A Weighted-Tree Simplicity Algorithm for Similarity Matching of Partial Product Descriptions

Lang, L.; Sarker, B.K.; Bhavsar, V.C.; Boley, Harold

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC · CNRC

A Weighted-Tree Simplicity Algorithm for Similarity Matching of Partial Product Descriptions *

Lang, L., Sarker, B.K., Bhavsar, V.C., and Boley, H.
July 2005

* published in the Proceedings of The International Society for Computers and Their Applications (ISCA) 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2005). Toronto, Ontario, Canada. July 20-22, 2005. pp. 55-60. NRC 48534.

Canada

A WEIGHTED-TREE SIMPLICITY ALGORITHM FOR SIMILARITY MATCHING OF PARTIAL PRODUCT DESCRIPTIONS

Lu Yang
Biplab K. Sarker
Virendrakumar C. Bhavsar
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada E3B 5A3
{lu.yang, sarker, bhavsar} AT unb.ca

Harold Boley
Institute for Information Technology e-Business
National Research Council
Fredericton, New Brunswick, Canada E3B 9W4
harold.boley AT nrc-cnrc.gc.ca

Abstract

Our weighted-tree similarity algorithm matches buyers and sellers in e-Business environments. We use arc-labeled, arc-weighted trees to represent the products (or services) sought/offered by buyers/sellers. Partial product descriptions can be represented via subtrees missing in either or both of the trees. In order to take into account the effect of a missing subtree on the similarity between two trees, our algorithm uses a (complexity or) simplicity measure. Besides tree size (breadth and depth), arc weights are taken into account by our tree simplicity algorithm. This paper formalizes our buyer/seller trees and analyzes the properties of the implemented tree simplicity measure. We discuss how this measure captures business intuitions, give computational results on the simplicity of balanced k-ary trees, and show that they conform to the theoretical analysis.

Key Words

Arc-labeled and arc-weighted tree, tree similarity, tree simplicity, balanced *k-ary* trees, e-Business, buyer and seller trees.

1. INTRODUCTION

We proposed earlier a weighted-tree similarity algorithm for multi-agent systems in e-Business environments [1], [8]. In e-Business environments, buyers or sellers seek matching sellers or buyers by exchanging the descriptions of products/services carried by them [3].

Trees are a common data structure for information representation in various areas, such as image comparison, information search and retrieval, clustering, classification, Case-Based Reasoning (CBR) and so on. In e-Business environments, we can also use trees instead of the commonly used key words/phrases to represent the product/service requirements and offers from buyers and sellers. Furthermore, we use node-labeled, arc-labeled and arc-weighted trees to represent parent-child relationship of product/service attributes. Thus, not only node labels but also arc labels can embody semantic information. The arc weights of our trees express the importance of arcs (product/service attributes). One objective of this paper is to formally define our arc-labeled and arc-weighted trees.

When we compute the similarity of two trees, it is common that a subtree in one tree might be missing in the other one. The similarity of the missing subtree and the empty tree is obtained by computing the simplicity of the missing subtree. An intuitive requirement of the tree simplicity measure is that wider and deeper trees lead to smaller tree simplicity values. However, since our trees are arc-weighted, we also take into account the contribution from arc weights to the tree simplicity. Each arc weight is multiplied with the simplicity value of the recursive subtree simplicity underneath. Hence, another objective of this paper is to analyze and evaluate the mathematical properties of our tree simplicity measure.

In our approach, the tree simplicity values are in the real interval [0, 1]. The simplest tree is a single node whose simplicity is defined as 1.0. Our tree simplicity measure is formalized as a recursive function. According to this function, the simplicity value of an infinite tree approaches 0.0. We conduct experiments on balanced *k-ary* tree whose arc weights are averaged at each level. The computational results conform to the properties and requirements of our tree simplicity measure.

This paper is organized as follows. Formal definitions of our trees and an overview of our tree similarity algorithm are presented in the following section. In Section 3, we provide the formal descriptions of our tree simplicity measure. Section 4 presents the theoretical and computational results of our tree simplicity measure on balanced *k-ary* trees. Finally concluding remarks are given in Section 5.

