Semi-automatic Merging of Content Networks: Matching

Studienarbeit

Hendry Chandra
Information and Communication Systems
Technische Universität Hamburg-Harburg

Advisor:

Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

8th March 2001

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview on Matching

Matching is one of the basic functions required in many applications like Versioning, Merging, and Information Retrieval. The usage of matching in applications can be categorised into:

Similarity Test: In this category, the applications involve the matching process only to test if a segment of similar data exists in two or more information objects. Sample applications in this category are:

- ◇ Information Retrieval which is used with all kinds of databases, especially with databases where vague queries play a dominant rule, like e.g. test databases or image databases. The matching is used for instance to query the database for certain feature vectors.
- ◇ Similarity Test Services. A sample usage of this kind of service in Internet is a service to compare online catalogues. This service allows the user to enter conditions about commodities he/she is interested in, and in return to get information about the commodities that fulfil these conditions.

Change Detection: In this category, the applications involve the matching process to detect the difference between two information objects. Sample applications in this category are:

- ◇ Change Detection Services: One example of this kind of service in the Internet is a service from NetMind called Mind-it. This service allows the user to register a web-page of his/her interest. When the web-page is updated, the user will be notified by Mind-it that his/her page of interest has been changed. In this sample application, the matching process is needed to compare the cached version of the web-page with the current version of the web-page. When the result of comparison states that the versions are different, the notification is sent to the user (see also [13] and [15]).
- ◇ Versioning: This application is used to compare different versions of the same information object. The user is mainly interested in the changes between the versions of the information object being matched. The matching process is needed to compute the changes between the versions, which are then presented in a nice

and meaningful way, for instance by using different fonts and colours (see also [8] and [1]).

◇ Merging: In this application, the matching is used to find both the similarity and differences between information objects being merged. The application then resolves the differences between the objects into one new object that contains consistent information from the original objects. The matching implemented for this report is also used for a merging application.

1.2 Matching Classified Networks of Content

The matching process implemented for this report focuses on matching a type of information object called Classified Network of Content (CNC). Classified Networks of Content are a way to categorise information (which may exist in any format, e.g. files, databases, etc.), so that it will be easy for a user to browse through the structure while searching for a particular piece of information. In the CNC matching process, a component of CNC is paired with a component from another CNC based on their similarity. The pairs obtained from the matching process represent the similar part of the CNCs. Components that do not have any matching partner represent the difference between the CNCs, also called the *delta collection*.

1.3 Organisation of This Report

The following describes the structure of this report.

Chapter 1 describes the matching process in detail, and the focus application of this report.

Chapter 2 describes what a Classified Network of Content is, and explains the general idea behind merging. Chapter 2 also explains how CNC merging is done in four phases, and different scenarios in which CNC merging is needed.

Chapter 3 gives a definition of matching, and conceptually describes the matching process phase by phase.

Chapter 4 describes the Brain, one of the existing applications, which can realise the concept of Classified Networks of Content. The implementation of the matching process based on the Brain is explained here in detail, phase by phase.

Chapter 2

Merging Classified Network of Content

2.1 Classified Network of Content (CNC)

Classified Networks of Content (CNC) are an effective representation of digital content. A CNC provides a way to browse through the semantic classification of information resources. Figure 3.3 represents a typical CNC.

The tree structure shown here is one of the commonly used structures in the information world nowadays. It can for example be found in web catalogues like Yahoo! [6]. Therefore, tree-structured CNCs are used here, to explain the general notion of a CNC.

In figure 3.3, each node in the graph represents one component in the CNC. The nodes numbered between 1 and 20 are termed as *classifier node*s. They build up a classification network, here in form of a hierarchy. The rest of the nodes which are drawn below the classifier nodes are representatives of digital content, and belong to the class represented by their parent classifier node. They are called *content node*s. The whole concept of classification used in the CNCs can for example be found in a biological classification system for living things.

With respect to the classification of living things, the content node is analogous to a real living thing, e.g. a human being. With respect to the information world, the content node represents a real information object, e.g. a document. A content node has a link to the document that it represents. The document exists outside the CNC. The reason is because the CNC is mainly used to browse the information and not to carry all of the information, which will need an enormously large space, and can also be rather time consuming.

For example, look at figure 2.1: Node 1 represents a class 'vertebrate'. A node below node 1 represents an entity that belongs to the class 'vertebrate', e.g. node 2 represents the class 'mammal' and node 3 represents the class 'reptiles'. If the developer of CNC wants to put the actual information object below a classifier node, then a content node must be used to represent this information object. In the example, node 4 represents 'human', and it has a link to a picture of a man.

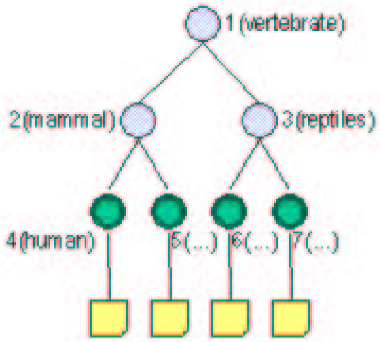


Figure 2.1: A sample CNC

2.2 Scenarios of Merging CNC

The usage of computers has invaded many facets of today's life. A wealth of information from many disciplines, like e.g. sciences, arts, and other parts of life has been produced, processed, and stored utilising a digital device or a computer. The present growth of the Internet makes it easier to obtain information from all around the world. This means that the amount of information accessible for a person or an institution is growing fast.

Developing a CNC that supports such quantity of information or even smaller subsets of it is not a trivial task. A co-operative and long-term effort is needed to construct a high quality CNC. The construction usually starts with one common CNC, and then updates (addition, modification or deletion) of classification and content nodes are constantly brought back into the CNC by many authors.

One possible scenario of constructing a CNC is as follows. Each author updates his/her own copy of the original CNC. After the updates, each author will have his/her own updated version. At the end of the construction, all updates from all authors must be combined again into one CNC. This process of consistently combining all updates from all CNC copies is called *merging* the CNC.

