A Weighted-Tree Similarity Algorithm for Multi-Agent Systems in E-Business Environments

Virendra C. Bhavsar[1], Harold Boley[2], and Lu Yang[1]

[1] Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

[2] Institute for Information Technology e-Business
National Research Council of Canada
Fredericton, NB, E3B 9W4
Canada

Abstract:
A tree similarity algorithm for match-making of agents in e-business environments is presented. Product/service descriptions of seller and buyer agents are represented as node-labelled, arc-labelled, arc-weighted trees. A similarity algorithm for such trees is developed as the basis for semantic match-making in a virtual marketplace. The trees are exchanged using a serialization in Object-Oriented RuleML. A corresponding Relfun representation is employed to implement the similarity algorithm as a parameterised, recursive functional-logic program. Results from our experiments are found to be meaningful for e-business/e-learning environments. The algorithm can also be applied in other environments wherein weighted trees are used.

Keywords/phrases: multi-agent system, e-business, e-learning, similarity measure, arc-labelled trees, arc-weighted trees, Object-Oriented RuleML, Relfun.

1. Introduction

With the increasing adoption of e-business, seller-buyer message exchange will be increasingly conducted using advanced technologies from the Semantic Web and Web Services. In the emerging multi-agent virtual marketplace, seller and buyer agents will engage into e-business activity basically as follows: Using a semantic representation for the message content, sellers advertise their product/service offers and buyers issue product/service requests so that a match-making procedure can pair semantically similar offer and request content, after which the paired agents can carry out negotiations and finalize their transactions. The present study employs a multi-agent architecture similar to Agent-based Community Oriented Routing Network (ACORN) [Marsh et al. 2003] as the

foundation for semantic match-making [Sycara et al. 2001] and focuses on its central similarity algorithm for comparing RuleML-like message contents [Boley 2003].

In a multi-agent system like ACORN [Marsh et al. 2003] a set of key words/phrases with their weights is used for describing the information an agent is carrying or seeking. Product/service advertising and requesting can be realized on top of sets of weighted key words/phrases. However, such a flat representation is limited in that it cannot represent tree-like product/service descriptions. Therefore, to allow more fine-grained interaction between agents, we propose to represent descriptions in the form of weighted trees. However, because of the many variants and refinements in modern products/services, a total match will rarely be possible; so partial matches, embodied in some measure of similarity, are needed. While some variety of trees has already been used in multi-agent systems to describe the content part of messages, tree similarity matching for such content representations has not been studied to our knowledge. On the other hand, many other flavours of similarity have been explored in Utility Theory and AI, in particular in Case-Based Reasoning [Richter, 2001], some of which should be combinable with our tree similarity.

Often (e.g. in Lisp, Prolog, and plain XML), *node-labelled, arc-ordered trees* have been employed for knowledge representation. In this paper, following Object Oriented (OO) modelling, F-Logic [Kifer et al., 1995], and the Resource Description Framework (RDF) [Lassila and Swick, 1999], we propose *node-labelled, arc-labelled (hence arc-unordered) trees*, where not only node labels but also arc labels can embody semantic information. Furthermore, our trees are *arc-weighted* to express the importance of arcs.

For the uniform representation and exchange of product/service trees we use a weighted extension of Object-Oriented RuleML [Boley 2003]. In Weighted Object-Oriented RuleML, besides 'type' labels on nodes, there are 'role' labels on arcs, as in the alternating (or 'striped') syntax of RDF graphs (which quite often are trees or can be unrolled into trees); we assume here arc labels to be unique on each level, i.e. every pair of outgoing arcs of a given node must use different arc labels. Arc weights are numbers taken from the real interval [0,1] and employed as a general measure of relevance for branches, neutral w.r.t. any specific probabilistic, fuzzy-logic, or other interpretation.

Based on this representation we develop a similarity measure as a recursive function, *treesim*, mapping any (unordered) pair of trees to a value in the real interval [0,1], not to be confused with the above arc weights, also taken from that interval. This will apply a co-recursive 'workhorse' function, *treemap*, to all pairs of subtrees with identical labels. For a branch in one tree without corresponding branch in the other tree, a recursive simplicity function, *treeplicity*, decreases the similarity with decreasing simplicity. These functions are implemented in the functional-logic language Relfun [Boley 1999].

The paper is organized as follows. The architecture of a multi-agent system that carries out match-making of buyer and seller agents is outlined in the following section. Several examples of *node-labelled, arc-labelled, arc-weighted trees* and symbolic tree representations in OO RuleML and Relfun are presented in Section 3. Section 4 gives an

application of the multi-agent system for e-Learning environments. Many issues that need to be addressed while developing a similarity measure for our trees are discussed in Section 5. This section also presents our algorithm for computing the similarity of trees. Section 6 presents similarity results obtained with the Relfun implementation (included in the Appendix) of our algorithm. Finally concluding remarks are given in Section 7.

2. Multi-agent Systems

Agent systems have been proposed and exploited for e-business environments (see for example, [Yang et al. 2000]). In such systems, Buyer agents deal with information about the items their owners want to buy and corresponding price they can accept, while seller agents deal with information about the items they sellers want to sell and the price they ask. Therefore, buyer agents and seller agents need to be matched for similarity of their interests and and subsequently they can carry out negotiations. Furthermore, they might need another agent acts as the middle man. These multiple agents working together toward form a multi-agent system.

2.1 The architecture

The Agent-based Community Oriented Routing Network (ACORN) is a multi-agent architecture that can manage, search, filter information across the network. Among the applications of ACORN, e-business is a very important one. We outline the architecture of an ACORN-like multi-agent system that views the information as agent. This multi-agent system uses mobile agents.

Figure 1 shows an overview of the ACORN-like multi-agent system architecture. The multi-agent system has the structure of a Client/Server system. Clients provide the interface to users. Users create and organize agents, such as buyer and seller agents, create and modify their profiles which describe the interests of users. As can be seen, user profiles are stored on the server side and the incoming agents are processed according to user profiles. The Main Server provides mobility to agents. When an agent reaches a new site, the agent can visit users and communicate with other agents at a central meeting point, named *Cafe*.

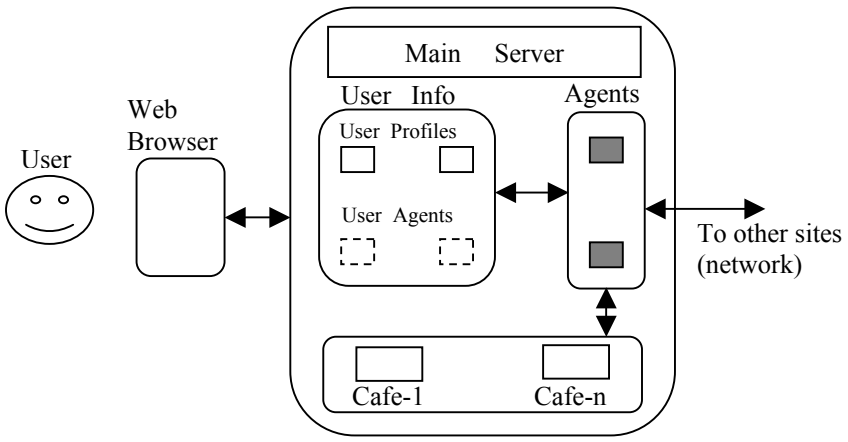


Figure 1. ACORN-like multi-agent system [Marsh et al. 2003].

