon-Teck Bau, Chin-Kuan Ho and Hong-Tat Ewe (2007). A New Ant Colony Optimization Approach for the Degree-Constrained Minimum Spanning Tree Problem Using Pruefer and Blob Codes Tree Coding, Swarm Intelligence, Focus on Ant and Particle Swarm Optimization, FelixT.S.Chan and Manoj KumarTiwari (Ed.), ISBN: 978-3-902613-09-7, InTech, Available from: http://www.intechopen.com/books/swarm_intelligence_focus_on_ant_and_particle_swarm_optimization/a_new_ant_colony_optimization_approach_for_the_degree-constrained_minimum_spanning_tree_problem_usin

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen

IntechOpen