In large CNC projects, the different separately updated CNCs are merged into a single common CNC periodically. A number of reasons make this necessary, e.g.:

- ◇ Refocusing the updates of all authors, so that each author has a clear cut responsibility of what kind of updates he/she is entitled to. A better management of update responsibilities makes conflicts at the merging process less likely to occur.
- ◇ Better coordination between authors. Often, an author needs the work results of other authors before he/she can continue with his/her updates. Instead of creating his/her own version of the work, which in turn will cause a conflict of overlapping updates, it is better to get the result of the responsible authors. This is best done through the merging process.

In the merging process, the last common predecessor of all newly updated CNC can be either involved in the merging process, or excluded.

The merging scenario explained above, in which a common predecessor of the CNCs exists, is called *consolidation merging.* In consolidation merging, most updates are assumed non-overlapping. The merged version incorporates most of the updates from all separate

CNC versions. This includes also the deletion of a node in one of the updated CNCs. The overlapping updates are solved interactively by the corresponding authors. See also [12], [14], and [16].

Another possibility is that no common predecessor of the CNCs exists. In this case, the CNCs being merged were developed totally separately, but they all share a common focus. This scenario is called *reconciliation merging.* In [12], [14], and [16], reconciliation merging is assumed to handle a lack of complementary differences between the CNCs. Most of the differences between the CNCs are overlapping. Reconciliation merging is executed on the CNCs to resolve the resulting conflicts.

2.3 Merging Phases and the Role of Matching

The merging process comprises four phases. The following describes each of these phases. From the descriptions the role of the matching process for merging can be concluded.

- ◇ *Merging Configuration* phase. In this phase the number of *merging candidate*s and the role of each is assigned. The *merging policy* is also determined in this phase. A *merging candidate* is an information object which is involved in the merging process. A *merging policy* can be seen as a set of rules on how the merging should be conducted. For example, the merging process can incorporate all complementary insertions (via addition of the nodes), but the user is always asked if there is a deletion of a node.

- ◇ *Matching* phase. In this phase the difference and the corresponding common parts of the CNCs are computed. The matching process reveals nodes from different CNCs, which are related to each other, and proposes them as a pair. The nodes in a pair are not necessarily exactly the same, but a certain degree of difference between them is tolerable. In this case, updates inside the nodes can also be merged.

  Apart from the common part of the CNCs, the difference is also revealed by the matching process. Nodes that do not have any relation with nodes from another CNC are called singles. These single nodes are the difference between CNCs, due to node insertion or node deletion.

- ◇ *Change Detection* phase. In this phase, a set of commands (an *edit script*) is formed for each single update (or difference) in the CNC. The formation of the edit script takes the result of matching phase as input. These commands will later be applied to one of the CNC merging candidates to get a merged CNC version.

- ◇ *Change Integration* phase. The edit script is applied to at least one of the CNCs. This phase may also involve user interaction.

The visualisation of merging can extend from the end of the matching phase (where the difference has been detected and displayed) up to the end of change integration (where user will see the result of integrating the differences).

Chapter 3

Matching

3.1 Definition of Matching

Matching an object A with an object B can be defined as the identification of which component in object A is related to which component in object B (see figure 3.1). One special case in focus is when the relation of the two components is the similarity between them. Once the two components — which are similar to each other — have been found, they are referred to as a *pair*.

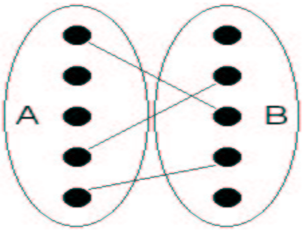


Figure 3.1: Matching definition

Pairs from object A and object B can be obtained by comparisons of their components. A straightforward but naïve way would be to take all components of object A one at a time, and compare them to all components from object B. This would lead us to N × M comparisons, where N is the number of components from object A, and M is the number of components from object B. This way of comparison is not recommended, particularly because components that are totally different from each other are also compared.

Section 3.5 discusses strategies to reduce the amount of comparisons by identifying and taking into account only components that have some level of similarity.

3.2 Related Works on Matching

Most of the existing matching applications used in programmes like [10] work on flat data, e.g. texts or documents. This works by finding the *longest common sequence* which exists in both information objects being matched. [3] describes how to compute the longest common sequence. The longest common sequence found by the system is then treated as a reference. Other common sequences can be computed as being *not changed* or *moved* with respect to the longest common sequence. Other sequences can be considered as being *inserted* or *deleted* from the information objects.

For structured data, not many solutions for the matching problem as given in this report can be found. [7] describes how to efficiently find a sub-tree (called *pattern tree*) in a tree-structured text database. Thus [7] describes a different case compared to the work in this report. The result of algorithms there is the location of the pattern tree inside the tree-structured data.

The work in [1] is more closely related to the work in this report. The described algorithm focuses on finding the most efficient edit script (called *minimum-cost edit script*). The implementation of [1] works on text data (LaTeX documents), therefore the algorithm creates its own nodes, node types, identifiers for each node etc. Another difference is that algorithm in [1] is sensitive to the relative position of nodes, while this report is more sensitive to the surrounding nodes (see section 3.5.2).

The matching in [11] uses fuzzy logic to query a catalogue database. This supports the fact that a user who queries the catalogue and defines that he/she wants an item that costs not more than 1000, would certainly have some interest on an item with a price of 1001 if the item is of good quality. The fact, that the user sometimes does have some interest on items that do not fulfil exact conditions, is also applicable in many other areas. The work in [11] finds that boolean predicates are severely limited and not suitable for this purpose.

3.3 Matching Process Overview

The whole matching process can be divided into four phases. In the first phase, the matching configuration phase, the user defines all operational parameters for the whole matching process. These parameters determine how the matching process will be conducted.

The second phase is the identifying candidate-pairs phase, where the system reduces the number of pairs to be computed in the next phase, the actual comparison phase. This is because the computation in the comparison phase consumes a lot of resources.

The third phase is the comparison phase, where the system computes the similarity of pairs proposed by the identifying candidate-pairs phase. The computation result is a value that represents the degree of similarity between the two components in the pair.

The last phase is the evaluation phase, where the system evaluates the pairs based on their similarity values, which are the result from the comparison phase. User interaction might be needed in this phase. The result from this phase might be e.g. the best pair, or the best two pairs, or even more.