The primary distribution of information in the multi-agent system is carried out by its Moblie Agents, or InfoAgents. An InfoAgent represents a piece of information its creator requires. In our application, these InfoAgents carry tree-form information during their journey accross the network. In fact, InfoAgents do not know what information they bring with them, what they carry is a link to the information and a set of metadata elements that describe it. Figure 2 shows the contents of InfoAgents.

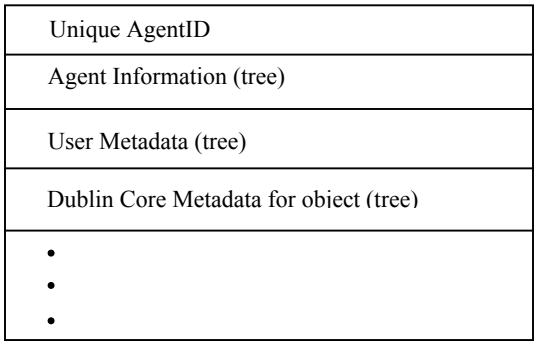


Figure 2. Contents of InfoAgents.

2.2 Match-making in the Cafe

One very important aspect of the multi-agent system is the exchange of the information between agents. A Cafe provides a common place for such information exchange. Figure 3 shows a Cafe with buyer and seller agents. The buyer and seller agents do not communicate with each other directly, but communicate through the Cafe Manager. The Cafe Manager is responsible for mediating between them. For example, if two agents enter the Cafe, one representing a car seller who wants to sell a Ford Explorer that was made in 2002 (see Figure 4 (a)), another agent representing a car buyer that wants to buy a Ford Explorer made in 1999 (see Figure 4 (b)), the Cafe is the place where they exchange their information and the similarity of the information is also computed by the Cafe Manager.

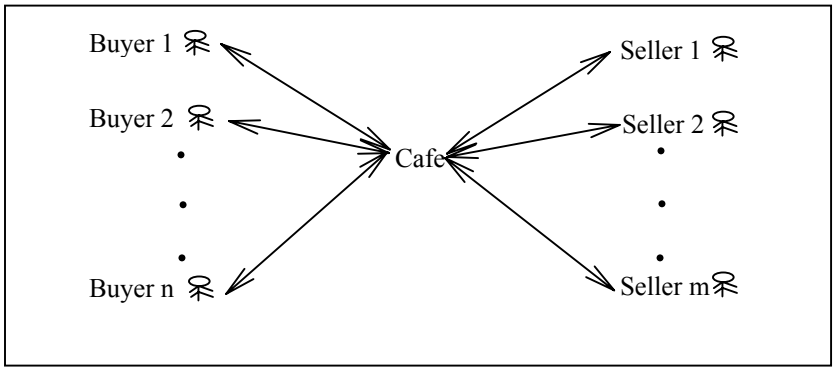
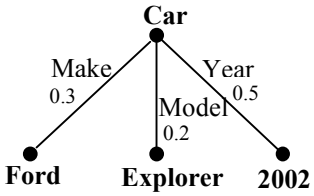


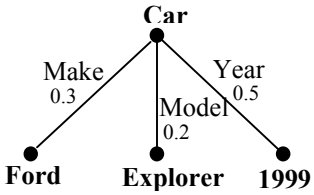
Figure 3. Match-making of buyer and seller agents in the Cafe (adopted from [Marsh et al. 2003]).

3. Tree Representations

Various representations of trees and their matching are possible. To simplify the algorithm, we assume our trees here are kept in a normalized form: the arcs will always be labelled in lexicographic (alphabetical) order. Two very simple examples that describe cars are illustrated in Figure 4 (a) and Figure 4 (b). To emphasize the difference between arc labels and node labels, node labels will always be bold-faced.



(a) Newer Ford Car



(b) Older Ford Car

Figure 4. Arc-labeled arc-weighted trees describing cars.

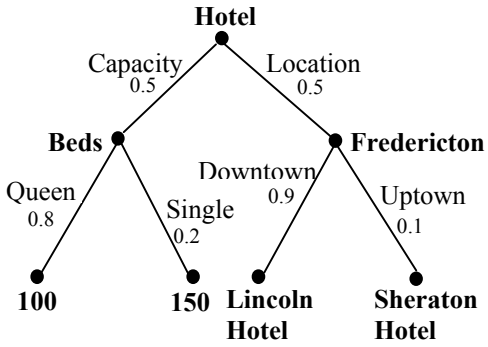


Figure 5. A tree describing hotel information.

In order to be applicable to real world description refinement, we do not limit the depth or breadth of any subtree. So, our trees can represent arbitrarily complex descriptions. For example, the tree in Figure 5 represents information about a hotel. While the tree in Figure 5 is a binary tree, it might later be refined in a non-binary way. A more complex tree that describes a job bank is shown in Figure 6.

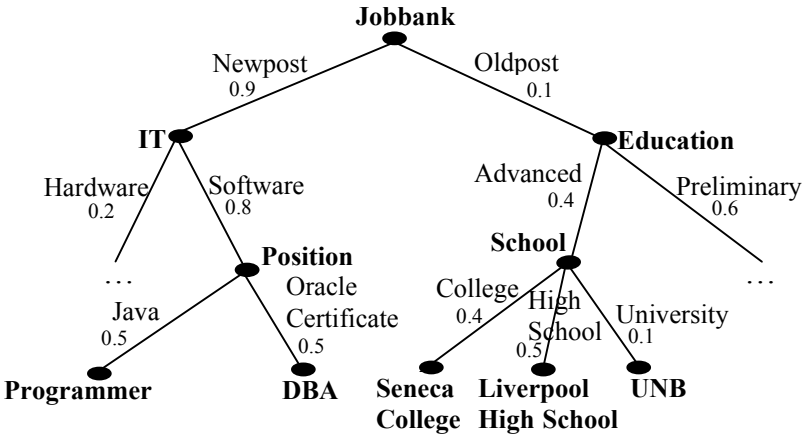


Figure 6. A tree describing a job bank.

Capturing the characteristics of our arc-labelled, arc-weighted trees, Weighted Object-Oriented RuleML, a RuleML version for OO modelling, is employed to serialize these various trees for Web-based agent interchange. The XML child-subchild structure reflects the shape of our normalized trees and XML attributes are used to serialize the arc labels and weights. So, the tree in Figure 4 (b) can be serialized as shown in Figure 7 (a).

