Rotation Algorithms for Balancing Search Trees

Chumphol Bunkhumpornpat[1], **Varin Chouvatut**[2], and **Saowaluk Rattanaudomsawat**[3], Non-members

ABSTRACT

A search tree is a data structure constructed from a minimum spanning tree. This data structure is used for determining the cluster membership of a query instance clustered by a similarity-guaranteed clustering algorithm. According to the line topology of a search tree in the worst case, the time complexity of tree traversing is O(n) where n is the number of nodes of the tree. Unfortunately, the AVL tree algorithm cannot solve this problem because the algorithm is unable to maintain the hierarchical structure of a search tree. From the definition of balance factor, our proposed algorithm is designed to rotate nodes until the tree becomes balanced while maintaining the hierarchical structure. Consequently, the balanced search tree achieves the optimal time complexity of O(log n) for the searching purpose.

Keywords: Data Structure, Clustering, Search Tree

1. INTRODUCTION

Tree searching Cormen T.H. et al., (2009) Drozdek A. (2010) Rosen K.H. (2006) is applied to seek a target instance, among a collection of instances in a data structure called a tree. The search is one of the significant problems in computer science since it involves several research fields such as data structures and algorithms, graph algorithms, approximation algorithms, computational geometry, and data mining. A well-known and widely-used application in tree searching is a binary search tree Cormen T.H. et al., (2009) Drozdek A. (2010) Rosen K.H. (2006).

In the clustering domain Jain A.K. et al., (1999) Brandes U. et al., (2003) Han J. et al., (2011), the existing clustering problems have no relationships with the tree searching problem. Even if there is some clustering algorithms Knorr E.M. and Ng R.T., (1997a, 1998b) using multidimensional indexing structures such as R-Trees or k-d Trees, the two problems are still not associated with each other. However, the similarity-guaranteed clustering algorithm (SIGCA) proposed in Kantabutra S. and Bunkhumpornpat C. (2004) considers both problems simultaneously.

In SIGCA algorithm, a minimum spanning tree (MST) Cormen T.H. et al., (2009) Drozdek A. (2010) Rosen K.H. (2006) is fed and a clustering tree is then returned to the binary tree property. The clustering tree can be used not only as a tool for clustering instances in a dataset, but also as a search tree for searching a cluster membership (a cluster ID) of a given instance. This advantage is useful for various applications. Moreover, the SIGCA can specify an inner-cluster distance and an intra-cluster distance, called diameter Ronkainen P. (1998), of a cluster of instances in the d-dimensional space.

Unfortunately, due to the line topology of a search tree in the worst case, the disadvantage of SIGCA is that its time complexity for traversing a search tree is O(n). The traversing is slower than those of the other tree-searching algorithms, for example, a binary search tree takes only O(log n); where n is the number of nodes in the tree.

In this paper, we present the rotating algorithms to adjust internal nodes, which affect a search tree; where the rotating process will persist until the tree becomes balanced. Consequently, the balanced search tree will acquire the balanced binary tree property Cormen T.H. et al., (2009) Drozdek A. (2010) Rosen K.H. (2006) dealing with the disadvantage of SIGCA and thus obtain the optimal time complexity of O(log n) for searching the balanced binary tree Cormen T.H. et al., (2009).

This paper is structured as follows. Section 2 briefly explains the related work. Section 3 proposes the design and analysis of our algorithms. Section 4 summarizes our research.

2. RELATED WORK

In research Kantabutra S. and Bunkhumpornpat C. (2004), their objective is to build a clustering tree from an MST, and then uses this tree for searching a cluster membership. We repeat the recursive algorithm to build the data structure and its definition as follows.

Definition 1: Given a minimum spanning tree T, a clustering tree is a rooted binary tree in which each

Manuscript received on February 8, 2016 ; revised on April 19, 2016.

Final manuscript received on May 19, 2016.