3.4 Matching Configuration

The matching configuration phase defines how the whole matching process should be carried on. The user defines in this phase information objects that will be participating in the matching process. Analytically, the participants are also merging candidates, but it this is not necessarily true the reverse way. A merging's *change target*, for instance, does not need to be involved in the matching process. In the case of a Network of Content, an information object can be the whole network, or just a subset of the network.

For the matching process, the information objects have to be divided into smaller parts, called *component*s. What constitutes a component depends on the structure of the information object itself and the chosen way of decomposition. A component in a Network of Content can be a node or a subset of the network (a collection of nodes). As an example, a document can be seen as a Network of Content, as depicted in figure 3.2. A component of the document can be a chapter, a paragraph, a sentence or a word, depending on the *granularity* defined for the matching process.

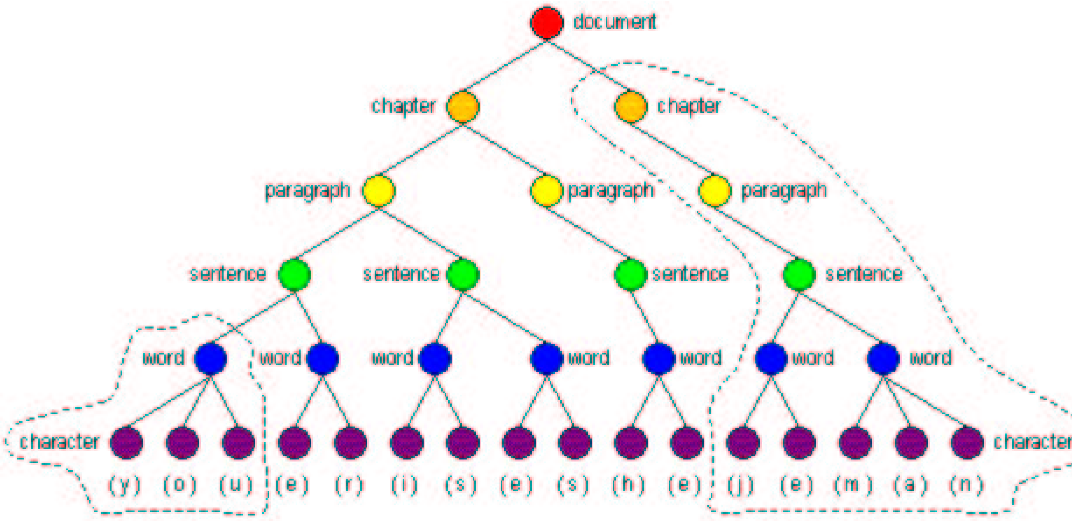


Figure 3.2: A document as a Network of Content

The *granularity* is also determined in this matching configuration phase. The user defines how fine or how coarse the system should consider a component. For the document example above, this means that when a user defines that a component is a word, this is a relatively fine granularity. In figure 3.2, this is indicated by a dashed curve drawn on the left part of the Network. The word here contains three characters, which are y, o, and u. When the user defines that a component is a chapter, then this is a relatively coarse granularity. In figure 3.2, this is indicated by a dashed curve drawn on the right part of the network. The content of the chapter here is deeply structured. The chapter has one paragraph, the paragraph has one sentence, the sentence has two words, and in total there are five characters at the base of the structure. In the matching configuration phase only one granularity can be defined, which is applied to all information objects involved throughout the matching process.

In this work, components of information objects are focused to the nodes of Classified Networks of Content (see figure 3.3). In the figure, each of the nodes is a component, so in total there are 36 components in the network.

In a structured information object, a component has a *type*. Examples of component types in a document are a chapter, an appendix, a preface, an index, or a bibliography. Notice that the structure of each type can be similar, but it can also be totally different. Structures of a chapter and a preface are similar, but structures of a chapter and a bibliography are different.

A component has some values, which are termed as the component's *content*. Example of a component's content in a document: A chapter consists of several paragraphs, which in turn contain several sentences, which in turn contain several words. As has been shown here, content of a component can be structured again.

Beyond its content, a component also carries another kind of information, for instance the size of a chapter, or the chapter's cardinal/ordinal number. This kind of information is termed

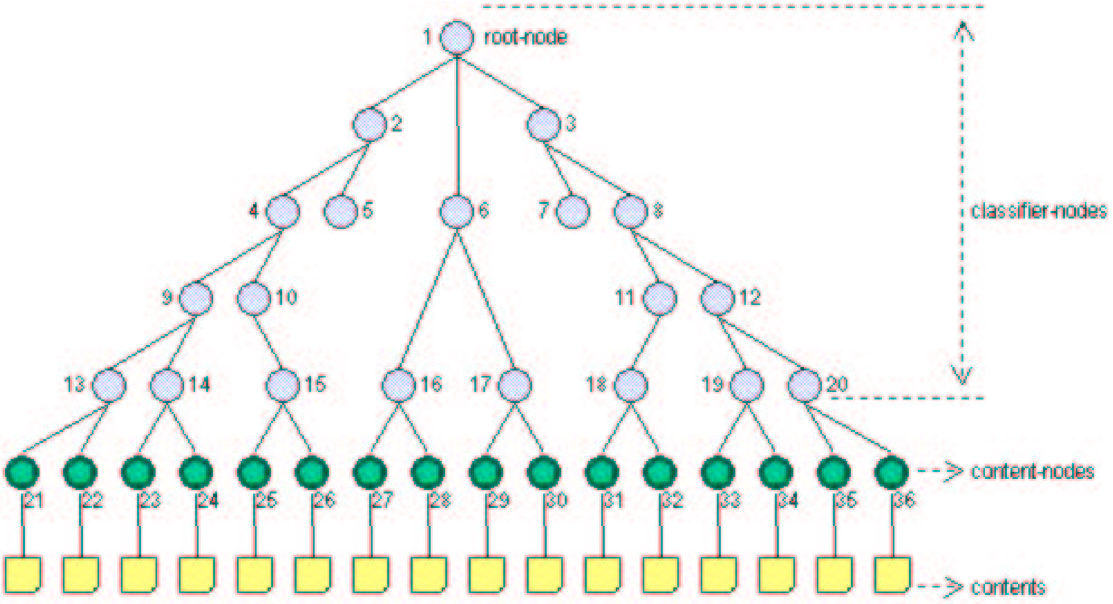


Figure 3.3: A Classified Network of Content

as the component's *properties*. It is also known as *meta-information*. Meta-information is information beyond the semantic content of an information object. It describes the object's characteristics in order to support a variety of operations on the object like better retrieval, administration, rights management etc. The operations can be carried out by either a human user or by a program (see also [5] and [4]).