In Figure 7 (a), the complex term (cterm) element serializes the entire tree and the _opc role leads to its root-node label, car. Each child element **_r** is a metarole, where the start tag contains the names and weights of arc labels as XML attributes **n** and **w**, respectively, and the element content is the role filler serializing a subtree (incl. a leaf). Consider the first **_r** metarole of the cterm as an example. The attribute **n** has the value "Make", describing the make name of the car. The other attribute **w**, with the value "0.3", endows the "Make" branch with its weight. The content between the **_r** tags is an ind(ividual constant) serializing a leaf node labelled "Ford". Such weights might have different interpretations in different cases. For example, from a buyer's point of view the "Make" weight may mean that the importance of the make of this car is 0.3 compared to other subtrees such as its "Model" and "Year", which have importance "0.2" and "0.5", respectively.

```
<cterm>
  <_opc><ctor>Car</ctor></_opc>
  <_r n="Make" w="0.3"><ind>Ford</ind></_r>
  <_r n="Model" w="0.2"><ind>Explorer</ind></_r>
  <_r n="Year" w="0.5"><ind>1999</ind></_r>
</cterm>
```

(a) Tree serialization in OO RuleML.

```
cterm[ -opc[ctor[car]],
       -r[n[make],w[0.3]][ind[ford]],
       -r[n[model],w[0.2]][ind[explorer],
       -r[n[year],w[0.5]][ind[1999]]
      ]
```

(b) Tree representation in Relfun.

Figure 7. Symbolic tree representation.

For the purpose of our Relfun implementation, Weighted OO RuleML serializations such as Figure 7 (a) become Relfun structures such as Figure 7 (b). The correspondence is quite obvious, but we have to use, e.g., **–r** to denote a metarole because any symbol beginning with a "_" (like any capitalized symbol) in Relfun is a variable.

4. E-Learning Scenario of a Multi-agent System

Our multi-agent system architecture can be instantiated for an e-Learning scenario. In e-Learning, the buyer agent represents a learner and the seller agent provides a course, as shown in Figure 8.

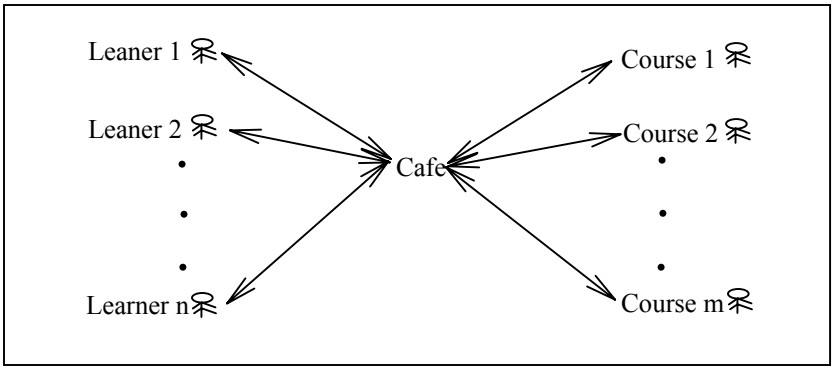
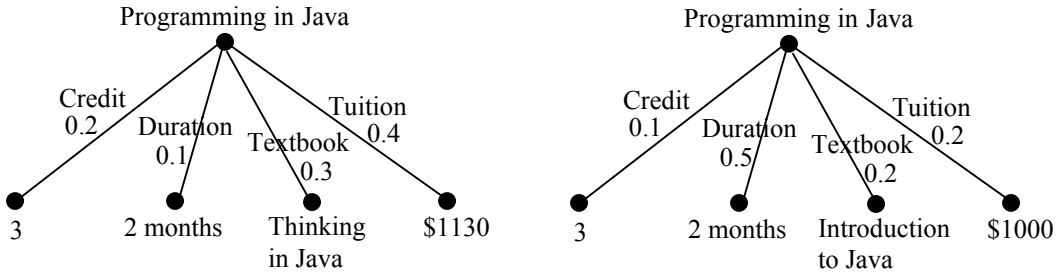


Figure 8. Instantiating Figure 3 for e-Learning.

The information that learner and course agents carry is again represented as trees. For example, in Figure 9 (a) the agent learner1 requests information of a course "Programming in Java", while in Figure 9 (b), the agent course6 offers information about such a course. The Cafe Manager matches these trees of both agents and computes their similarity based on the algorithm introduced in Section 5.2. The Café Manager computes such similarity measures for learner1 and all m course agents, ranks them, and then the learner agent decides which course to choose. Analogously, the Café Manager considers the other (n-1) learner agents and matches them with course agents.



(a) Tree of learner1 agent. (b) Tree of course6 agent.

Figure 9. Leaner and course trees.

5. Similarity of Trees

When developing a tree similarity measure, several issues have to be tackled because of the general shape of trees, their recursive nature, and their arbitrary of sizes. The similarity function maps two such potentially very complex trees to a single real number ranging between 0 and 1. These issues are discussed in the first subsection with examples. The similarity algorithm is outlined in the subsequent subsection.

5.1 Issues

In this subsection, we present seven groups of characteristic sample trees that explore relevant issues for developing a similarity measure. Section 6 gives similarity values of these trees using the algorithm given in Section 5.2.

Example 1:

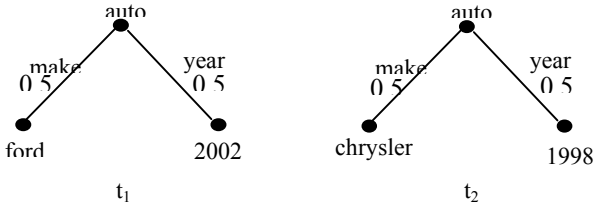
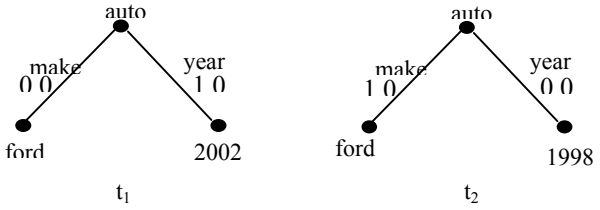


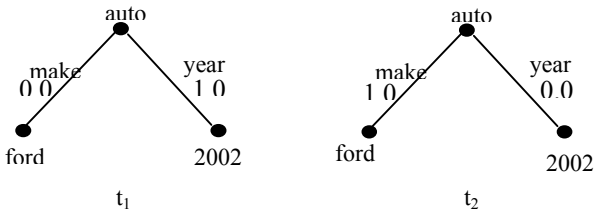
Figure 10. Two trees with mismatching leaves.