[1,2,3] The authors are with Theoretical and Empirical Research Group Department of Computer Science, Faculty of Science, Chiang Mai University Chiang Mai 50200, Thailand, E-mail: chumphol.b@cmu.ac.th, varin.ch@cmu.ac.th and saowaluk.r@gmail.com

node contains, among other things, a distance between one pair of instances a, b in T and the corresponding instance identification numbers of a and b. Each parent node always contains a longer distance than that of its children.

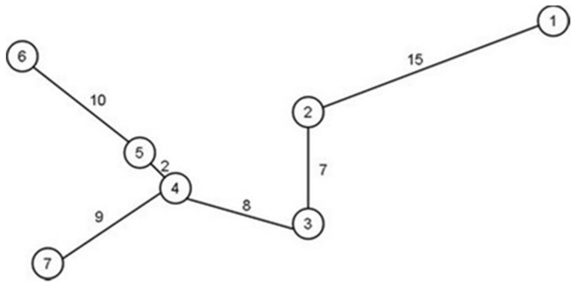


Fig.1: *Minimum spanning tree MST_A.*

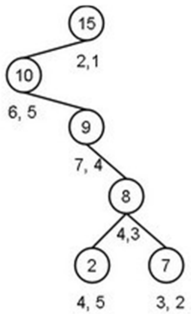


Fig.2: *Clustering tree: CT_A.*

From Definition 1, CT_A in Fig.2 is constructed from MST_A in Fig.1. An example relationship between a clustering tree (CT) and a minimum spanning tree (MST) is that node 15 of CT_A contains a distance between the connected nodes 2 and 1 of the MST_A.

A clustering tree obtaining the subcluster property and the hierarchical structure means that clusters of a child node are sub-clusters of a parent node's clusters.

Algorithm Clustering Tree
Input: an MST T and its root r.
Output: a clustering tree S.

```
Tree(T, r)
1. if(T has some edge)
2.    e = get-max-edge(T)
3.    rs = make-blank-node()
4.    fill-info(rs, e)
5.    return connect(Tree(leftSub-tree(r), r.left-
       node-id),Tree(rightSub-tree(r), r.right-node
       -id), rs)
6. else
7.    rs = make-blank-node()
8. return rs
```

In the *Clustering* Tree algorithm, an edge with the maximum weight of an MST is firstly built as a root node of the clustering tree. The child node is then built by partitioning this MST into two subtrees. After that, the other nodes will be built until all edges are operated. This algorithm takes O(n^2) where n is the number of edges of the MST. Once the clustering tree is constructed, the following algorithm will be executed to form clusters.

Algorithm Clustering
Input: a clustering tree S and a desired diameter d.
Output: k clusters where k is also an output.

```
Clustering(S, d)
1. if(diameter(S) > d)
2.    Clustering(left-sub-tree(root(S)), d)
3.    Clustering(right-sub-tree(root(S)), d)
4. else
5.    root(S).cluster-number = cluster-number()
6.    print-cluster-number()
7.    traverse-and-print(S)
8. return
```

The *Clustering* algorithm may be explained as an example that it discovers three clusters, including {1}, {6}, and {2, 3, 4, 5, 7} by retrieving a clustering tree when the parameter d is set to 9 by the user. The maximum distance between any pair of instances in any cluster will not exceed the guaranteed distance d. The algorithm takes O(n^3); n is the number of nodes in the clustering tree.

In some clustering applications, users may need to query a cluster membership of a given instance. According to Definition 1, a clustering tree can be applied as a search tree for this searching purpose by executing the following algorithm.

Algorithm Search
Input: a clustering tree S, its root node r, and a query instance p.
Output: a cluster membership number of p.

```
Search(S, r, p)
1. if(r.cluster-number = nil)
2.    if(d(r.left-point, p) > d(r.right-point, p))
3.       Search(right-sub-tree(r), r.right-node, p)
4. else
5.    if(d(r.left-point, p) < d(r.right-point, p))
6.       Search(left-sub-tree(r), r.left-node, p)
7.    else
8.       print(r.cluster-number)
9. return
```

An assumption of the *Search* algorithm is that no case of $d(r.left-point, p) = d(r.right-point, p)$; besides, the mechanism of this algorithm is similar to that of a binary search tree. The algorithm takes O(h) where h is the height of the search tree. An important lemma of SIGCA is recalled as follows.