The component's surrounding or its position relative to other components is termed as its *context*. Example: The contexts of the second chapter are the first chapter and the third chapter.

All four factors described above (*content*, *context*, *properties*, and *type*) are termed as *similarity factors*. The similarity factors are also defined by the user in the matching configuration phase. The user defines which of those factors are used in which phase of the matching process. The main usage of the similarity factors is for the system to come up with a decision whether two components are similar or not. This decision takes place in the comparison phase of the matching process (see section 3.6). The other usage of those factors is for the system to come up with *candidate-pair*s, which takes place in the identifying candidate-pairs phase (see section 3.5).

In the comparison phase, the system uses a function called *similarity function* in order to decide whether two components are similar or not. The similarity function has several parameters that must be set by the user in the matching configuration phase. The parameters can be categorised into two parts. The first category comprises all parameters related to the algorithm used to compute the similarity between two components. The matching process can implement many possible algorithms, and the user selects which one is suitable for the current application. The second category is the similarity factors, which have been described above. The similarity factors are the object of computation. The user selects which of those factors are suitable to be used for the current application, and which algorithm is used for each respective similarity factor. The result of similarity function is similarity value of each pair of components. See also section 3.6 regarding the comparison phase.

After the comparison phase, it is possible that there are more than one candidate-pair per component. Thus, a filtering mechanism has to be applied to choose the best pairs of the comparison result. This filtering is done in the results evaluation phase. There are several filtering strategies used in the results evaluation phase (see section 3.7), which also have some parameters that must be set by user in the matching configuration phase.

3.5 Identifying Candidate-Pairs

3.5.1 Identification Process

A *candidate-pair* is a pair with the preliminary assumption that the two components in the pair have a certain degree of similarity between them. This assumption has to be proven in later phases of the matching process.

A straightforward way of getting candidate-pairs for component X from Network A is to consider all components in Network B as possible partners for component X. With this way of finding candidate-pairs, the comparison phase will have to execute N × M comparisons. That is; each component in network A is compared to all components in network B.

Comparison is performed on pair by pair basis. Each comparison lowers the system performance. Therefore the number of comparisons needs to be reduced as much as necessary, i.e. candidate-pairs which are not promising have to be excluded. In this phase, the system makes an assumption of when two components can be considered as having a certain degree of similarity. This assumption is based on the similarity factors. If two components do not fulfil the assumption, they are not considered as a candidate-pair. If two components fulfil the assumption, they are considered as a candidate-pair, but their similarity has to be proven later in the comparison phase.

In order to avoid adding more computation complexity, which in turn will consume more system resources, the computation of whether two components fulfil the assumption or not in this phase needs to be as simple as possible. If this phase requires relatively complex computation that matches the complexity of a comparison, then it will cost as much as a comparison. Because in this phase the system needs to go through all components from both networks, and the comparison phase needs to go through some pairs again, the total resources consumption from both phases would be even higher than the resources consumption of N × M comparison steps. This, however, is against the purpose of this phase, which is to lower the number of computing steps under N × M comparisons.

3.5.2 Similarity Factors

The assumption that two components are considered as having a certain degree of similarity is based on either one or combination of similarity factors (see also section 3.4). This assumption is used to obtain candidate-pairs. The following describes how similarity factors can be used to obtain candidate-pairs.

Properties are meta-information of the corresponding component. Examples of component's properties are:

◇ Identifier

◇ Creation time stamp

◇ Size, etc . . . .

This meta-information can be used to quickly identify candidate-pairs. Example: The assumption is if two components have the same identifier, they must have a certain degree of similarity. Therefore, they should be considered as a candidate-pair. Other examples: For being considered a candidate-pair, two components need to have the same creation time stamp, or their size may not differ more than 10 percent. In case of the last example, good sorting and search functions are required to support the identification process.

Some properties mentioned in this section make sense only if they are used to match information objects, which have a common predecessor. E.g. it does not make any sense to consider a candidate-pair based on the components' creation time if the two information objects were created separately. The same is true for the components' unique identifiers. If the information objects were created separately, most probably similar components will have different identifiers.

All of the general explanations above are also true when the information objects are Classified Networks of Content, where a component is represented by a node in the network.

Context: The context of a component is its surroundings (neighbouring components). The context of a component can also be its position relative to other components inside the information object, which in case of this report is a Classified Network of Content.

As example, see figure 3.3 again. A component in this Network of Content is a node. Each node is one component. The surrounding nodes of the node with identifier number 17 are nodes with the following identifiers: 6 (as parent node), 29 and 30 (as child nodes), and 16 (as sibling node). The relative position of the same node 17 is the following; It is the second child of node 6, and node 6 is the second child of the root node (node 1). In this network, the root node serves as a reference node. The relative positions of all nodes are recorded from the reference node. The reference node does not always have to be the root node. Another special node can also take the role of the reference node.

In the network example representing a document (figure 3.2), lets assume that the granularity takes one word as one component. The surrounding components (words) of the word 'er' are the words 'you' and 'is'. In its relative position the word 'er' is the second word in the document.

The context of components can be used to obtain candidate-pairs. In figure 3.4, the Network of Content from figure 3.3 has been updated by changing the position of the nodes 6 and 12 together with their subtrees.

Suppose the matching process is executed on the two Networks of Content, from figure 3.3 and from figure 3.4. The method of finding candidate-pairs by using the *surrounding context* of a component proves to be useful in this case.