In Figure 10, tree t_1 and tree t_2 have the same root node label "auto". But the node labels of subtrees (leaf nodes) are all different. In this example, although the two leaf nodes have numeric values, we do not carry out any arithmetic operation on these values to find out their closeness. Therefore, the similarity of these trees could be defined as zero. However, we have decided to award a user specifiable similarity 'bonus' reflecting the identity of the root nodes.

Example 2:



(a) Trees with opposite extreme weights.



(b) Trees as in (a) but with identical leaves.

Figure 11. Trees with opposite branch weights.

This example can be viewed as a modification of Example1. In Figure 11 (a), tree t_1 and tree t_2 have one identical subtree, "ford," and therefore the similarity of these two subtrees can be considered as 1. However, we have to note that the weights of the arcs labelled "make" are 0.0 versus 1.0. This might indicate that the agent of tree t_1 does not care about the maker of the automobile, while the agent of tree t_2 *only* cares about the

maker of the automobile. Thus, the similarity of the "make" branches should not be 1. We have chosen to average the weights of the branches using the arithmetic mean, i.e. 1*(0.0 + 1.0)/2 = 0.5. We could have chosen to use the geometric mean, which would give zero branch similarity. However, we think that the branch similarity should be nonzero for identical subtrees.

The "year" branches lead to different leaf nodes, so we consider the similarity as zero independent of the branch weights and the branch similarity is 0*(1.0 + 0.0)/2 = 0.

Thus, for the entire trees of this example, we obtain the similarity as follows:
1*(0.0 + 1.0)/2 + 0*(1.0 + 0.0)/2 = 0.5.

In Figure 11 (b), tree t_1 and tree t_2 are the same as in (a) but have identical leaves.
In this case, the trees are exactly the same except for their weights. In an e-business environment this can interpreted as follows. While the seller and buyer agents have attached opposite branch weights to reflect their subjective preferences, their autos represented are exactly the same. This implies that the similarity of the two trees should be equal to 1.0. Indeed, we obtain the similarity analogously to the case of (a), as follows:
1*(0.0 + 1.0)/2 + 1*(1.0 + 0.0)/2 = 1.0.

Example 3:

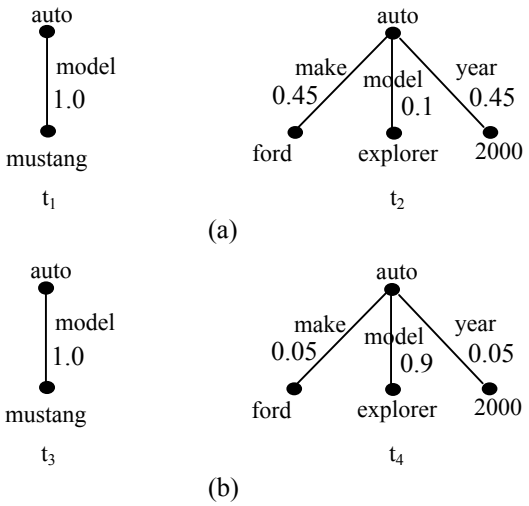


Figure 12. Tree pairs only differing in arc weights.

Figures 12 (a) and (b) represent two pairs of trees only differing in the weights of the arcs of t_2 and t_4. In Figure 12 (a), t_1 has only one arc with label "model," which also occurs in tree t_2. But their leaf node labels are different. The situation of Figure 12 (b) is the same as Figure 12 (a) except that the weight of the label "model" is 0.9 in tree t_4, while it is 0.1 in tree t_2. On cursory look, the similarity of both pairs of trees should be identical because the leaf node differences between each pair of trees are identical. However, we should not overlook the contribution of the weights. In tree t_2, the weight of arc-label "model" is much less than that in tree t_4. Thus, during the computation of similarity, the weight of the arc labelled "model" should make a different contribution to the similarity: the importance of the "mustang-explorer" mismatch in Figure 12 (a) should be much lower

than the same mismatch in Figure 12 (b). So, if we denote the similarity between trees t_i and t_j as $S(t_i, t_j)$, then $S(t_1, t_2) > S(t_3, t_4)$.

Example 4:

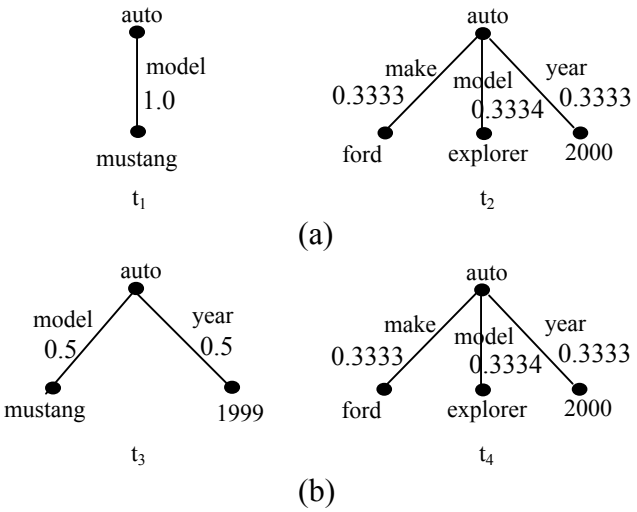


Figure 13. Tree pairs with left-tree refinement.

In Figure13 (a), tree t_1 only has one arc, while tree t_3 in Figure 13 (b) has two arcs. Tree t_2 and tree t_4 are identical. However, in both pairs of trees, for identical arc labels, the leaf nodes are different. For example, both tree t_3 and tree t_4 have are-label "year", but their node labels are "1999" and "2000" respectively. Tree t_1 only has one different subtree compared to tree t_2, but tree t_3 has two different subtrees compared to tree t_4. Therefore, $S(t_1, t_2) > S(t_3, t_4)$.

Example 5:

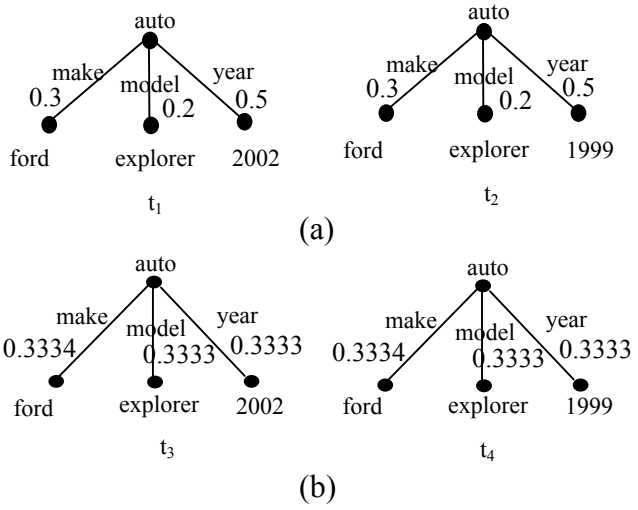


Figure 14. Tree pairs with the same structure.

