

# Parallel Algorithms for the Degree-Constrained Minimum Spanning Tree Problem Using Nearest-Neighbor Chains and the Heap-Traversal Technique

Li-Jen Mao\* and Sheau-Dong Lang\*\*

\*Department of Information Management, Yuda Institute of Business Technology

Miaoli, Taiwan, R.O.C. Email: [mao@msl.ydu.edu.tw](mailto:mao@msl.ydu.edu.tw)

\*\*School of Electrical Engineering and Computer Science, University of Central Florida

Orlando, FL 32816-2362, U.S.A. Email: [lang@cs.ucf.edu](mailto:lang@cs.ucf.edu)

## Abstract

*The Degree-Constrained Minimum Spanning Tree ( $d$ -MST) problem attempts to find a minimum spanning tree with an added constraint that no nodes in the tree have a degree larger than a specified integer  $d$ . It is known that computing the  $d$ -MST is NP-hard for every  $d$  in the range  $2 \leq d \leq (n - 2)$ , where  $n$  denotes the total number of nodes. Several approximate algorithms (heuristics) have been proposed in the literature. We have previously proposed three approximate algorithms, IR, TC-RNN, and TC-NNC, for solving the  $d$ -MST problem, the last two (TC-RNN and TC-NNC) take advantages of nearest neighbors and their properties. Our experimental results showed that both the TC-RNN and TC-NNC algorithms consistently produce spanning trees with a smaller weight (better quality-of-solution) than that of IR, but using slightly longer execution time. In this paper, we propose a new heap traversal technique that further improves the time efficiency of TC-RNN and TC-NNC. Our experiments using randomly generated, weighted graphs as inputs show that the TC-NNC algorithm outperforms the other two approximate algorithms in terms of the execution time and quality-of-solution.*

**Keywords:** Parallel approximate algorithm, degree-constrained minimum spanning tree, nearest neighbor chain, heap traversal.

## 1. Introduction

Given a connected, edge-weighted, undirected graph  $G$  and a positive integer  $d$ , the Degree-Constrained MST ( $d$ -MST) problem is to find a spanning tree with the smallest weight among all possible spanning trees of  $G$  which contain no nodes of degree greater than  $d$ . This problem is NP-hard, because the *Hamiltonian Path* problem (Problem ND1 in Garey and Johnson [4]), which

is NP-complete, is a special case of  $d$ -MST with  $d = 2$  and all edge weights identical.

The *Degree-Constrained* MST problem was first studied by Deo and Hakimi [3]. Since computing the  $d$ -MST is NP-hard for every  $d$  in the range  $2 \leq d \leq (n - 2)$ , several approximate algorithms have been proposed in the literature [2, 6, 8, 13, 15, 16]. Ravi, Marathe, Ravi, Rosenkrantz, and Hunt [16] showed that finding approximate solutions to  $d$ -MST within a constant factor of the weight of an optimal tree is also NP-hard. For the  $d$ -MST problem on general weighted graphs, several heuristic solutions have been proposed which, in general, have no guaranteed bounds on the quality of the solutions. Yamamoto [22] proposed an approximate solution by computing the minimum cycle basis of two matroids associated with the given graph. Narula and Ho [14] formulated a branch-and-bound procedure based on Held and Karp's [4, 7] method for the Traveling Salesman problem. Gavish [5] used subgradient optimization for deriving Lagrangian-based lower bounds. Savelsbergh and Volegnant [17] used a branch-and-bound method based on an edge-elimination approach that was previously applied to the Traveling Salesman problem [18]. Volegnant [19] gave a branch-and-bound procedure based on Lagrangian Relaxation and edge exchanges. Krishnamoorthy, Craig, and Palaniswami [9] recently explored several heuristic solutions to the problem using neural networks, simulated annealing, greedy algorithms, and greedy randomized algorithms. Most of these methods, with the exception of the greedy heuristics of Krishnamoorthy et al., seem very time-consuming and become ineffective for graphs of more than a few hundred nodes.