The nodes 16, 17, and 27 – 30 have the same surrounding in both networks even if a whole subset of the network has been moved around. Therefore the identifying candidate-pairs phase will propose these nodes as candidate-pairs. Node 16 from the original network in figure 3.3 is proposed to be paired with node 16 from the network

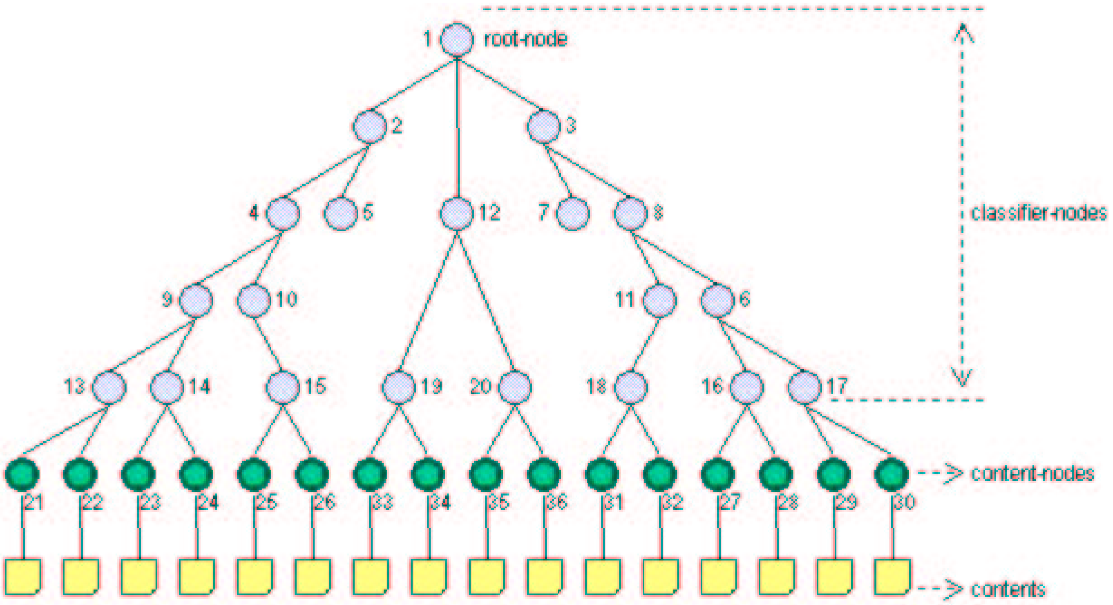


Figure 3.4: Moving subsets of the network

in figure 3.4, node 17 from the network in figure 3.3 is proposed to be paired with the node 17 from the network in figure 3.4, etc . . . .

In figure 3.5, the same Network of Content from figure 3.3 has been updated by modifying the branch of node 6 into a new branch of node 37. Suppose the nodes in each respective position have some level of similarity. Node 6 is similar to node 37, node 16 is similar with to node 38, etc . . . . In this case, the method of finding candidate-pairs by using the *relative position* context will prove to be helpful to find good candidate-pairs. Node 6 has the same position as node 37, so they are proposed as a candidate-pair. Node 16 has the same position as node 38, they are also proposed as a candidate-pair. The same applies for the rest of the nodes in the branch.

In case there are two copies of a component in the Network which are exactly the same, but have different neighbours and positions, the system can differentiate these two components and propose the right candidate-pair for each of them by looking to their neighbours, or by using their difference in position.

Type: Types of components also help to filter out pairs which are not promising. Components in the information object can have different types. They can be categorised into different types mainly by their data structure.

One possibility to categorise components into different types is their function in the information object. For example, information object in figure 3.3, which is a Tree-like Network of Content, has two types of nodes, i.e. classifier nodes and content nodes. The function of a classifier node is to group other nodes (content nodes and other classifier nodes) into a category. The function of a content node is to carry the information content. A content node links directly to the actual content. Analogously, classifier nodes can be seen as directories, and content nodes can be seen as files. Theoretically, both types of nodes should have a different data structure. But as the implementation

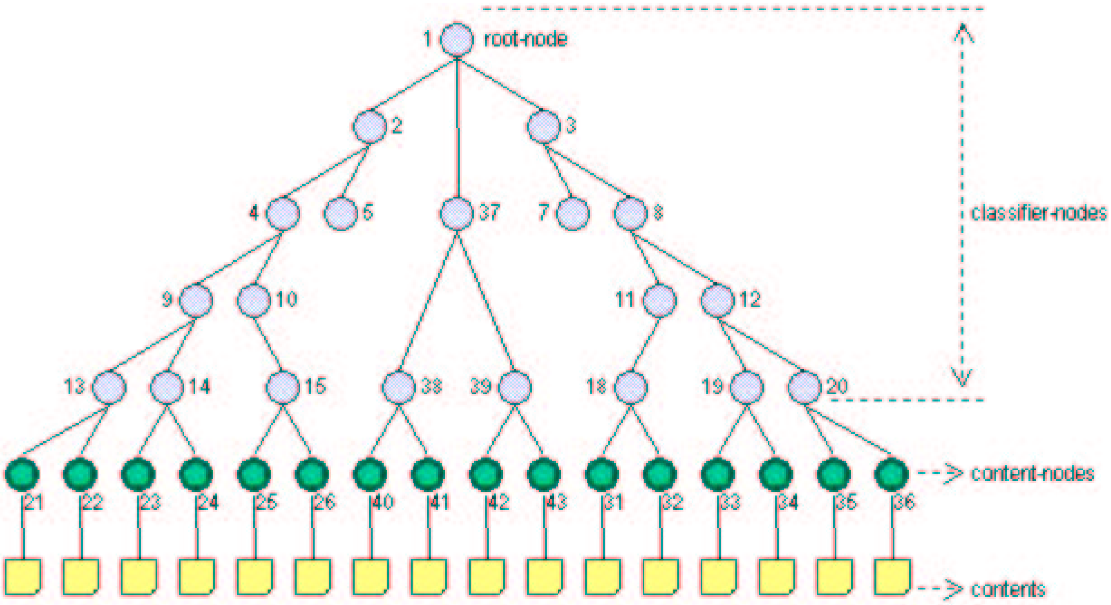


Figure 3.5: Updating a subset of the network

relies on a third-party system, both types of nodes are implemented with the same data structure. Classifier nodes have a *null* value in the link to the content.

By defining which types can be considered in a pair, the system can filter out all other components, whose types are not considered. As an example: By defining that both components in a candidate-pair must have same type, then the system only look for such candidate-pairs, therefore neglecting the possibility of having candidate-pairs of mixed component types.

It is also possible to mix different types of nodes in a pair, for example nodes with type X from network A can only be paired with nodes with type Y from network B. Comparing different types of nodes requires one extra phase called *schema matching*, which will be explained in section 3.6.