In Figures 14 (a) and (b), trees t_1 and t_2 and trees t_3 and t_4 have almost the same structure except one pair of node labels "2002" and "1999". So, we are sure that node labels and arc labels cannot make $S(t_1, t_2)$ and $S(t_3, t_4)$ different. But for the arc-label "year", Figure

14 (a) and Figure 14 (b) have different weights which should lead to $S(t_1, t_2) < S(t_3, t_4)$, because of the higher mismatch of weights in Figure 14 (a) compared to Figure 14 (b).

Example 6:

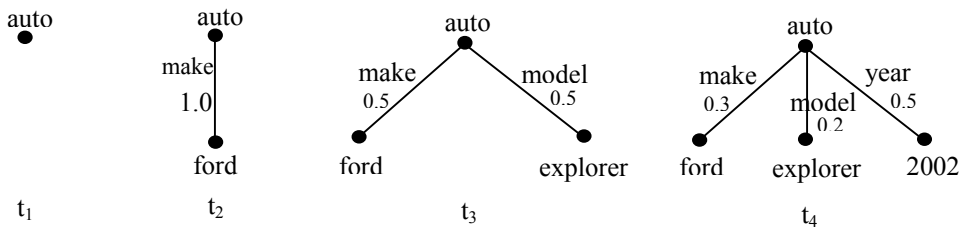


Figure 15. Trees with an increasing number of subtrees.

Figure 15 gives four sample trees with an increasing number of subtrees. Tree t_1 is an empty tree while the other trees have one or more branches. Intuitively, when comparing tree t_1 to the other tress, the more complex a tree, i.e. less simple the tree, the less should be the similarity. Thus, we would expect that $S(t_1, t_2) > S(t_1, t_3) > S(t_1, t_4)$.

Example 7:

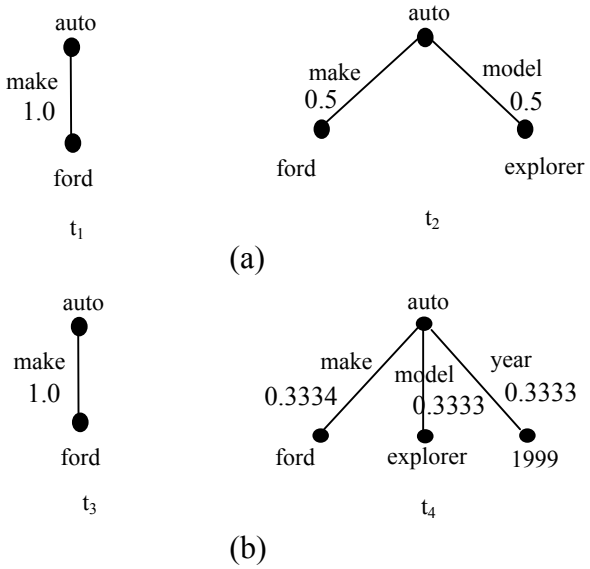


Figure 16. Tree pairs with right-tree refinement.

In Figure 16, trees t_1 and t_3 are identical. Note also that t_2 is simpler than t_4, i.e. the simplicity of t_2 is greater than that of t_4. For this example, we make the following observations: (i) The similarity of the "make" branches in (a) is greater than that of those in (b) due to the greater weight of the matching "make" label for t_2. (ii) The dissimilarity of the remaining branches in t_2 (compared to t_1) is less than that of t_4 (compared to t_3), because of the greater simplicity of the "model" branch of t_2, compared to the combined simplicities of the "model" and "year" branches of t_4. Clearly, (i) and (ii) together imply $S(t_1, t_2) > S(t_3, t_4)$.

5.2 The Algorithm

The similarity algorithm is defined recursively, by case analysis. Here we explain its main functions *treesim*, *treemap*, and *treeplicity* in semi-formal English. Using the RuleML cterm representations introduced in Figure 7, the appendix gives the full formal definition in Relfun [Boley 1999]. The call structure is as follows, where [bracketed] parameters are distinguished from (parenthesised) arguments: The main function treesim[N,A](t,t') calls the 'workhorse' function treemap[N,A](l,l'), which co-recursively calls treesim; treemap also calls treeplicity(i,t).

treesim[N,A](t,t')

The treesim recursion is terminated by two (sub)trees t and t' that are leaf nodes or empty trees, in which case their similarity is 1 if their node labels are identical and 0 otherwise.

If only one of the (sub)trees is a leaf node, the other being a non-leaf node (including an empty tree), the leaf node is 'converted' into an empty tree taking over the leaf node label, and the similarity of the resulting trees is computed.

The similarity of two (sub)trees, t and t', including leaves, with different node labels is defined to be 0.

If two non-empty (sub)trees have identical root node labels, their similarity will be computed via treemap by a recursive top-down (root-to-leaf) traversal through the subtrees, t_i and t'_i, that are accessible on each level via identical arc labels l_i; on a given level, the index i ─ implicit in the recursion of our implementation ─ accesses the arc labels in lexicographic (left-to-right) order. A node-identity fraction N ─ a 'bonus' value from [0,1] ─ is added to the complementary fraction (1-N) of this subtree comparison (in this paper, the value of N is consistently assumed to be 0.1).

treemap[N,A](l,l')

The treemap function recursively compares two lists, l and l', of weighted arcs on each level of identical-root fan-out identified by treesim.
In general, the arcs can carry arbitrary weights, w_i and w'_i from [0,1]. The weights are averaged using the arithmetic mean, $(w_i + w'_i)/2$, and the recursively obtained similarity s_i of trees t_i and t'_i ─ adjusted to $A(s_i)$ by an arc function A ─ is multiplied by the averaged weight. Finally, on each level the sum of all such weighted adjusted similarities, $A(s_i)(w_i + w'_i)/2$, is divided by the sum of all averaged weights:

$$\sum (A(s_i)(w_i + w'_i)/2) \,/\, \sum (w_i + w'_i)/2 \qquad (1)$$

In general, A can be any function from [0,1] to [0,1], but we assume $A(x) \geq x$ to compensate similarity degradation for nested trees. Two examples are the identity function, $A(s_i) = s_i$, and the square root, $A(s_i) = \sqrt{s_i}$. This arc-similarity-adjusting function also permits different (seller, buyer) agents to use different 'yardsticks', even

when comparing the same pair of trees. For example, for agents biased towards higher similarities, the square root function is preferable to the identity function.