2. ARC-LABELED, ARC-WEIGHTED TREES AND SIMILARITY

In a common e-marketplace, both buyers and sellers advertise their product/service requirements and offers. In order to find matching buyers or sellers, a similarity

computation between them is a must to obtain a ranked similarity list for sellers or buyers. We use arc-labeled and arc-weighted trees to represent product/service requirements and offers. Previous tree similarity (distance) algorithms mostly dealt with trees that have node labels only [4], [5], whether they were ordered [7] or unordered [6]. Due to our unique tree representation for product descriptions, we developed a new weighted-tree similarity algorithm [1], [8]. In this section, we present the definitions of our trees which represent the product/service requirements and offers from buyers and sellers. We also present a brief description of our proposed tree similarity algorithm [1].

2.1 Definitions

Definition 1. Node-labeled trees. *A tree $T = (V, E, L_V)$ is a 3-tuple where V, E and L_V are sets of nodes, arcs and node labels, respectively, which satisfy the following conditions:*
1. One element in V is designated as the 'root'.
2. Each element in E connects a pair of elements in V.
3. There is a unique directed path, consisting of a sequence of elements in E, from the root to each of the other elements in V.
4. There is an $(n \rightarrow 1, n \geq 1)$ mapping from the elements in V to the elements in L_V (i.e. different nodes can carry the same labels).

Trees as defined above with only node labels cannot conveniently represent the attributes of products/services, and can lead to ambiguity in some cases.

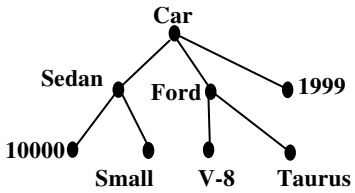

Figure 1. A node-labeled tree (a used car product).

For example, the tree in Figure 1 describes a used car. It is easy to see that this is a small Ford sedan with V-8 engine whose model is Taurus. However, since this is a used car, we cannot figure out what the numbers "1999" and "10000" mean. Although "1999" tends to be the year, it may stand for the price or mileage as well. Similarly, "10000" can also represent the price or mileage of it. Therefore, we add labels to arcs which represent product/service attributes to remove the ambiguity. Figure 2 shows the arc-labeled tree describing the used car shown in Figure 1. We formalize our arc-labeled tree based on Definition 1.

Definition 2. Arc-labeled trees. *An arc-labeled tree is a 4-tuple $T = (V, E, L_V, L_E)$ of a set of nodes V, a set of arcs E, a set of node labels L_V, and a set of arc labels L_E such that (V, E, L_V) is a node-labeled tree and there is an $(n \rightarrow 1, n \geq 1$, fan-out-unique$)$ mapping from the elements in E to the elements in L_E (i.e. non-sibling arcs can carry the same labels).*

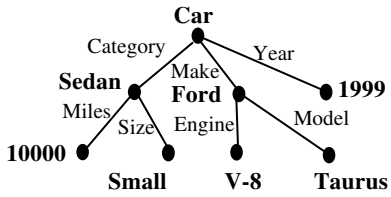

Figure 2. An arc-labeled tree.

For algorithmic convenience we always keep arc labels of siblings in lexicographical order. Now suppose the tree in Figure 2 should represent the requirements of a car buyer. This buyer might have special interests in some car attributes. For example, he/she may want a "Ford" i.e. the maker of the car is quite important to him/her but the "Year" is not that important. In order to reveal special preferences indicated by buyers/sellers, we allow them to specify an importance value (arc weight) for each arc. Figure 3 shows the tree with arc weights. According to the corresponding Definition 3, we represent our arc-labeled and arc-weighted tree as follows.

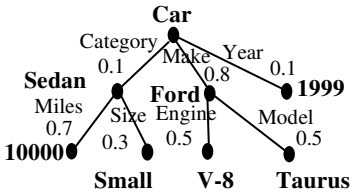

Figure 3. A (normalized) arc-labeled, arc-weighted tree.

Definition 3. Arc-labeled, arc-weighted trees. *An arc-labeled, arc-weighted tree is a 5-tuple $T = (V, E, L_V, L_E, L_W)$ of a set of nodes V, a set of arcs E, a set of node labels L_V, a set of arc labels L_E and a set of arc weights $L_W = [0,1]$ such that (V, E, L_V, L_E) is an arc-labeled tree and there is an $(n \rightarrow 1, n \geq 1)$ mapping from the elements in E to the elements in L_W (i.e. different arcs can carry the same weights).*

Finally, Definition 4 introduces a user and algorithmic convenience, also illustrated in Figure 3.