We have recently proposed three complementary approaches to solving the  $d$ -MST problem, and presented a comparison of the parallel implementation of three approximate algorithms, IR, TC-RNN and TC-NNC [11, 12]. The IR (Iterative Refinement) algorithm applies Prim's algorithm with a penalty heuristic which

successively recalculates the MST and penalizes those edges incident to nodes violating the specified degree-bound. It was shown [10] that the IR algorithm is expected to converge to a feasible solution for the  $d$ -MST problem when  $d \geq 3$ . The TC-RNN (Tree-Construction with Reciprocal Nearest Neighbors) algorithm was obtained by modifying Sollin's MST algorithm, incorporating a check for the degree constraint in every step of the tree-construction. The TC-NNC (Tree-Construction with Nearest Neighbor chains) algorithm improves upon the two previous algorithms in both speed and quality of solution. All the three algorithms lend themselves naturally to parallel implementation, especially on massively parallel SIMD machines.

In this paper, we present a new technique that further improves the speed of algorithm TC-NNC. The remainder of the paper is organized as follows. Section 2 briefly reviews the three approximate algorithms IR, TC-RNN, and TC-NNC. The new *heap traversal* technique is described in Sections 3, along with its application to the TC-NNC algorithm and implementation details. Section 4 presents the experimental results comparing the three approximate algorithms. Section 5 concludes the paper and points out directions for further research. Throughout the paper, graphs are assumed to be complete and undirected, with the edge weights represented by a weight-matrix.

## 2. Three Approximate $d$ -MST Algorithms

In this Section, we briefly describe the approximate algorithms IR, TC-RNN and TC-NNC that we designed earlier for the  $d$ -MST problem.

### 2.1. The IR Algorithm

The IR algorithm uses an iterative refinement process consisting of two phases: (1) Computing an MST; and (2) Penalizing edges. First, an MST using the initial weight-matrix is computed. Then, in the penalty phase it increase the weights of those tree edges that are incident to the nodes with the degree exceeding the constraint  $d$ . The MST of the graph with the new weights is computed next. Note that those tree edges with an increased weight from the penalty phase are discouraged from appearing in the next spanning tree. Alternations of the penalty phase followed by the MST computation are repeated until a spanning tree is produced in which every node satisfies the degree bound. Details of the penalty function and the parallel implementation of the IR algorithm can be found in [2, 10, 11].

### 2.2. NN-chains and Reciprocal Nearest Neighbors

The main idea of following two tree-construction algorithms is based on the concept of *nearest neighbor chain* (NN-chain) and *reciprocal nearest neighbors*. For any given connected and edge-weighted graph, a *nearest neighbor chain* (NN-chain) is constructed as follows. Starting from an arbitrary node  $v_1$ , find its nearest neighbor, call it  $v_2 = NN(v_1)$ . Then, find the nearest neighbor of  $v_2$ , call it  $v_3 = NN(v_2)$ , etc. An NN-chain consists of this path of nearest neighbors  $(v_1, v_2, \dots)$ . Since the edge weights (distances between adjacent nodes) along the chain are always monotonically decreasing, an NN-chain must end in a pair of nodes which are nearest neighbors of each other, which are called *reciprocal nearest neighbors* (RNN).

### 2.3. The TC-RNN and TC-NNC Algorithms

Both TC-RNN algorithm and TC-NNC algorithm are based on Sollin's MST algorithm described as follows: First, we initialize a forest  $F$  consisting of all nodes as single-node trees. Then, the shortest edge incident to each tree is selected, and these shortest edges for all the trees are added to the forest  $F$ . This process continues until the forest contains  $(n - 1)$  edges, where  $n$  denotes the total number of nodes. A straightforward sequential implementation of Sollin's algorithm requires  $O(n^2 \log n)$  time, which has been improved to  $O(n^2 \log \log n)$  by Yao [21] using clever data structures.