In the special case that weights on some level of both trees add up to 1, the denominator of formula (1) becomes 1. Hence, if we require that the weights on each level of both trees add up to 1 (a reasonable assumption used throughout this paper), formula (1) will be simplified:

$$\sum (A(s_i)(w_i + w'_i)/2) \qquad (2)$$

Suppose that, on a given level, for an arc label l_i in t there exists no identical arc label in t' (or vice versa). In this case, the algorithm uses treeplicity to compute the similarity based on the simplicity of an 'extracted' tree consisting of a newly generated root and the subtree reachable via l_i. Intuitively, the simpler the tree extracted from t, the larger its similarity to the corresponding empty tree extractable from t'. So, we use the simplicity as a contribution to the similarity of t and t'.

treeplicity(i,t)

The simplicity measure is defined recursively to map an arbitrary single tree t to a value from [0,1], decreasing with both the tree breadth and depth. The reciprocal of the tree breadth on any level is used as the breadth degradation factor. The depth degradation value i ─ initialized with 1-N by treemap ─ is multiplied by a global factor treeplideg ≤ 0.5 (= 0.5 will always be assumed here) on each level deepening. The current i value then becomes the simplicity for a leaf node or an empty tree.

6. Experimental results

In this section we consider the examples discussed in Section 5.1 and use our implemented algorithm in Section 5.2 to obtain similarity values. Table 1 summarizes the similairity values obtained for all tree pairs. In our experiments we have set the node-identity parameter to 0.1.

For Example 1, the complete mismatch of coresponding leaf nodes leaves only the user specified similarity 'bonus' of 0.1, reflecting the identity of the root nodes contributing to the similarity. The similarities of Example 2 (a) and (b) are obtained as 0.5 and 1.0, respectively, as desired (see Section 5.1).

Since the importance of the "mustang-explorer" mismatch in Figure 12 (a) of Example 3 should be much lower than the same mismatch in Figure 12 (b), we expected $S(t_1, t_2) > S(t_3, t_4)$. It turns out that $S(t_1, t_2)$ is more than two times bigger than $S(t_3, t_4)$.

For Example 4 we anticipated that $S(t_1, t_2) > S(t_3, t_4)$. Our program gives consistent results. Our results confirm the expected result of $S(t_1, t_2) < S(t_3, t_4)$ for the Example 5.

Example	Tree	Tree	Similarity
Example 1	t_1	t_2	0.1
Example 2	t_1	t_2	(a) 0. 5
	t_1	t_2	(b) 1.0
Example 3	t_1	t_2	0.2823
	t_3	t_4	0.1203
Example 4	t_1	t_2	0.2350
	t_3	t_4	0.1675
Example 5	t_1	t_2	0.55
	t_1	t_3	0.7000
Example 6	t_1	t_2	0.3025
	t_1	t_3	0.3025
	t_1	t_4	0.3025
Example 7	t_1	t_2	0.8763
	t_3	t_4	0.8350

Table 1.Experimental results of the examples in Section 5.1.

The Example 6 consists of four trees and we expected $S(t_1, t_2) > S(t_1, t_3) > S(t_1, t_4)$. However, we observe that we obtain $S(t_1, t_2) = S(t_1, t_3) = S(t_1, t_4)$. This can be explained as follows. For a branch of depth 1, the treeplicity function gives a simplicity value of 0.45. For the pair (t_1, t_2), we get simplicity value of 0.45 for the missing branch in t_1, which is weighted 1.0. For the pair (t_1, t_3), we get simplicity value of 0.45 twice for the two missing branches in t_1, which are weighted 0.5 each, resulting in the same overall simplicity of 0.45, hence in $S(t_1, t_2) = S(t_1, t_3)$. A similar explanation can be given for $S(t_1, t_3) = S(t_1, t_4)$. We expected $S(t_1, t_2) > S(t_3, t_4)$ for Example 7. Our experimental result agrees.

In our experiments we use the two arc-similarity-adjusting functions discussed in section 5.2 as similarity parameters to test if the results are intuitive and reasonable. Here, we give a somewhat more complex example to illustrate the results of our experiments. Two three-level trees are shown in Figure 17. The differing parts of these two trees are marked by dashed circles and triangles. In these two trees, we assign approximately equal weights to every arc that belongs to the same parent to permit easy validation of the similarity results.

If we use $A(s_i) = s_i$, we get: $S(t_1,t_2)=0.5950$; while if we use $A(s_i) = \sqrt{s_i}$, we obtain: $S(t_1,t_2)=0.7611$. In this example, the root node label "jeep" of tree t_1 (marked with a dashed circle) is different from its corresponding node labelled "van" (marked with a dashed triangle) in tree t_2, so the similarity of these two trees is 0. If we change any one of these two nodes to make them identical, we obtain $S(t_1,t_2) = 0.6359$, when $A(s_i) = s_i$; $S(t_1,t_2) = 0.8148$, when $A(s_i) = \sqrt{s_i}$. The reason for this increase of the similarity is the nonzero contribution of the node-identity fraction 0.1 and the simplicity of the tree rooted at node "jeep" marked with a dashed circle.

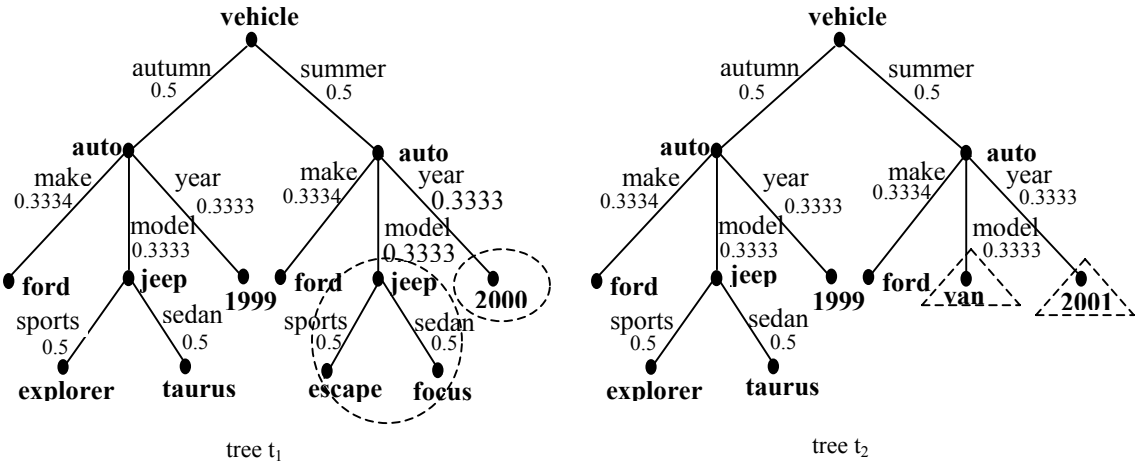


Figure 17. Applying two arc-adjusting functions to a pair of trees.

The example above contains two relatively similar trees. In Figure 18, two trees that are quite different are shown. The biggest difference between these two trees is that the left part of tree t_1 is missing compared to tree t_2. The other two different parts are marked with a dashed circle and a dashed triangle like before. When using $A(s_i) = s_i$, $S(t_1,t_2) = 0.5894$, and $S(t_1,t_2) = 0.6816$, when $A(s_i) = \sqrt{s_i}$.