Lemma 1: Let S be a clustering tree, r be the root node of the S, and p be a given instance. If $d(y, p) < d(z, p)$ where y is one instance in the root node r and z is the other instance in r, then $d(x_1, p) < d(x_2, p)$ for all instances x_1 in the same sub-tree as y's and for all instances x_2 in the same sub-tree as $z's$.

By Lemma 1, the Euclidean function $d(x, y)$ returns a Euclidean distance between the instances x and y in the d-dimensional space where x and y represent nodes in an MST. A node of the clustering tree contains a distance between x and y. In addition, p and y will be considered to be in the same cluster if the distance between p and y is less than that between p and z. In Fig.2, for example, we assume r is node 8 thus y and z are nodes 4 and 3, respectively.

3. PROPOSED ALGORITHM

The objective of this paper is to reconstruct an imbalanced search tree to be balanced. We aim to achieve the optimal time searching in a complete binary tree that takes O(log n) where n is the number of nodes in the tree.

In this section, the property of having the same data structure between a clustering tree and a search tree is considered. Typically, the term "clustering tree" is used when the tree is converted from an MST while the term "search tree" is used when the clustering tree is traversed in order for searching.

3.1 Balanced Search Tree

A balanced search tree is a balanced binary tree applied for searching the cluster membership of a query instance. In addition, the time complexity of searching in a balanced search tree equals that of a balanced binary tree, which is optimal. A balanced search tree is defined as follows.

Definition 2: A balanced search tree is a balanced binary tree in which the difference between the heights of the left and right sub-trees of every internal node (Balance Factor: BF) does not exceed 1.

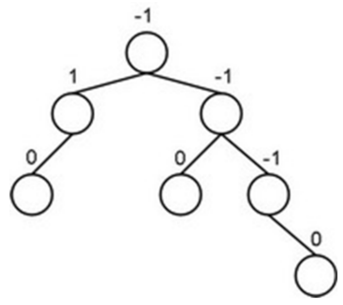


Fig.3: *Balanced search tree.*

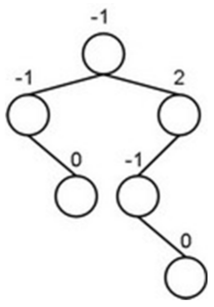


Fig.4: *Unbalanced search tree.*

A balanced search tree and an unbalanced search tree are illustrated in Fig.3 and Fig.4, respectively. In Fig.3, the specified numbers represent BFs of nodes. From Definition 2, we conclude that a search tree is balanced if BF of every internal node be only -1, 0, or 1. In Fig.4, the search tree is not considered to be balanced because there exist at least one BF of an internal node that is not -1, 0, or 1 (the right child-node of the root node has BF of 2).

The *Search* algorithm takes time complexity of O(n); n is the number of nodes in a search tree. This time complexity is not optimal for searching in a balanced binary tree since the optimal value will be obtained only when the search tree is balanced. Our statement can be proved in Theorem 1 below.

Theorem 1: The *Search* algorithm takes the optimal time complexity of O(log n) for a balanced search tree where n is the number of internal nodes.
Proof :
According to Definition 2, since a balance search tree is a balanced binary tree so the height of a balance search tree is O(log n) where n is the number of internal nodes. From the time complexity of the *Search* algorithm, O(h) where h is the height of a search tree, the Search takes O(log n) for a balance search tree. This is the optimal time complexity of a balanced binary tree.□

The objective of our research is to reconstruct an unbalanced search tree to be balanced, in order to achieve a speedup in time complexity to the optimal of searching in a balanced binary tree.