Definition 4. Normalized arc-labeled, arc-weighted trees. *A normalized arc-labeled, arc-weighted tree is an arc-labeled, arc-weighted tree $T = (V, E, L_V, L_E, L_W)$ having a fan-out-weight-normalized $(n \rightarrow 1, n \geq 1)$ mapping from the elements in E to the elements in L_W, i.e. the weights of every fan-out add up to 1.0.*

Such normalized trees will be assumed throughout for the rest of this paper.

2.2 Tree Similarity

Here, we briefly review our proposed tree similarity algorithm [1], [8]. Generally speaking, our algorithm traverses input trees top-down (root-leaf) and then computes their similarity bottom-up. If two non-empty (sub)trees have identical root node labels, their similarity is computed by a recursive top-down (root-leaf) traversal through the subtrees that are accessible on each level via identical arc labels. The recursion is terminated by two (sub)trees (root-leaf) that are leaf nodes, in which case their similarity is 1.0 if their node labels are identical and 0.0 otherwise. Every tree is divided into some subtrees. So, the top-down traversal and bottom-up computation is recursively employed for every pair of subtrees.

Our algorithm has been incorporated into a subproject of the eduSource project [2]. A goal of this project is to search procurable learning objects for learners. The search results for a learner are represented as a percentage-ranked list of learning objects according to their similarity values with the learner's query. Figure 4 shows a segment of a learner tree. Another application of the algorithm is our Teclantic portal (http://teclantic.cs.unb.ca) which matches projects according to project profiles represented as trees.

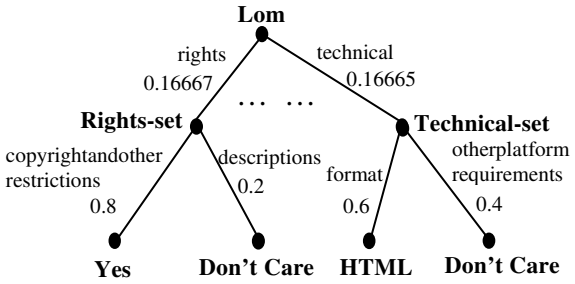

Figure 4. Segment of a learner tree.

3. TREE SIMPLICITY

During tree similarity computation, when a subtree in tree T_1 is missing in tree T_2 (or vice versa), we compute the simplicity of the missing subtree. Our tree simplicity measure takes into account not only the node degree (breadth) at each level and leaf node depth but also the arc weights for the recursive tree simplicity computation. Intuitively, the simpler the single subtree in T_1, the larger its simplicity and thus the larger its similarity to the corresponding empty tree in T_2. So, we use the simplicity as a contribution to the similarity of T_1 and T_2. Using our tree simplicity measure, the simplicity converges towards 0.0 when the tree is infinite. However, the simplest tree is a single node and we define that its simplicity is 1.0.

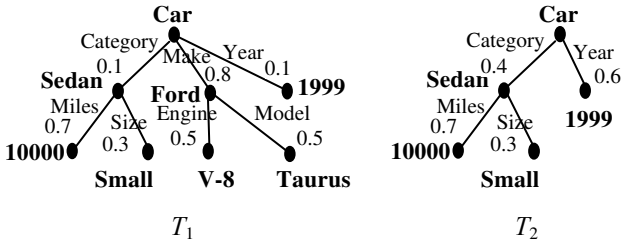

Figure 5. Arc-labeled, arc-weighted trees.

In tree T_1 of Figure 5, the subtree rooted at node "Ford" under the arc "Make" is missing in T_2. Our tree similarity algorithm passes this missing subtree to the *treeplicity* function to compute its simplicity. Figure 6 shows the pseudo code of this function.