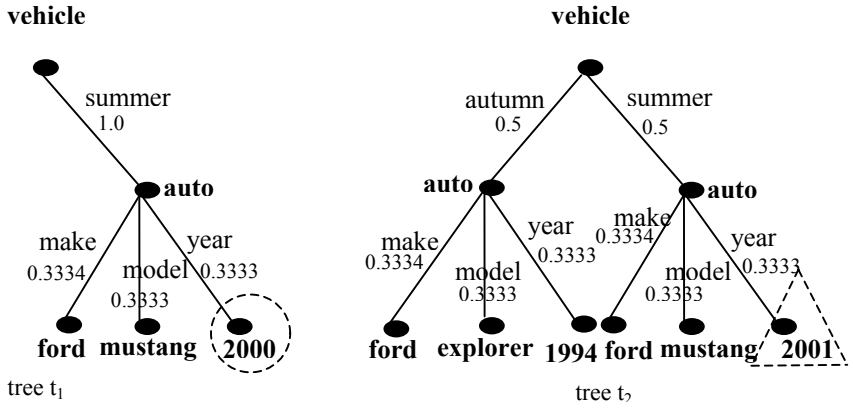


Figure 18. 'Summer' tree compared to balanced tree.

7. Conclusion

The outlined multi-agent system is an architecture where agents represent pieces of information and interact with each other. We focus on the information these agents carry and a similarity measurement over this information. In order to make the interaction between agents more meaningful and fine-grained, we chose trees to represent this information. More precisely, these trees are *node-labelled*, *arc-labelled*, and *arc-weighted*. The characteristics of these kinds of trees led us use complex term of Object-Oriented RuleML and XML attributes within them to represent arc labels and arc weights.

Our tree similarity algorithm, as part of semantic match-making, computes the similarity of subtrees in a recursive way. It can be parameterized by different functions to adjust the similarity of these subtrees. This gives agents the option to tailor their similarity measure for trees based on their subjective preferences. The appendix gives the full definition of the algorithm in Relfun. This executable functional-logic specification has proved to be a flexible test bed for our experiments. Our experiments have given us meaningful results for e-business/e-learning environments. The algorithm can also be applied in other environments wherein weighted trees are used.

This multi-agent system could be further improved by using clustering techniques. Based on the similarity between every pair of trees, the future multi-agent system should have the ability of clustering agents according to selected clustering algorithms based on our tree similarity measure.

Acknowledgements

We thank the anonymous reviewers for helpful feedback and Daniel Lemire, NRC IIT e-Business, for discussions about possible alternatives to the arithmetic mean, challenging examples, and ideas about future similarity work in connection with our collaborative e-learning projects. We also thank the CANARIE eduSource Project as well as NSERC for their support.

References

[Boley, 2003] Boley, H., The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations, Invited Talk, INAP2001, Tokyo, October 2001. LNAI 2543, Springer-Verlag, 2003.

[Boley, 2002] Boley, H., Cross-Fertilizing Logic Programming and XML for Knowledge Representation, in: Rolf Grütter (Ed.), Knowledge Media in Healthcare: Opportunities and Challenges, Idea Group Publishing, 2002.

[Boley, 1999] Boley, H., Functional-Logic Integration via Minimal Reciprocal Extensions, Theoretical Computer Science 212, Elsevier, 1999.

[Boley, 2003] Boley, H., Object-Oriented RuleML. http://www.ruleml.org/indoo. March 2003.

[Kifer et al., 1995] Michael Kifer, Georg Lausen, Michael Kifer, and James Wu, Logical Foundations of Object-Oriented and Frame-Based Languages. JACM 42(4): 741-843, 1995.

[Kamat et al., 1993] Kamat, V.N., V.C. Bhavsar and L. Goldfarb, "Learning Machines Based on the Evolving Transformation Model," Proc. 1993 DND Workshop on Advanced Technologies in Knowledge-based Systems and Robotics, Ottawa, Nov. 14-17, 1993.

[Kamat et al., 1993] Kamat, V.N., V.C. Bhavsar and L. Goldfarb, "Learning handwritten Characters using Tree Edit Distances," Proc. of 5th UNB AI Symposium, Fredericton, NB, Aug. 12-14, 1993, L. Goldfarb (Ed.), UNB Press, pp. 185-98, Aug. 1993.

[Kamat et al., 1996] Kamat, V.N., Inductive Learning with the Evolving Tree Transformation System, Ph.D. Thesis, 372 pages, Faculty of Computer Science, University of New Brunswick, Fredericton, 1996.

[Lassila and Swick, 1999] Lassila, O. and Swick, R.R., Resource Description Framework (RDF) Model and Syntax Specification. Recommendation REC-rdf-syntax-19990222, W3C, February 1999.

[Marsh et al., 2003] Marsh, S., A. Ghorbani and V.C. Bhavsar, 'The ACORN Multi-Agent System', *Web Intelligence and Agent Systems: An International Journal*, to appear in 2003.

[Richter, 2001] Richter, M.M., Case-Based Reasoning: Past, Present, and Future. Invited Futures Talk, International Conference on Case-Based Reasoning (ICCBR-2001), Vancouver, British Columbia, Canada, July/August 2001.

[Shasha et al., 2001] Shasha, D., J. Wang, K. Zhang, "Treediff: Approximate Tree Matcher for Ordered Trees," October 25, 2001.

[Sycara et al., 2001] Sycara, K., Paolucci, M., Van Velsen, M., Giampapa, J.A. The RETSINA MAS Infrastructure, Robotics Institute, Carnegie Mellon University, CMU-RI-TR-01-05, March 2001.

[Van et al., 2001] Van Eijk R.M., F.S. de Boer, W. van de Hoek, and J.C. Meyer, "On dynamically generated ontology translators in agent communication," Int. J. of Intelligent Systems, Vol. 16, No. 5, pp. 587-607, May 2001.

[Yang et al. 2000] Yang Q., C. Carrick, J. Lam, Y. Wang, H. Zhang, "Exploiting Intelligent Agents and Data Mining for e-Commerce," in 'Electronic Commerce Technology Trends,' W. Kou and Yesha (Eds.), IBM Press, pp. 115-134, Feb. 2000.

[Zhang et al., 1994] Zhang, K., D. Shasha, and J.T.L. Wang, "Approximate Tree Matching in the Presence of Variable Length Don't Cares," J. of Algorithms, Vo. 16, No. 1, pp. 33-66, 1994.