In adapting Sollin's algorithm to solving the  $d$ -MST problem on a parallel machine, we assign each node to a separate processor, and these processors simultaneously find nearest neighboring trees to merge until a single tree remains. The only difference between algorithms TC-RNN and TC-NNC is the way by which multiple trees are merged. For the TC-RNN algorithm; in each iteration of the parallel loop, two trees assigned to two processors are merged if they are nearest neighbors of each other through a common shortest edge connecting the two trees, assuming the edge doesn't cause degree constraint violations. (The two trees are named *reciprocal nearest neighbors*). For the TC-NNC algorithm, in each iteration of the parallel loop, all trees along the nearest neighbor chain are merged at once, as long as the edges along the NN-chains don't cause any degree constraint violations.

In the following sub-sections we explain the overall strategy and the procedures that implemented the two algorithms on a parallel computer. Details of the parallel implementation can be found in [10, 11, 12].

**2.3.1. The Overall Strategy of Tree Construction.** We describe the overall strategy of the TC-NNC and TC-RNN implementations on a parallel computer, and point out the differences where they differ.

1. Each processor is in charge of one node throughout the algorithm's execution. Initially, each node is in a

tree by itself. In each iteration, a tree is merged with its nearest neighbor tree, resulting in a set of RNN pairs (for TC-RNN) or NN-chains (for TC-NNC).

2. Each processor in charge of a node maintains a min-heap of size  $n - 1$ , storing the indices of the adjacent nodes based on the associated edge weights. The index of the nearest neighbor node is at the top of the heap.
3. Exactly one node in each tree is designated as the *root-node* of that tree, and every node of the same tree points to the same *root-node* with the variable *root\_ID*. Thus, the variable *root\_ID* allows two nodes to determine if they belong to the same tree.
4. In each tree-merge operation, only one outgoing edge of a tree is allowed to link to a node in another tree. For the incoming edges, more than one edge are allowed to link to the nodes of a tree for TC-NNC algorithm, but only one edge is allowed for TC-RNN.
5. In each tree-merge operation, all nodes of the same tree perform the following tasks in parallel:
  - Vote for a single outgoing edge to be used to link to another tree.
  - Vote for the incoming edges that can be used to connect to a tree, which will form one RNN pair for TC-RNN, or form one or more NN-chains for TC-NNC.
  - Merge all trees of RNN-pairs or along the NN-chains and update the new root information.
  - Update the total number of trees to decide if the algorithm should terminate.

#### Algorithm Par\_TC-NNC / TC-RNN

1.  $No\_of\_roots = n$
2. **all** processors **do par**
3. make a MIN-heap out of  $n - 1$  edges
4. **while** ( $No\_of\_roots > 1$ ) **do**
5. construct the **NN-chains** (or **RNN pairs**) as follows:
  - (a) each tree finds the shortest outgoing edge (from MIN-heap of each node of the tree) that links to another tree, sends a message to that tree requesting a merge
  - (b) each tree votes for the incoming edge selected from Step (a)
  - (c) connect all winning edges of Step (b) to form **NN-chains** (or **RNN pairs**)
6. merge all trees along the **NN-chains** (or along every **RNN pair**) and update their roots to new roots
7. update  $No\_of\_roots$
8. **end while**
9. **end do par**

Notice unlike the TC-RNN algorithm which merges only the RNN pairs in each tree-merge operation, the TC-NNC algorithm merges all the trees along each NN-chain to improve the speed. We use the following procedures to ensure that trees are merged without creating cycles, and that a unique root is assigned to each merged tree without causing so-call *multi-root* conflict:

1. When one tree is merged to its nearest neighbor tree, the *root\_ID* of each of the nodes of the first tree is changed to the *root\_ID* of the second tree.
2. If two trees are nearest neighbors to each other (i.e. an *RNN* pair), the tree with the lower id will be the *root* of the new tree. For TC-NNC; since every NN-Chain has exactly one pair of *RNN* trees, only one root is assigned to the new tree.
3. To avoid possible cycles in forming an NN-chain, if one tree has more than one nearest neighbor with equal weights, the one with the lowest id is selected.

A high-level description of algorithm TC-NNC/TC-RNN is given in Figure 1. The TC-NNC/TC-RNN algorithm consists of two major tasks that are executed in parallel: the first task is to make a MIN-heap for each node (Step 3); the second task (Step 4 to Step 9) involves a sequence of tree-merge operations. We explain these tasks in more detail in the following sections.