***Input*:** The depth degradation index *i*. A single tree *T*.
Output: The simplicity value of *T*.
***Initialization*:** *treeplideg* = 0.5 //depth degradation factor
treeplicity (*i* ,*T*)
Begin
  If *T* only contains a single node **return** *i;*
  endif
  else
   sum=0;
   for (*j* =0; *j* < root node degree of *T*; *j*++);
    sum+=
      (weight of the j^{th} arc) ****treeplicity***(*i***treeplideg*, T_j);
        //T_j is the subtree under the j^{th} arc
   endfor
   TreeSimplicity =(1/root node degree of *T*) * sum;
   return TreeSimplicity;
  endelse
End.

Figure 6. Pseudo-code of the tree simplicity algorithm.

When calling *treeplicity* with a depth degradation index *i* and a single tree *T* as inputs, our simplicity measure is defined recursively to map an arbitrary single tree *T* to a value from [0, 1], decreasing with both the node degree at each level and leaf node depth. The recursion process terminates when *T* is a leaf node. For a (sub)tree, simplicity is computed by a recursive top-down traversal through its subtrees. Basically, the simplicity value of *T* is the sum of the simplicity values of its subtrees multiplied with arc weights from [0, 1], a subtree depth degradation factor ≤ 0.5, and the reciprocal of root node degree of a subtree that is from (0, 1].

For any subtree T_j underneath an arc l_j, we multiply the arc weight of l_j with the recursive simplicity of T_j. To enforce smaller simplicity for wider trees, the reciprocal of the node degree is used on every level. On each level of deepening, the depth degradation index *i* is multiplied with a global depth degradation factor *treeplideg* ≤ 0.5

and the result is the new value of i in the recursion. We always assume that the value of *treeplideg* is 0.5.

The smaller the *treeplideg* factor, the smaller the tree simplicity value. Based on the sum of the infinite decreasing geometric progression $\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^n}+\ldots$ being 1, we assume that *treeplideg* is 0.5 in order to enforce the smallest acceptable degradation through the recursion: this guarantees that the simplicity values of finite trees are always smaller than 1. For leaf nodes found on any level, the current i value is their simplicity.

Our tree simplicity algorithm is given as a recursive function shown below.

$$\check{S}(T) = \begin{cases} D_{\mathrm{I}} \cdot (D_{\mathrm{F}})^{d} & \text{if } T \text{ is a leaf node,} \\ \frac{1}{m}\sum_{\mathrm{j}=1}^{m} w_{\mathrm{j}} \cdot \check{S}(T_{\mathrm{j}}) & \text{otherwise.} \end{cases} \quad (1)$$

where,
$\check{S}(T)$: the simplicity value of a single tree T
D_{I} and D_{F}: depth degradation index and depth degradation factor
d: depth of a leaf node
m: root node degree of tree T that is not a leaf
w_{j}: arc weight of the j^{th} arc below the root node of tree T
T_{j}: subtree below the j^{th} arc with arc weight w_{j}

The initial value of D_{I} represents the simplicity value of the node with depth 0 (d = 0). Therefore, the simplicity of a tree that is a single node equals to D_{I}. Since an empty tree is meaningless in our tree simplicity algorithm, we define that the simplicity value of a "single-node" tree is 1.0 which also implies that D_{I} = 1.0. As mentioned before, the value of D_{F} is defined as 0.5 because of the employment of decreasing geometric progression.

When a tree horizontally and vertically grows infinitely, the value of m and d will be infinite. Therefore, both $(D_{\mathrm{F}})^{d}$ and $1/m$ approach 0.0. Since both w_{j} and $\check{S}(T_{\mathrm{j}})$ are in the interval [0, 1], equation (1) approaches 0.0 when m and d are infinite.

$$\lim_{\substack{m\to\infty \\ d\to\infty}} \check{S}(T) = 0 \quad (2)$$

Now, in order to clarify our tree simplicity algorithm, we show the computation of simplicity for an arbitrary tree (Figure 7) with the following illustrative example.

In Figure 7, we show the depth (d) and its corresponding depth degradation index (D_{I}) on the right hand side of tree T. According to equation (1), the simplicity of this tree is computed as follows.