3.2 Rotation Algorithm

The rotation algorithms are applied to rotate the internal nodes of a search tree. The algorithms are similar to the mechanism of the AVL Tree rotations Cormen T.H. et al., (2009) Drozdek A. (2010) Rosen K.H. (2006). In case of blank nodes, they are set to 0. The following are four rotation algorithms.

```
Algorithm Rotate Left Left
Input: a root node $i$
Output: a new root node $i$.left
Rotate_Left_Left($i$)
1. set-root($i$.left)
```

```
2. $i$.left.left = $i$
3. $i$.left = $i$.left.left
4. return $i$.left
```

Algorithm Rotate Left Right
Input: a root node i
Output: a new root node i.left
Rotate_Left_Right(i)

```
1. set-root($i$.left)
2. $i$.left.right = $i$
3. $i$.left = $i$.left.right
4. return $i$.left
```

Algorithm Rotate Right Left
Input: a root node i
Output: a new root node i.right
Rotate_Right_Left(i)

```
1. set-root($i$.right)
2. $i$.right.left = $i$
3. $i$.right = $i$.right.left
4. return $i$.right
```

Algorithm Rotate Right Right
Input: a root node i
Output: a new root node
Rotate_Right_Right(i)

```
1. set-root($i$.right)
2. $i$.right.right = $i$
3. $i$.right = $i$.right.right
4. return $i$.right
```

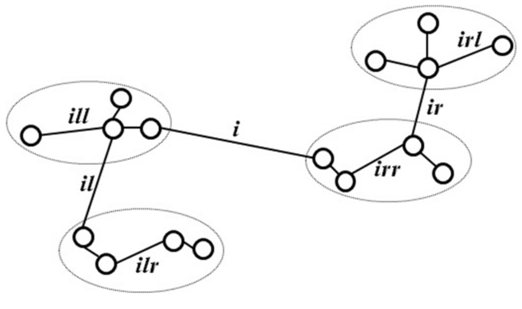


Fig.5: *General MST.*

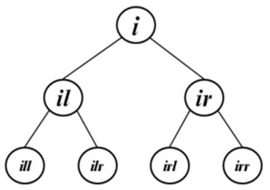


Fig.6: *General clustering tree.*

A general clustering tree shown in Fig.6 is constructed from a general MST in Fig.5. In the two figures, letters i, il, ir, ill, ilr, irl, and irr represent the weights of this MST with respect to the values contained in nodes of the clustering tree.

i represents the maximum weight of the MST; in addition, i is contained in the root node of the clustering tree to be rotated by the rotation algorithms.

il represents the maximum weight of the left sub-tree of the MST partitioned at i, whose edge connects the left sub-tree to the right. Both left and right groups are of the left sub-tree of this MST. In addition, i is contained in the left node of i in the clustering tree. The interpretation of ir shall include il and vice versa.

l and r are abbreviated for "left" and "right", respectively. For example, ilr represents $i.left.right$, which represents the maximum weight of the right sub-tree of the left sub-tree of the MST partitioned at i and il, respectively. Its edge connects between the left and right sub-trees of the right sub-tree of the left sub-tree of this MST; in addition, i is contained in the right node of the left node of i in the clustering tree. The interpretations of ill, irl, and irr shall include ilr and vice versa.

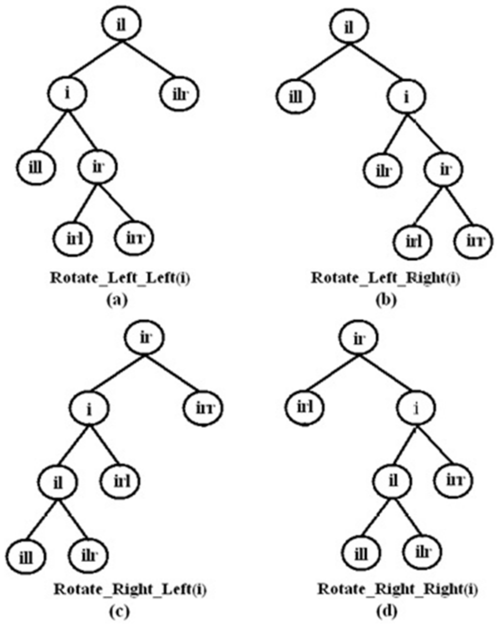


Fig.7: *Rotation algorithms.*

Fig.7 illustrates four clustering trees after executing the four rotations with i be the root node of the clustering tree from Fig.6. There are four rotation cases that remain the hierarchical structure in a clustering tree as proved in Lemma 2.