Content of components can also be used to find candidate-pairs, but only to a certain extent. When using the content of components, generally only a small part of the components' content is used. Remember that the computation in this phase must be kept simple. Using the whole content of components in this phase would likely result in a too complex computation.

The most accurate way — in terms of result — to find candidate-pairs is the straightforward way, because it considers all components of information object B as candidate-partners for a component of information object A. Therefore, it considers all possible pairs. When the identifying candidate-pairs phase is executed, it excludes some of the pairs. Therefore, not all possible pairs are considered in the comparison phase anymore. When the identifying candidate-pairs phase does not utilise correct similarity factors to find candidate-pairs, it could happen that good candidate-pairs are not proposed. This in turn will affect the final result of the whole matching process. The identifying candidate-pairs phase in a way balances the speed of computation with the correctness of the result. Therefore a correct choice of the similarity factors to be used plays an important role. Which similarity factors are the correct ones, mainly depends on the information objects being compared and for what applications the matching process is done.

3.6 Comparison, Computation of Similarity

The comparison phase computes the similarity of the two components in each candidate-pair resulting from the previous phase (identifying candidate-pairs phase). The result of the computation will be a value that represents the degree of similarity between the two components in the respective pair. This phase uses the similarity function to compute the degree of similarity between the two components in a pair. The similarity function needs two kinds of inputs, one of which are the similarity factors.

When more than one similarity factors are defined to be used in the similarity function in this phase, the comparison result will be several boolean or fuzzy values. Example: In a comparison of node C and node D, the similarity function utilises all content and context of the nodes for the comparison phase. One possible result from the utilisation of the content of both nodes:

- ◇ Field 1 of the content from both nodes are similar.
- ◇ Field 2 from both nodes are not similar, etc ....

And possible result from the utilisation of the context of both nodes:

- ◇ Neighbour 1 of the Context from both nodes are similar.
- ◇ Neighbour 2 of the nodes are not similar, etc ....

In this case, some fuzzy approach or functions can be used to come up with one representative value of the similarity between both nodes. Example: Because there are more than one field in the content, the value of content-similarity can be represented by the percentage value of how many fields match over the total amount of fields available. Match means both fields' values are similar or have a high probability that they are the same. The same can be done with the context of nodes' results. To combine the value of content-similarity and context-similarity, both of them can be summed up with the help of weights. When, for instance, content-similarity is more important in the decision of the nodes' similarity then the content-similarity can be multiplied with a larger weight. The context-similarity, which is less important compared to content-similarity, can be multiplied with a smaller weight, and finally the intermediate results are summed up to obtain one representative value of the similarity between the two nodes.

One important point to be discussed here is taking the components' type into consideration to come up with a decision of similarity between two components. In comparison of components of the same type — called *instance matching* — a direct map of the content's fields exists, since the data structures of both components are the same. Example: Field 1 is directly related to Field 1 of the other component from the other information object. Therefore, a comparison of both respective fields can be done directly.

However, a candidate-pair can be composed of components of mixed types. Example: Nodes with type X from network A can be paired with nodes with type Y from network B.

To compare the components correctly, an extra phase — called *schema matching* — has to be done before the actual comparison is executed. The schema matching can be considered as a precomputation before the actual comparison. The following is an example of schema matching: Nodes with type X from network A have two fields in their content. One field is named 'labour cost' and the other is named 'material cost'. Nodes with type Y from network B have only one field in the content, which is named 'total cost'. Before comparisons of the two types of nodes can be done, a mapping of the fields should be carried out. In this example, the 'labour cost' field and 'material cost' field each have some level of contribution into the 'total cost' field. The two fields in a node of type X are mapped to the 'total cost' field in a node of type Y. Both 'labour cost' and 'material cost' values need to be summed up, and then the result is compared with the 'total cost' value.

The complexity of schema matching is largely dependent of the type of nodes being compared. It could be that only a simple mapping is needed, for instance, the field 'taxi fares' for nodes that represent employees who do not have a car, is mapped to the 'gasoline cost' field in nodes that represent employees who have a car. In another case, more complicated pre computations of fields are needed before a comparison can be executed.

3.7 Results Evaluation

It is possible that a component is involved in more than one candidate-pair after the comparison phase. That means the component has more than one candidate-pair. When only one pair per component or some limited number of pairs is needed, as in case of this report, the pairs should be evaluated further to decide which of them is taken and which are ignored. The evaluation can use one or a combination of several strategies to select one pair per component:

Best Match strategy: With this strategy, the pairs are ranked by their similarity value and the best is taken.

Threshold strategy: A threshold value is defined and every pair that has a similarity value above the threshold is taken, otherwise the pair is ignored.

User Decision strategy: When there is more than one candidate-pair for a component, interaction with the user is triggered to select which pair will be taken. This is because the user has a deeper knowledge of which pair is the best.

The involvement of user makes the system a semi-automatic matching process, while a user-involvement free system is called automatic matching process.

Which evaluation strategies should be used depends largely on the application of the matching process. Examples: In Change Detection, Information Retrieval, and Similarity Test applications, there is no absolute necessity to have only one pair per component, since the user often wants to see all possible information regarding the result of the matching process. In this case, the threshold strategy or the user decision strategy or a combination of both is suitable for the purpose.

In merging, only one pair per component is needed. Therefore, the best match strategy or the user decision strategy would be necessary. A combination of the best match strategy and the threshold strategy, or a combination of the user decision strategy and the threshold strategy is also suitable.

Chapter 4

Implementation

4.1 The Brain

The *E-MergeNC* tool, which is developed for this work, is implemented based on an already-existing Network of Content structure developed by *Natrificial LLC.* called *The Brain*™. Natrificial offers several products. Two of them are mentioned here because of the close relationship to this work.

PersonalBrain™ is a product that enables personal usage of The Brain. The product comes in form of an executable programme. With this product, users can create and update a Network of Content and also create, edit, delete, and access the nodes and their information inside this Network of Content. Each of these Networks of Content is called a *brain*, and each node in the Network of Content is called a *thought*. This work uses PersonalBrain (version 1.73) to develop the Networks of Content, which are the information objects used in the matching process.