Appendix: The Weighted Tree Similarity Algorithm in Relfun

```
% Assumes both weighted trees are already in a 'lexicographic' normal form
% (roles branch off in 'alphabetic' order).

% Tree representation uses OO RuleML syntax in Relfun, where roles start
% with a "-", not with a "_" (free in XML, but variable prefix of Relfun):
% http://www.ruleml.org/indoo

% Uses the parameter N(ode) for specifying the fraction to which the equality
% of tree-root node labels is counted relative to the result of the recursive
% comparison of its list of subtrees.

% Uses the functional parameter A(djust), which can be the id(enti)ty, sqrt,
% etc., for compensating similarity degradation for nested trees.

% A binary function, treesim, computes the similarity between its two tree
% arguments, co-recursively calling treemap, which sums up the similarities
% of its two lists of (sub)trees. We assume that the weights in each subtree
% fan-out sum up to 1. Here, 1 means total similarity; 0 represents absolute
% dissimilarity. Since treesim is symmetric in its two arguments, some of the
% algorithm's clauses could be simplified (incurring an efficiency penalty).

% Analogously, a unary function, treeplicity computes the simplicity of its
% single tree arguments, co-recursively calling treeplimap. The simplicity
% measure is used by the similarity measure for subtrees occurring in only one
% of the trees. Here, 1 means total simplicity; 0 (never to be reached) would
% represent infinite complexity.

% Only the deterministic, functional subset of Relfun is required, where
% '!&' is the 'cut-return' infix and "lhs :- cond !& rhs" stands for the
% conditional equation "lhs if cond cut-return rhs" or "lhs = rhs if cond";
% the unconditional equation "lhs !& rhs" abbreviates "lhs :- true !& rhs".
% Built-ins are taken from Common Lisp. E.g., string< and string> compare
% two string or symbol arguments for 'lexicographic' less and greater order.

% Runs at http://serv-4100.dfki.uni-kl.de:8000/~vega/cgi-bin/rfi



% Paste the program into the above URL's Database window:



treesim[N,A](ind[Leaf],ind[Leaf]) !& 1.    % Same leaves

treesim[N,A](ind[Leaf1],ind[Leaf2]) !& 0.  % Different leaves

treesim[N,A](cterm[ -opc[ctor[Label]]      % Same node Label
                  ],
             cterm[ -opc[ctor[Label]]      % on two empty trees:
                  ])
        !& 1.                              % 1

treesim[N,A](cterm[ -opc[ctor[Label]]      % Same node Label on two trees,
                  | Rest1 ],
             cterm[ -opc[ctor[Label]]      % one or both being non-empty:
                  | Rest2 ])
        !&
            +(N,                                  % add N(ode) fraction
```

```
             *( -(1,N),                          % to 1-N fraction of
                treemap[N,A](Rest1,Rest2) ) ).   % subtree list comparison

treesim[N,A](cterm[ -opc[ctor[Label1]]     % Different node Labels
                  | Rest1 ],
             cterm[ -opc[ctor[Label2]]     % on arbitrary trees:
                  | Rest2 ])
        !& 0.                              % 0

treesim[N,A](ind[Label],                   % Same Label on leaf and on
             cterm[ -opc[ctor[Label]]      % arbitrary tree (e.g. empty):
                  | Rest ])
        !&
            treesim[N,A](cterm[ -opc[ctor[Label]] % Substitute empty tree
                              ],
                         cterm[ -opc[ctor[Label]]
                              | Rest ]).

treesim[N,A](cterm[ -opc[ctor[Label]]      % Same Label on arbitrary tree
                  | Rest ],
             ind[Label])                   % (e.g. empty) and on leaf:
        !&
            treesim[N,A](cterm[ -opc[ctor[Label]]
                              | Rest ],
                         cterm[ -opc[ctor[Label]] % Substitute empty tree
                              ]).

treesim[N,A](ind[Label1],                   % Different Labels on leaf and
             cterm[ -opc[ctor[Label2]]      % on arbitrary tree (e.g. empty):
                  | Rest ])
        !& 0.                              % 0

treesim[N,A](cterm[ -opc[ctor[Label1]]      % Different Labels on arbitrary
                  | Rest ],                % tree (e.g. empty)
             ind[Label2])                  % and on leaf:
        !& 0.                              % 0



treemap[N,A]([],[]) !& 0.

treemap[N,A]([ First | Rest ],
             [])                                % no Role1 in empty T2
       !&
            +( *( 0.5,treeplicity(-(1,N),
                                  cterm[ -opc[ctor[Label]],First ])),
               treemap[N,A](Rest,[]) ).

treemap[N,A]([],                                % no Role2 in empty T1
             [ First | Rest ])
       !&
            +( *( 0.5,treeplicity(-(1,N),
                                  cterm[ -opc[ctor[Label]],First ])),
               treemap[N,A]([],Rest) ).

treemap[N,A]([ -r[n[Role],w[Weight1]][Subtree1]   % T1 and T2 start
               | Rest1 ],
             [ -r[n[Role],w[Weight2]][Subtree2]   % with same role
               | Rest2 ])
       !&                                              % With assumption
            +( *( /( +(Weight1,Weight2),2),            % Weight sum = 1:
                  A(treesim[N,A](Subtree1,Subtree2))),  % A(djusted) sub-
               treemap[N,A](Rest1,Rest2) ).             % tree comparison
```

```
treemap[N,A]([ -r[n[Role1],w[Weight1]][Subtree1]
               | Rest1 ],
             [ -r[n[Role2],w[Weight2]][Subtree2]
               | Rest2 ])
       :-
            string<(Role1,Role2)                    % Role1 missing in T2
       !&
            +( *( 0.5,
                  treeplicity(-(1,N),
                              cterm[ -opc[ctor[Label]],
                                     -r[n[Role1],w[Weight1]][Subtree1] ])),
               treemap[N,A](Rest1,
                            [ -r[n[Role2],w[Weight2]][Subtree2]
                              | Rest2 ]) ).

treemap[N,A]([ -r[n[Role1],w[Weight1]][Subtree1]
               | Rest1 ],
             [ -r[n[Role2],w[Weight2]][Subtree2]
               | Rest2 ])
       :-
            string>(Role1,Role2)                    % Role2 missing in T1
       !&
            +( *( 0.5,
                  treeplicity(-(1,N),
                              cterm[ -opc[ctor[Label]],
                                     -r[n[Role2],w[Weight2]][Subtree2] ])),
               treemap[N,A]([ -r[n[Role1],w[Weight1]][Subtree1]
                              | Rest1 ],
                            Rest2) ).



treeplicity(I,ind[Leaf]) !& I.             % I: Current depth degradation value

treeplicity(I,cterm[-opc[ctor[Label]]]) !& I. % (initialized with 1-N fraction)

treeplicity(I,cterm[ -opc[ctor[Label]],First | Rest ])
      !&
           *( /(1,1+(len(Rest))),        %    Breadth degradation factor
              treeplimap(I,[First | Rest ])).


treeplimap(I,[]) !& 0.

treeplimap(I,[-r[n[Role],w[Weight]][Subtree]
              | Rest ])
      !&
           +(*(Weight,treeplicity(*(treeplideg(),I),Subtree)),
             treeplimap(I,Rest)).


treeplideg() !& 0.5.   % Depth degradation factor (change global constant here)
```