Definition 3: A clustering tree retains the hierarchical structure if and only if any pair of MST nodes with respect to nodes in the same sub-clustering tree can be communicated even after cutting an MST edge connecting to root of the tree.

Lemma 2: After a node of a clustering tree is rotated, the sub-tree rooted at the rotated node remains the hierarchical structure.

Proof:
We prove this lemma case by case. In the first case illustrated in Fig.7 (a), the node i of the clustering tree from Fig.6 is rotated by $Rotate\_Left\_Left$. When the edge i in the sub-tree of MST from Fig.5 corresponding to the clustering sub-tree rooted at i in Fig.7 (a) is partitioned, this sub-tree is divided into two sub-trees. The first sub-tree has the edge ill and the second sub-tree has the edges ir, irl, and irr. Two sub-trees of the MST from Fig.5 still rely on the clustering sub-tree rooted at i in Fig.7 (a) thus the hierarchical structure is remaining. With no loss of generality, we do not prove the rest cases.□

3.3 Balancing Algorithm

The balancing algorithm is applied to restructure a search tree to be balanced by calling the rotation algorithms. The algorithm is as shown as follows.

```
Algorithm Balance
Input: a search tree $T$ and its root $r$.
Output: a balanced search tree $T$.
Balance($T, r$)
1. lookup($T, r$)
2. inorder($T, r, A$)
3. if (is_balance($A$) = TRUE)
4.    return $T$
5. for i = sizeof($A$) - 1 downto 0
6.    $\pi$= T.node[$A[i]$.node]
7.     if ($A[i]$.BF > 1)
8.       if ($d(\pi, \pi$.left.left$) \leq d(\pi, \pi$.left.right$)$)
9.         $\pi$ = Rotate_Left_Left($\pi$) ; case 1
10.      else
11.        $\pi$ = Rotate_Left_Right($\pi$) ; case 2
12.      exit_loop()
13.   else if ($A[i]$.BF < -1)
14.      if($d(\pi, \pi$.right.left$) \leq d(\pi, \pi$.right.right$)$)
15.        $\pi$= Rotate_Right_Left($\pi$) ; case 3
16.      else
17.        $\pi$= Rotate_Right_Right($\pi$) ; case 4
18.      exit_loop()
19. return Balance($T, r$)
```

The *Balance* algorithm is explained as the following steps.

Line 1: *lookup* determines BF of each node of the search tree T.

Line 2: *inorder* contains BF of each node in array A. This step is done by performing Inorder Tree Traversal Cormen T.H. et al., (2009) on T.

Lines 3 and 4: *is_balance* checks each BF in A. The tree T is returned if T is a balanced search tree whose every BF in A is in the range [-1, 1].

Line 5: the loop for traversing T in reverse order is processed. The reverse order ensures that T is initially restructured by rotating from the leaf node of the rightmost sub - tree through the root node. Thus, re-rotation on the same node can be prevented.

Each iteration of the downward loop guarantees that every searching subtree whose root node descending from the currently rotated node will be balanced.

Line 6: represents a node of T corresponding to the node index in array A.

Lines 7-17: BF of the considering node is checked to choose one suitable rotation case of the four.

Line 18: *exit_loop* terminates the downward loop.

Line 19: *Balance* is called recursively until T is balanced.

How the *Balance* algorithm operates on the spanning tree ST_B from Fig.8 is illustrated in Fig.9. The algorithm starts considering BF of each node in ST_B, which is built from MST_B. After that, node 5 is rotated by *Rotate_Right_Left*, the third case. Next, the algorithm is called recursively. Thus, node 8 is rotated by *Rotate_Right_Left*, again, the third case. Finally, the algorithm terminates because the BF's of all nodes are in the range of [-1, 1]; this means ST_B becomes a balanced search tree.