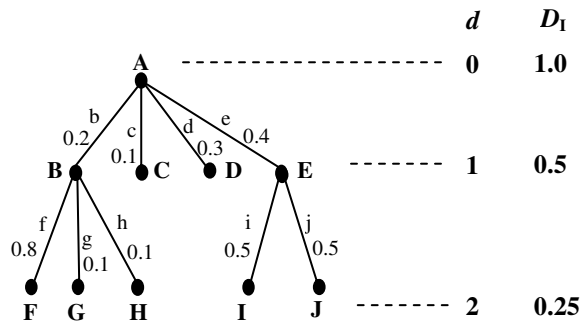

Figure 7. An illustrative tree T with d and D_{I}.

$$\check{S}(T) = \frac{1}{4}(0.2*\check{S}(T_{\mathrm{B}})+0.1*\check{S}(T_{\mathrm{C}})+ 0.3*\check{S}(T_{\mathrm{D}})+ 0.4*\check{S}(T_{\mathrm{E}})) \quad (3)$$

In equation (3), the denominator 4 is the degree of the root node of tree T. T_{B}, T_{C}, T_{D} and T_{E} represent the subtrees rooted at nodes B, C, D and E respectively. The simplicity values for subtrees (leaf nodes) T_{C} and T_{D} are their values of D_{I}, 0.5. We recursively use equation (1) for the simplicity computation of subtrees T_{B} and T_{E} and get equations (4) and (5), respectively.

$$\check{S}(T_{\mathrm{B}}) = \frac{1}{3}(0.8*\check{S}(T_{\mathrm{F}})+0.1*\check{S}(T_{\mathrm{G}})+ 0.1*\check{S}(T_{\mathrm{H}})) \quad (4)$$

$$\check{S}(T_{\mathrm{E}}) = \frac{1}{2}(0.5*\check{S}(T_{\mathrm{I}})+0.5*\check{S}(T_{\mathrm{J}})) \quad (5)$$

Simplicity values for subtrees (leaf nodes) T_{F}, T_{G}, T_{H}, T_{I} and T_{J} are their D_{I} value 0.25. Therefore, $\check{S}(T_{\mathrm{B}}) = \frac{0.25}{3}$ and $\check{S}(T_{\mathrm{E}}) = \frac{0.25}{2}$. Consequently, we obtain the value of $\check{S}(T)$ 0.06667.

4. ANALYSIS

In this section, we provide the theoretical and computational results to analyze the properties of the proposed tree simplicity measure given in Section 3. For this purpose, we consider balanced *k-ary* trees, although we do not restrict the degree and depth of each node in the tree. The theoretical results on balanced *k-ary* tree derived from equation (1) are a non-recursive function with two variables k and d. A *k-ary* tree is a normalized arc-labeled, arc-weighted tree according to Definition 4 such that the value of m in equation (1) is always a fixed k. The computational results indicate that tree simplicity values smoothly decrease with increasing k and d, which conform to our requirement. Here, both k and d are equal

to or greater than 1. When both of them are equal to 0, the tree is a single node which has been analyzed in Section 3.

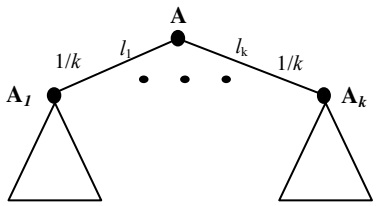

Figure 8. A balanced *k-ary* tree.

In a balanced *k-ary* tree shown in Figure 8, subtrees under nodes A_1 to A_k are shown by triangles since this tree may have arbitrary depth. All leaf nodes are at the same level; therefore, all leaf nodes have identical depth which is also the tree depth. All arc weights for sibling arcs are averaged. Therefore, we get arc weight $1/k$ for each arc.

For this special case, equation (1) is simplified as

$$\check{S}(T_{k-ary}) = (\frac{1}{k})^d \cdot D_F{}^d \cdot D_I \qquad (6)$$

where, $T_{k\text{-}ary}$ represents a *k-ary* tree. After incorporating the values of D_I and D_F, we get

$$\check{S}(T_{k-ary}) = (\frac{1}{k})^d \cdot 0.5^d \cdot 1.0 \qquad (7)$$

Although equation (2) reveals how the values of m (here, k) and d affect the tree simplicity values, we show the plots for equation (7) to get more intuitive understandings.