**2.3.2. Make MIN-heaps in Parallel.** The first task of algorithm TC-NNC/TC-RNN is to make a MIN-heap (in parallel) for each node using the edge weights of the adjacent  $n - 1$  edges. After completing this task, the nearest neighbor (with the smallest edge weight) of each node is located at the top of the heap. We adapted the well-known heap procedures *heapify* as well as *adjust\_min* [1] and developed their parallel version for this task. It is well known that the time complexity of this Procedure is  $O(n)$  [1,20]. Figure 2 shows the parallel implementation of *par-heapify* and *par-adjust\_min*. In the procedures, an array of node indexes *HP* represent the edges that are connected to the node in operation, and array *W* stores the corresponding edge weights. In Procedure *Par-adjust\_min*, the node *index* represents an edge from the calling node which is compared to other edges and moved along the heap structure in a top-down fashion, and is adjusted to the proper position according to its edge weight. The Procedure *Par-heapify* creates the *MIN*-heap by repeatedly calling Procedure *Par-adjust* for  $\lfloor N/2 \rfloor$  times, which builds the heap from bottom up, note in the old version of algorithm TC-NNC, the *Par-adjust\_min* is also being called by another procedure (*Par-delete-min*) to choose an outgoing edge for tree-merge operation.

**2.3.3. Vote for an Outgoing Edge.** The first subtask of the tree-merge operation of the TC-NNC/TC-RNN

Figure 1. High-level description of TC-NNC and TC-RNN

algorithms is for each tree to choose a shortest outgoing edge to its nearest neighbor tree. This operation consists of the following steps:

1. Every node sets itself to be an active node if the degree limit is not reached; otherwise, if the node has already reached the degree limit, it sets itself to be an inactive node and does nothing further.
2. Every active node reads the id of its nearest neighbor node from the local *MINheap*, let it be *nid*. The node inactivates itself if its nearest neighbor *nid* has reached the degree limit; otherwise, let *nidWgt* denote the edge weight of the edge connected to *nid*.
3. Every active node uses the MasPar's parallel communication function *send\_with\_min()* to send the local variable *nidWgt* to its root node. After the function call, the minimum among these edge weights is stored at the root node.
4. Every active node reads the variable *nidWgt* from the remote root node, and compares it to the same variable stored locally. If a node finds both variables have the same value, it means this node is a winner of the competition. Let all such nodes be *win\_nodes*.
5. To avoid conflicting nodes with the same minimum *nidWgt* value, all *win\_nodes* use the MasPar's function *send\_with\_min()* to send their own node id to their root node. If more than one node in a tree have the same minimum *nidWgt* value, the node with the smallest node id will win. We name the winning node *win\_out*, and name the associated minimum *nidWgt* value *minWgt*.
6. Every node other than the *win\_out* nodes adjusts its local *MINheap*, so that the next nearest neighbor node will be available at the top of the *MINheap*.

After executing the above procedure, each tree produces exactly one outgoing edge to the associated *win\_out* node. The next task is to let *win\_out* node connect to its nearest neighbor node *nid*, with an edge weight equals *minWgt*. This is described in the next section.

**2.3.4. Vote for Incoming Edges.** After each tree has selected an outgoing edge with the smallest weight, the next subtask is for each tree to choose one (for TC-RNN) or more (for TC-NNC) incoming edges to form RNN pairs or NN-Chains, while conforming to the degree constraint. This subtask is described as follows:

1. Every tree *T* has a node *win\_out* that wishes to connect to its nearest neighbor node *nid* which belongs to the tree *NN(T)*, using an edge weight equal to *minWgt*. Before going any further, the node *win\_out* first checks the degree of node *nid* to make sure it has not reached the degree limit; otherwise, the node *win\_out* becomes inactive and does nothing in this iteration.