Once the *Balance* algorithm is executed to restructure a search tree with n nodes to be balanced, according to Lemma 2, searching in the balanced search tree would take O(log n). The *Balance* takes one additional time costing O(n^3) as proved in Theorem 2.

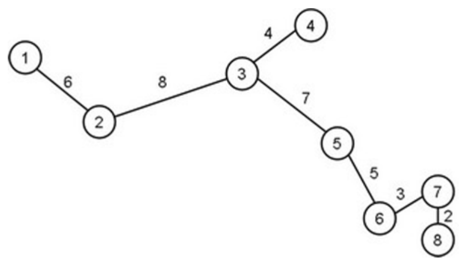
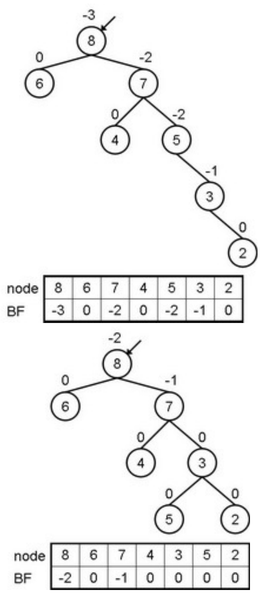
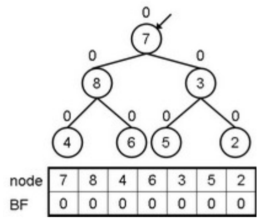


Fig.8: *Minimum spanning tree: MST_B.*



node	8	6	7	4	5	3	2
BF	-3	0	-2	0	-2	-1	0

node	8	6	7	4	3	5	2
BF	-2	0	-1	0	0	0	0



node	7	8	4	6	3	5	2
BF	0	0	0	0	0	0	0

Fig.9: *Balance on* ST_B.

Theorem 2: The Balance algorithm takes O(n^3) on a search tree with n nodes.
Proof :
The time complexity of each line in the algorithm is analyzed below.

Line 1 takes O(n^2) because computing BF of one node takes O(n).

Line 2 takes O(n) for Inorder Tree Traversal.

Lines 3-4 takes O(n) to scan all members in A and the return T if any.

Line 5 takes O(n) for accessing the array of size n.

Line 6 takes O(1) to acquire a node index.

Lines 7-18 take O(1) for checking four conditions before rotating a node. Each rotation takes one more O(1).

Line 19 takes O(1).

In the worst case, the algorithm is recursively called for O(n) times when every node must be rotated. Let T(n) be the time complexity of the algorithm and derived as

$$\begin{aligned} T(n) &= O(n)\cdot(O(n^2)+O(n)+O(n)+O(n)\cdot(O(1)+O(1))+O(1)) \\ &= O(n)\cdot(O(n^2)+O(n)+O(1)) \\ &= O(n^3). \end{aligned}$$

Therefore, the Balance takes O(n^3).□

Correctness of the *Balance* (a restructured search tree becomes balanced and also remains the hierarchical structure) is proved in Theorem 3.

Theorem 3: The *Balance* algorithm is correct.
Proof :
Let n be the level of a search tree, and H be the height of the tree. This theorem is proved using the induction on a tree with level n applied to Lemma 2.

Base Case: $n = H - 2$ If BF of a node at this level is not in [-1, 1], the node is correctly rotated according to Lemma 2. Thus, a search sub-tree rooted at the rotated node becomes balanced with the hierarchical structure.

Inductive Step: $n < H - 2$
Since the rotation on a search sub-tree rooted at $n = H - 2$ is correct, a node in level $n < H - 2$ will initially be rotated from n to $n - 1$ according to Lemma 2. Then, the search sub-tree rooted at the rotated node would become balanced with the hierarchical structure.
Obviously, if $n = H$ or $n = H - 1$, there are no nodes to be rotated because the BF of each node at this level is in fact in [-1, 1]. Besides, the induction on n terminates at the level $n = 0$ because the node at this level is a root node. Consequently, the correctness of Balance has been proved.□

4. CONCLUSIONS

This paper aims to improve the time complexity for searching in a search tree (clustering tree) constructed from an MST using SIGCA.