Plots generated by Matlab in Figure 9 represent the trend of tree simplicity values for varying d, when k is fixed to 1, 2 and 3. When k is fixed, equation (7) is an exponential function. According to equation (2), the tree simplicity should approach 0.0 when k and d approach infinity. In e-Business/e-Learning environments, it is most likely that trees with very big depth would not arise. For example, in Figure 4, the depth of the given tree is 2. It is evident that we can show the complete tree with this depth for a learner. Therefore, we consider changes to the value of d from 1 to 40 in the subsequent figures. This applies to the value of k as well.

Each plot in Figure 9 has the same trend: tree simplicity decreases when the depth (d) of the tree increases. However, for the same d, smaller the value of k, bigger is the tree simplicity. Furthermore, bigger the value of k, faster the plot approaches 0.0. This agrees with our analysis of equation (2).

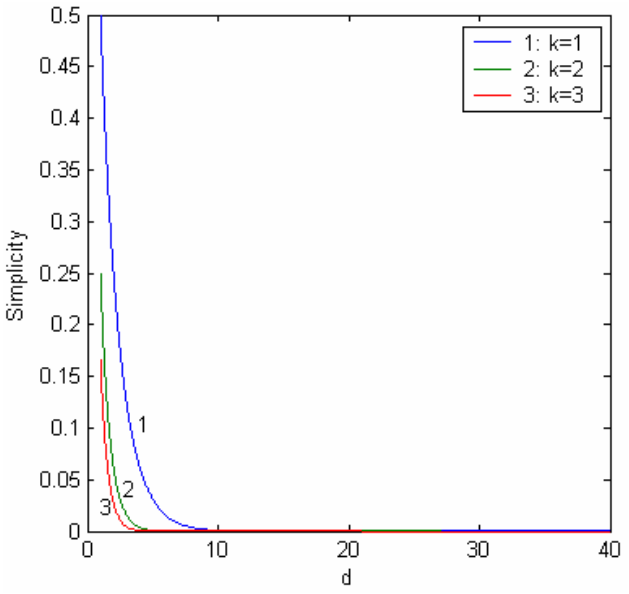

Figure 9. Simplicity as a function of d.

The plots in Figure 10 represent the trend of tree simplicity values for varying k, when the value of d is fixed to 1, 2 and 3. When d is fixed, equation (7) is a polynomial function.

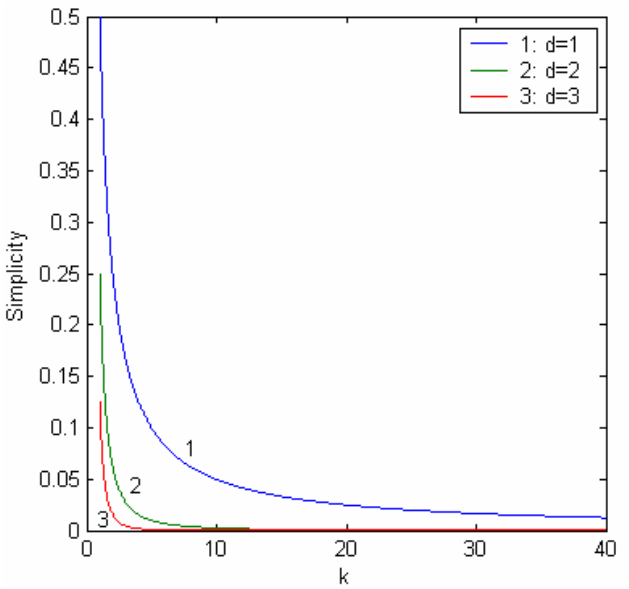

Figure 10. Simplicity as a function of k.

Plots in Figure 10 have trends similar to those in Figure 9. The tree simplicity decreases when the value of k increases. However, for the same k, smaller the value of d, bigger is the tree simplicity. Furthermore, bigger the value of d, faster the plot approaches 0.0.

The maximum simplicity value for a *k-ary* tree is 0.5 when both *k* and *d* are 1 (*1-ary* tree). An example tree of this case is shown in Figure 11.

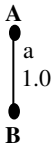

Figure 11. The *k-ary* tree when both *k* and *d* equal to 1.