2. Each node *nid* sends the edge weight *nidWgt* to its destination tree *NN(T)* and competes with others to be the next shortest incoming edge to *NN(T)*.
3. Every tree chooses a shortest incoming edge received from other trees, and replies to the senders the winner which is named node *win-in*.
4. If node *win\_out* of tree *T* becomes the winner *win-in* of its destination tree *NN(T)*, tree *T* is allowed to be merged with tree *NN(T)*. Both node *win\_out* and the corresponding node *nid* increase their degree by one; node *win\_out* becomes inactive for this iteration.
5. If node *win\_out* of tree *T* failed to become the *win-in* of its destination tree, repeat step 1 through step 4 until the merge procedure is completed or until the nearest neighbor node has reached the degree limit.

**Procedure Par-adjust\_min(HP[], W[], index, N)**

**Input:** *N* // number of nodes  
*W[N]* // weight array  
*HP[N]* // heap array of size *N*  
*index* // index of item to be adjusted in the heap  
**plural variables:** *index, item, not\_found, pos;*

1. *pos* = 2\**index*;
2. *item* = *HP[index]*;
3. *not\_found* = true;
4. **while** ((*pos* < *N*) **and** (*not\_found*)) **do**
5.     **if** ((*pos* < *N*-1) **and** (*W*[*HP*[*pos*]] > *W*[*HP*[*pos*+1]]))
6.         *pos*++;
7.     **if** (*W*[*item*] <= *W*[*HP*[*pos*]])
8.         *not\_found* = false;
9.     **else** *HP*[*pos*/2] = *HP*[*pos*];
10.         *pos* = 2\**pos*;
11. **end while**
12. *W*[*pos*/2] = *item*;

**Procedure Par-heapify(HP[], W[], N)**

**Input:** *N* // number of nodes  
*W[N]* // weight array  
*HP[N]* // heap array of size *N*  
**plural variable :** *I*

1. **All** processors (with id="i<sub>proc</sub>") **dopar**
2.     **for** (*i*=1; *i*<*N*; *i*++)
3.         *HP*[*i*] = *i*;
4.     *HP*[0] = *i<sub>proc</sub>*;
5.     *HP*[*i<sub>proc</sub>*] = 0;
6.     **for** (*i* = *N*/2; *i* > 0; *i*-)
7.         **Par\_adjust\_min**(*HP*[], *W*[], *i*, *N*);
8. **end dopar**

Figure 2. Procedure Par-adjust and Par-heapify

**2.4.4 Merge Trees and Update to a New Root.** After all trees have completed their voting of the incoming edges, the next subtask is to perform the tree merge operations of

all trees of RNN pairs or trees along the NN-chains. This involves first updating the root pointer for the root nodes of the trees of RNN pairs or trees along the NN-chains, which is accomplished by using the Maspar's *router()* function to access the data in another processor. The double-pointer technique is used in TC-NNC to ensure an  $O(\log n)$  worst-case time complexity. After all root nodes have finished updating to the new root, we then update the root pointers for the non-root nodes of each tree of RNN pairs or trees along the NN-chains.

### 3. The Heap Traversal Technique

We now describe a new heap traversal technique which we used to improve the execution time of the original TC-NNC/TC-RNN algorithm. Consider the NN-chain construction phase of algorithm TC-NNC/TC-RNN (Step 5(a) in Figure 1). In each tree-merge iteration, every node of a tree chooses a shortest outgoing edge to its nearest neighbor tree by making a procedure call to *Par-delete-min*. The *Par-delete-min* procedure takes the top item from a heap and returns it to the calling program, replaces the top item with the last item of the heap, then calls the procedure *Par-adjust\_min* to restore the heap property. The *Par-delete-min* operation takes  $O(\log n)$  worst-time in each iteration, which is one of the most significant factors that influence the efficiency of TC-NNC/TC-RNN.

```

Procedure get_min(HP[], Ti, idx, N)
  Input: N // number of nodes
           Ti // Auxiliary binary search tree
           HP[N] // heap array of size N
           idx // index of item to be traversed in the heap
  plural variables: Lson, Rson;
  1. Lson = 2*idx;
  2. Rson = Lson+1;
  3. if {HP[Lson], HP[Rson]} > HP[Min(Ti)]
  4.   treemin = Remove_Min(Ti)
  5.   insert_into_tree(Ti, Rson);
  6.   insert_into_tree(Ti, Lson);
  7.   return (treemin);
  8. end if
  9.
  10. if (W[HP[Lson]] < W[HP[Rson]])
  11.   insert_into_tree(Ti, Rson);
  12.   return (Lson);
  13. else
  14.   insert_into_tree(Ti, Lson);
  15.   return (Rson);
  16. end if
  