BrainSDK™ is a product that enables IT professionals to embed The Brain into their own applications. The main part of the package is a Java API, that enables programmers to create and update a brain; and also to create, edit, delete, and access thoughts in the brain. This work uses BrainSDK version 2.1.

From [9] and [2], there are several differences between the PersonalBrain and BrainSDK with regards to the Network of Content (brain) that they produce. The brain produced by PersonalBrain contains more information compared to the one BrainSDK produces. This is because PersonalBrain is a self-stand programme; therefore the brain it produces provides as much information as needed for personal usage. In contrast, BrainSDK is intended to be used as generally as possible; therefore the brain it produces contains only information that is commonly needed by all applications referring to the API. A brain file which is produced by PersonalBrain has the extension *.brn* and a brain which is produced by BrainSDK has the extension *.jbr*. Included in the BrainSDK package is a brain conversion tool that converts a *.brn* brain file produced by PersonalBrain into a *.jbr* brain file that is usable by BrainSDK. The matching process implementation is based on *.jbr* brain files. The brain can be created in a manner to exactly conform to a Network of Content.

Figure 4.1 depicts complete relationships between thoughts in a brain. The *active thought* is the thought that is currently being processed by the system. A *parent thought* is a thought which can be seen as a super concept of the active thought. The classification scope of the parent thought confines a larger class, one of whose sub classes is represented by the

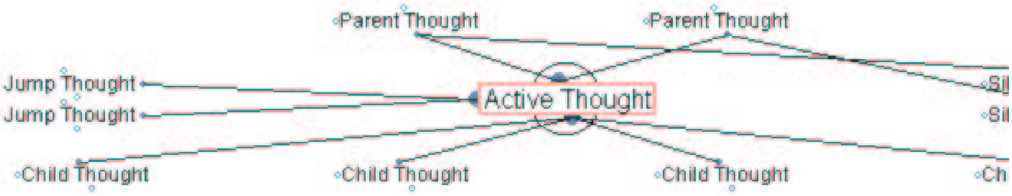


Figure 4.1: A cluster of thoughts in a brain

active thought. The link that connects the active thought with the parent thought is called *parent–child link*. The link name comes from the role of the thoughts at each end of the link. Consequently the active thought is a *child thought* of the parent thought. The active thought can further be divided into smaller sub concepts, each of which is represented by a *child thought*. The link type between the active thought and a child thought is also a *parent–child link*. A *jump thought* can be seen as a cross-reference from the active thought. The jump thought has a relation to the active thought, but it does not fit in the hierarchical structure of the active thought. The link type between the active thought and the jump thought is called *jump link*. This means that to the jump thought, the active thought is also a jump thought.

According to [9], each thought can have from 0 up to 32 multiple parent thoughts. It can have from 0 up to 128 children, and it can have from 0 up to 32 jump thoughts. For sibling thoughts, it is simply the amount of other children of the parents, ranges from 0 up to $((32 \times 128) - 1)$ siblings.

In figure 3.3, when the system is trying to find pairs for the node with the identifier number 17, then this node is referred to as the *active thought*. Node 6 is referred to as the *parent thought*, nodes 29 and 30 are referred to as *child thought*s, and node 16 is referred to as *sibling thought*. When the system is trying to find pairs for node 6, then the active thought is node 6. The parent thought is node 1, child thoughts are node 16 and 17, and sibling thoughts are nodes 2 and 3. As a conclusion, the relation-name of thoughts is relative to the active thought, which is changing depending on the current thought being processed by the system.

There is only one rule that restricts the formation of relationship between thoughts, which is: there can be only one link between two thoughts. With this rule in hand, the brain can be created as a *full-mesh* Network of Content. In a full-mesh network, each node is directly connected to all other nodes. In order to form a tree-like Network of Content with the brain, an important restriction must be added when creating the brain, which is: An active thought cannot have a child thought that also appears above the active thought within the hierarchy. Relationships between two thoughts that can cause the formation of a cyclic graph cannot exist in the brain.

All thoughts in a brain have the same type and data structure. Each thought has the capability to be linked to a file — the content information — outside the brain. This fact is good enough for implementation of tree-like Classified Networks of Content. The implementation of a classifier node uses exactly the same type of node — which is a thought — as the implementation of a content node, with some assumptions to differentiate between both implementations. The assumptions are:

A classifier thought has a *null* value in its link to the content information. This practically means that a classifier thought does not have links to any file outside the brain.

A content thought has a link to a file outside the brain, but it does not have any child thought beneath it.

4.2 Matching Configuration

The matching process implementation is based on brains produced by BrainSDK. From [2], a summarisation of the availability of similarity factors present in the brain has been put into table 4.1. Table 4.1 also states which of those similarity factors are used in the matching process implementation.

Similarity Factors	Available in brain	Used in implementation
Properties	Identifier, Private/Public flag and date, Modification flag, ReadOnly flag, Activation flag	Identifier (when it is stable)
Content	Name, Keywords, Location	For identifying candidate-pairs, ranges from *either one* to *all.* For comparison, ranges from *none* to *all.*
Context	Parent thoughts, Child thoughts, Sibling thoughts, Jump thoughts	For identifying candidate-pairs, ranges from *either one* to *all.* For comparison, ranges from *none* to *all.*
Node Type	*none*, all thoughts are of the same type	*none*, all thoughts are considered of the same type

Table 4.1: Similarity factors available in the brain, and implementation

4.2.1 Properties of Thought

From table 4.1, it can be seen that each thought has several properties, which are:

Identifier: Each thought has an identifier of type integer, which is unique inside a brain. Unfortunately the identifier is unstable.

When a brain is updated, the identifier of the same thought might change. This is especially true in the case of deletion of thoughts. The identifiers are counted in incremental order from 1 to the number of thoughts in the brain. There are no gaps between identifiers. Example: A brain contains 100 thoughts. So the thoughts are numbered from 1 to 100. When the user deletes a thought with identifier number 55, all thoughts with an identifier larger than 55 will be shifted one step down. After the deletion, thought number 56 gets a new identifier with number 55, thought 57 will have a new ID number 56, etc . . . . So at the end, the counting of the thoughts' identifiers is continuous again, from 1 to 99 (the new number of thoughts in the respective brain).