However, both *k* and *d* in equation (7) may have arbitrary values. In our computation, we consider every combination when both of them vary from 1 to 40. This means that the maximum root node degree (*k*) and the maximum depth (*d*) of the tree are equal to 40. Figure 12 shows the 3-dimensional plot of equation (7). It is easy to find that even when the values of *k* and *d* approach 40 (not infinity), the simplicity values are very close to 0.0. This agrees with our requirement that simpler the tree, bigger is the simplicity.

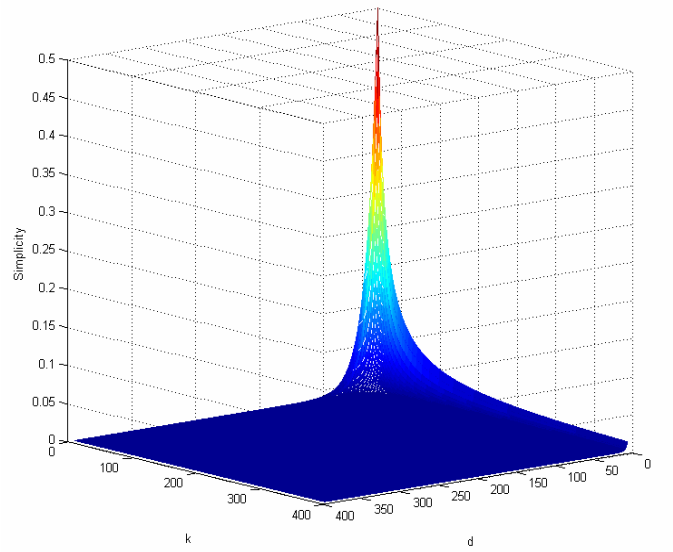

Figure 12. Simplicity as a function of *d* and *k*.

5. CONCLUSION

We have used arc-labeled, arc-weighted trees to represent the product/service descriptions of buyers/ sellers in e-Business/e-Learning environments. We have formalized the definitions of our trees and conducted experiments on balanced *k-ary* tree. The computational results on this special case agree with our requirement.

Our trees are arbitrary trees without restrictions on node degree and depth. A simpler tree possesses a bigger simplicity value. The simplest tree in our algorithm is a single node and its simplicity is defined as 1.0. The most complex tree is an infinite tree. The simplicity value for an infinite tree approaches 0.0. We have formalized our tree simplicity measure as a recursive function.

In future, we will further analyze more on the properties of the general recursive function of our tree simplicity measure and justify our approach not only for *k-ary* trees.

ACKNOWLEDGEMENTS

We thank Joseph D. Horton, University of New Brunswick, for discussions about tree definitions and tree simplicity formalizations. We also thank the NSERC for its support through discovery grants of Virendra C. Bhavsar and Harold Boley.

REFERENCES

[1] V. C. Bhavsar, H. Boley and L. Yang, "A weighted-tree similarity algorithm for multi-agent systems in e-business environments," *Computational Intelligence*, vol.20, no.4, pp.584-602, 2004.

[2] H. Boley, V. C. Bhavsar, D. Hirtle, A. Singh, Z. Sun and L. Yang, "A match-making system for learners and learning objects," *Learning & Leading with Technology,* International Society for Technology in Education, Eugene, Oregon, 2005 (to appear).

[3] A. Chavez and P. Maes, "Kasbah: An agent marketplace for buying and selling goods," Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, London, pp.75-90, 1996.

[4] T. Liu and D. Geiger, "Approximate tree matching and shape similarity," Proceedings of the Seventh International Conference on Computer Vision, Kerkyra, pp.456-462, 1999.

[5] S. Lu, "A tree-to-tree distance and its application to cluster analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2), pp.219-224, 1979.

[6] D. Shasha, J. Wang and K. Zhang, "Exact and approximate algorithm for unordered tree matching," *IEEE Transactions on Systems*, *Man and Cybernetics*, vol.24, no.4, pp.668-678, 1994.

[7] J. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and K. M. Currey, "An algorithm for finding the largest approximately common substructures of two trees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.20, pp.889-895, 1998.

[8] L. Yang, M. Ball, V. C. Bhavsar and H. Boley, "Weighted partonomy-taxonomy trees with local similarity measures for semantic buyer-seller matchmaking," Proceedings of 2005 Workshop on Business Agents and the Semantic Web, Victoria, Canada, pp. 23-35, May 8, 2005.