```

Figure 3. Procedure *get\_min* using the heap traversal technique

We devised a *heap traversal* technique which improves the TC-NNC/TC-RNN algorithm by avoiding the calls to *Par-adjust\_min*. Instead of adjusting the heap structure after each *delete-min* operation, the *heap traversal* technique only 'traverses' the heap in increasing order of the data values without making any changes to the heap. Note that performing a *heap traversal* is like performing an in-order traversal of a binary search tree; the only difference is that the heap structure does not provide enough ordering information as in a binary search tree, thus the procedure to traverse a heap is more complicated and requires auxiliary storage. We now describe the implementation details of *heap traversal* as follows

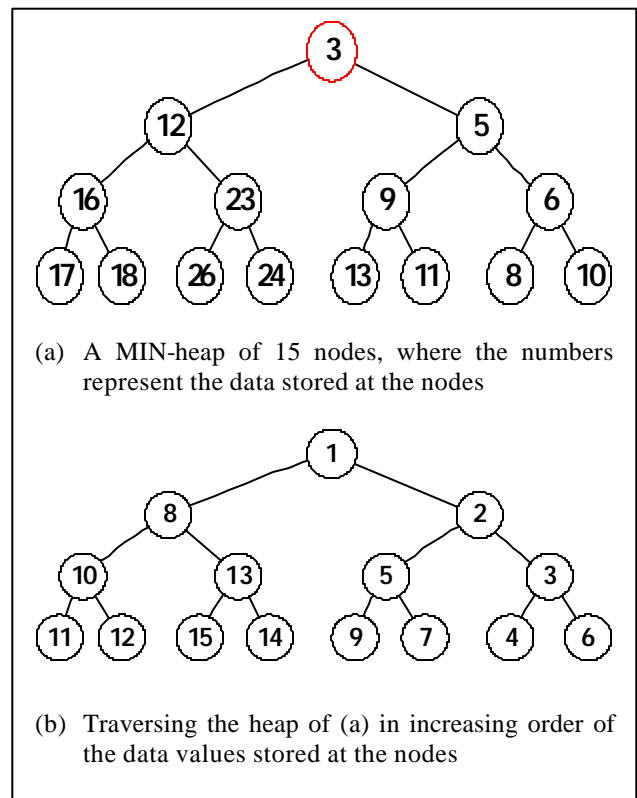


Figure 4. An example of heap traversal

Figure 3 shows the new version of procedure *get\_min* applying the heap traversal technique. Figure 4 uses an example to demonstrate heap traversal. Figure 4(a) shows a MIN-heap of 15 nodes, where the numbers at the nodes represent the data stored at the nodes. When we apply the heap traversal technique to 'visit' the nodes in the heap of Figure 4(a), the order in which the nodes are traversed is shown in Figure 4 (b). We now describe the implementation details of *heap traversal*.

Assume a given heap is stored in an array *HP[]* of size *n*. The *heap traversal* version of *delete-min* starts

from the top of the heap  $HP[1]$ , then traverses the nodes of the heap one by one in the order of the parent, older child (smaller data), and younger child (larger data). After each parent node (say  $HP[k]$ ) has been processed (and returned as the result of a call to  $get-min$ ), the two child nodes ( $HP[k*2]$  and  $HP[k*2+1]$ ) will be examined next. The larger of the two child nodes will be put into an auxiliary data structure for future processing; the smaller will be put in a temporary variable to be returned on the next  $delete-min$  call (if it is smaller than the smallest item of the auxiliary data structure). This traversal operation of the parent, older child, and younger child, is repeated until all nodes in the heap are processed. The auxiliary data structure used for *heap traversal* must also keep the ordering information of nodes that are put into it. A binary search tree (or even a binary heap) will be a good choice. Notice that since the height of the auxiliary data structure tends to be ‘shallower’ than the original data heap, the time efficiency of TC-NNC/TC-RNN is improved by applying this new technique.