The proposed algorithms are applied for reconstructing an unbalanced search tree to be balanced. The balanced tree also has the balanced binary tree's property, that is, the height of the tree with n nodes is O(log n). Consequently, searching in a balanced search tree takes the optimal time complexity of O(log n); faster than the time complexity of O(n) taken by an unbalanced search tree.

Although the optimal search is achieved, the balanced search tree loses the MST-based clustering property. This is because the root node of the search tree may not contain the maximum weight of an MST. Thus, the dual trees are required, i.e. one original clustering tree for clustering and one balanced search tree for searching speedup.

Our further research would improve the algorithm's performance so that the clustering property together with the searching property will be retained while building a balanced search tree. Thus, we would be able to use a single tree for both clustering and searching.

ACKNOWLEDGMENTS

Authors would like to acknowledge that this research is supported by Academic Support Grant from Department of Computer Science, Faculty of Science, Chiang Mai University.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to Algorithms*, 3rd Edn., McGraw-Hill Book Company, New York, 2009.

[2] A. Drozdek, *Data Structures and Algorithms in C++*, 3rd Edn., Brooks/Cole, 2006.Sabin, W.A. The Gregg Reference Manual: A Manual of Style, Grammar, Usage, and Formatting. McGraw-Hill, New York, NY, USA, 2009.

[3] K.H. Rosen, *Discrete Mathematics and Its Applications*, 6th Edn., McGraw-Hill, 2006.

[4] A.K. Jain, M.N. Murty and P.J. Flynn, "Data

Clustering: A Review," *ACM computing surveys (CSUR)*, 31(8):, pp.264-323, 1999.

[5] U. Brandes, M. Gaertler and D. Wagner, *Experiments on Graph Clustering Algorithms*, Lecture Notes in Computer Science 2832: pp. 568-579, 2003.

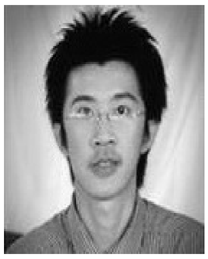
[6] J. Han, M. Kamber and J. Pei, *Data Mining: Concepts and Techniques*, 3rd Edn., Morgan Kaufmann Publishers, San Francisco, 2011.

[7] E. M. Knorr, and R. T. Ng., "A unified notion of outliers: Properties and computation," *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD 97)*, Newport Beach, CA, pp.213-222, 1997.

[8] E. M. Knorr, and R. T. Ng., "Algorithms for mining distancebased outliers in large datasets," *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB 98)*, New York, pp.24-27, 1998.

[9] S. Kantabutra and C. Bunkhumpornpat, "Two birds with one stone: a similarity-guaranteed clustering algorithm and its search tree," *TENCON 2004. 2004 IEEE Region 10 Conference*, Chiang Mai, Thailand, Vol. 2, pp. 251-254, 2004.

[10] P. Ronkainen, *Attribute Similarity and Event Sequence Similarity in Data Mining*, Licentiate Thesis, University of Helsinki, Sweden, 1998.



Chumphol Bunkhumpornpat received his Ph.D. in Computer Science from Chulalongkorn University. He is currently an Assistant Professor and the group leader of TERG (Theoretical and Empirical Research Group) in the Department of Computer Science, Chiang Mai University. His research interest is Pattern Recognition-especially in the Class Imbalance Problem. In particular his laboratory is currently working on improving over-sampling and under-sampling techniques on imbalanced domains.

Varin Chouvatut received her Ph.D. in Electrical and Computer Engineering from King Mongkut's University of Technology Thonburi. She is currently an Assistant Professor and a member of TERG in the Department of Computer Science, Chiang Mai University. Her research of interest includes Computer Graphics, Image Processing, and Computer Vision.

Saowaluk Rattanaudomsawat received her M.S. in Computer Science from Chiang Mai University. She was a lecture at School of Information Technology, Mae Fah Luang University. In this research, she is working with the TERG laboratory, CMU. Her research of interest includes randomized algorithms and graph algorithms.