#### 4. Empirical Results

All three algorithms were implemented on a SIMD parallel computer MasPar MP-1 with 8192 processors. We generated randomly weighted graphs as input for our experiments; the same data we have been used in our pervious experiment [10, 11]. A biased-random weight-matrix generator was used to construct the input graphs for which the initial MST has a high value for the maximum node-degree. The random-graph generator takes the following input parameters (more details can be found in [10, 11]):

- $n$  – the size of the matrix;
- $f$  – the number of nodes with large degree; and
- $ld$  ( $ud$ ) – lower (upper) bounds for the degree of the large-degree nodes.

Figure 5 shows the experimental results comparing the three algorithms for the 5-MST problem. For all the randomly weighted input graphs, the degree bounds  $ld$  and  $ud$  were fixed at 15 and 20, respectively, as the number of nodes  $n$  varies from 500 to 3500. The runtime for the TC-NNC algorithm ranges from 0.70 seconds ( $n = 500$ ) to 2.14 seconds ( $n = 3500$ ), which is much faster than the TC-RNN algorithm (2.56 seconds to 9.87 seconds), and is mostly faster than the IR algorithm (0.68 seconds to 5.61 seconds). These experimental results demonstrate that algorithm TC-NNC now outperform the other two approximate algorithms in terms of execution time and quality-of-solution (results of quality-of-solution are same as previous experiment in [11]).

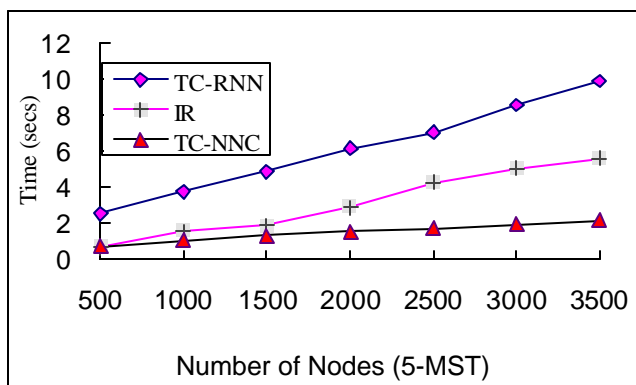


Figure 5. Comparison of MP-1 execution time of IR, TC-RNN and TC-NNC for computing  $d$ -MST with degree-bound =5, using randomly-weighted complete graphs with an MST forced to have max-degree 20.

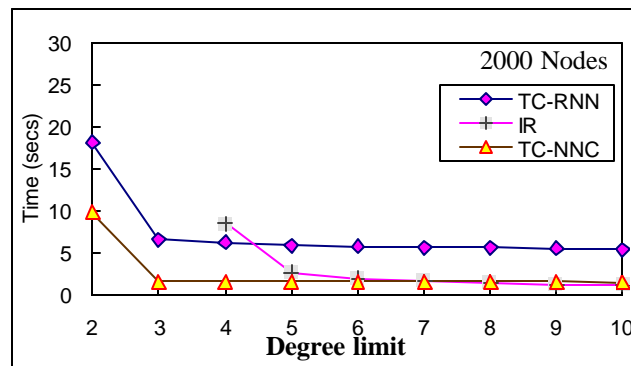


Figure 6. Comparison of MP-1 Execution Time of IR, TC-RNN, and TC-NNC for computing  $d$ -MST with varying degree-bound  $d$ , using a randomly-weighted complete graph of 2000 nodes with an MST forced to have max-degree 20.

To study how the degree constraints may affect the performance of these algorithms, we compared the three algorithms using a randomly weighted graph of 2000 nodes with varying degree constraints. Figure 6 presents the experimental results. When the degree constraint increases from 2 to 10, the execution times of these algorithms decrease and approach the same limit. These results demonstrate that the TC-NNC algorithm still keep stable and competitive over a range of the degree constraints.

## 6. Conclusion

In this paper, we proposed a *heap traversal* approach to improve the parallel TC-NNC/TC-RNN algorithm for solving the *d*-MST problem. The experimental results on randomly weighted graphs demonstrated the following:

- The execution time of TC-NNC is smaller than that of TC-RNN and IR; and
- The quality of solutions of TC-NNC is better than that of IR and is very close to that of TC-RNN.

For further research, we plan to apply other auxiliary data structures for *heap traversal*. We also plan to improve the performance of the IR algorithm by using other more efficient penalty functions, and apply the ideas of iterative refinement and nearest neighbor chains to other constrained spanning tree problems.

## REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of computer Algorithm*. Addison-Wesley, 1974.

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

[3] N. Deo and S. L. Hakimi. The shortest generalized Hamiltonian tree. *Proc. 6th Annual Allerton Conference*, pp. 879-888, 1968.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[5] B. Gavish. Topological design of centralized computer networks -- Formulation and algorithms. *Networks*, 12(4):355-377, 1982.

[6] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138-1162, 1970.

[7] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1(1):6-25, 1971.

[8] S. Khuller, B. Raghavachari, and N. Young. Low degree spanning tree of small weights. *Proc. of 26th Annual ACM STOCS*, pp. 412-421, 1994.

[9] M. Krishnamoorthy, G. Craig, and M. Palaniswami. Comparison of heuristic algorithms for the degree constrained minimum spanning tree. In I. H. Osman and J. P. Kelly, editors, *Metaheuristics: Theory and Applications*, pp. 83-96, 1996.

[10] N. Kumar, Parallel Computation of Constrained Spanning Trees: Heuristics and SIMD Implementations. Ph.D.

Dissertation, Department of Computer Science, University of Central Florida, Orlando, 1997.

[11] L. J. Mao, N. Deo, N. Kumar, and S. D. Lang. A Comparison of Two Parallel Approximate Algorithms for the Degree-Constrained Minimum Spanning Tree Problem. *CONGRESSUS NUMERANTIUM*, 123, pp. 15-32, 1997.

[12] L.-J. Mao, N. Deo, and S.D. Lang, A Parallel Algorithm for the Degree-Constrained Minimum Spanning Tree Problem Using the Nearest Neighbor Chains, Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN99), pp. 184-189. 1999.

[13] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics*, DIMACS Workshop, March 12-14, 1992. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, V. 15, pp. 99-117, 1994.

[14] S. C. Narula and C. A. Ho. Degree-constrained minimum spanning tree. *Computers and Operations Research*, 7(4):239-249, 1980.

[15] C. H. Papadimitriou and U. V. Vazirani. On two geometric problems related to traveling salesman problem. *J. Algorithms*, 5:231-246, 1984.

[16] R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III. Many birds with one stone: Multi-objective approximation algorithms. *Proc. of the 25th Annual ACM Symposium on Theory of Computing (STOCS '93)*, pp. 438-447, 1993.

[17] M. Savelsbergh and T. Volegnant. Edge exchanges in the degree-constrained minimum spanning tree problem. *Computers and Operations Research*, 12(4):341-348, 1985.

[18] T. Volegnant and R. Jonker. The symmetric traveling salesman problem and edge exchanges in minimum 1-trees. *European Journal of Operational Research*, 12(4):394-403, 1983.

[19] T. Volegnant. A Lagrangian approach to the degree-constrained minimum spanning tree problem. *European Journal of Operational Research*, 39(3):325-331, 1989.

[20] M.A. Weiss. Data structures and algorithm analysis in C. 2<sup>nd</sup> edition, Addison-Wesley, 1997.

[21] C.-C. Yao. An  $O(E \log \log V)$  algorithm for finding minimum spanning trees. *Inf. Process. Lett.*, 4(1):21-23, 1975.

[22] Y. Yamamoto. The Held-Karp algorithm and degree-constrained minimum 1-tree. *Mathematical Programming*, 15:228-238, 1978.