Due to this instability, the identifier of thought can not be used each time the matching process is performed. Before the identifier is used, the stability of thought identifiers between two brains being processed must be detected. This is just true in case of matching two brains which have a common predecessor, because searching candidate-pairs by identifier only makes sense in this case. Section 4.3.1 describes the implementation of detecting the stability of thought identifiers in two brains.

Private/Public flag and date: These properties of a thought are meant to protect the content of thoughts from being freely accessed by public, when the brain is put on a network environment, i.e. the brain is accessible via the Internet. The flag has 4 possible values, which are:

- ◇ Private
- ◇ Public
- ◇ Private before
- ◇ Private after

For the first two options, the date value is not needed — although it is still accessible — because the thought is always in *private* status or always in *public* status. For the later two options, the date value is needed to specify the time-border between *private* and *public* access to the respective thought.

Usage of these properties in the identifying candidate-pairs phase and the comparison phase is not implemented, as the resulting benefits are not worth the effort.

Modification flag: This property indicates whether a thought has been modified in the current session or not. When the brain is closed and opened later, the value of this property is reset. Because this flag's value is valid only temporarily, it cannot be used as similarity factor.

ReadOnly flag: This property indicates whether a thought can be modified or not. Although it is possible to use this property as a similarity factor, it is not implemented because of the same reason as for the private/public flag and date fields. The benefit it brings is not worth the effort of implementation.

Activation flag: This property indicates whether the respective thought has been activated in the current session or not. The validity of this property's value is also only temporary. Therefore, it is not usable as similarity factor.

4.2.2 Content of Thoughts

Each thought in a brain has content that consist of three fields, which are:

- ◇ Name
- ◇ Keywords
- ◇ Location

The type of each field is a string. The location field contains a link to a file — the actual content of the thought — outside the brain.

In the implemented matching configuration phase, a user can choose to use which of the content fields — name, keywords, and location — to use in the identifying candidate-pairs phase and in the comparison phase. The selection of thought content is distributed between two separate UIs, one for the identifying candidate-pairs phase, and the other for the comparison phase.

The separate selection interfaces reflects the distribution of the general similarity test of the whole matching process over the two phases of identifying candidate-pairs and comparing components. For example: The *definition of what is similar* states that when two thoughts have the same name and similar keywords, the two thoughts are a pair. The user can select to use only the name of thoughts to find candidate-pairs. The result of the identifying candidate-pairs phase will be all candidate-pairs, where both partners have the same name. Therefore, because all couples of thoughts in the candidate-pairs are guaranteed to have same name, it is not efficient if in the comparison phase the system must compute again whether each couple of thoughts in a candidate-pair shares the same name. Instead, the system can directly compute whether the keywords are similar or not. In general, the distribution of similarity factors between the identifying candidate-pairs phase and the comparison phase can be seen as distributing the computation complexity over the two phases of the matching process. This, in turn also increases the efficiency of system resources' consumption.

In the implementation, the selection options for the comparison phase ranges from none to all. The comparison phase takes into account only the content and the context of thoughts when deciding whether two thoughts are similar or not. When the user selects none of the thought contents, the decision whether two thoughts are similar is based only on the context of the thoughts.

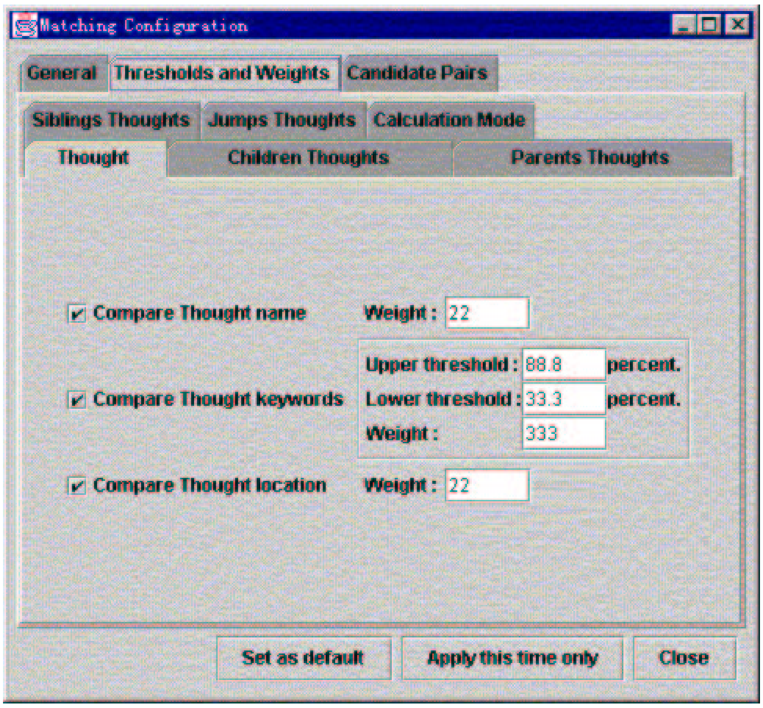
Figure 4.2 depicts the GUI for selection of thought content for the comparison phase. Here user can specify whether the system must compare the name of thoughts or not. If the names must be compared, the user needs to specify the weight of the comparison result. The same applies to the keywords and the location. For keywords there are two further entries the user must fill in, namely the thresholds. Keywords of thoughts are composed of many words, which are concatenated into one string. The system compares the keywords word by word. The result of this comparison is a percentage value of how many words from both thoughts match. By entering higher values for the thresholds, the user can put more confidence to the result, if the system finds a pair of similar thoughts.

The selection of the thoughts' content for the identifying candidate-pairs phase is coupled with the selection of the thoughts' context. Both will be described in more detail in section 4.2.3.

4.2.3 Context of Thought

The implementation of the matching process also uses the surrounding thoughts in the identifying candidate-pairs and comparison phase (see also section 3.5.2). The surrounding thoughts of an active thought are:

- ◇ Parent thoughts
- ◇ Child thoughts



☑ Compare Thought name Weight: 22

☑ Compare Thought keywords
Upper threshold: 88.8 percent.
Lower threshold: 33.3 percent.
Weight: 333

☑ Compare Thought location Weight: 22

Figure 4.2: Matching configuration GUI, active thought content

◇ Sibling thoughts

◇ Jump thoughts

The identification of candidate-pairs by the thoughts' relative position is not implemented, mainly because at the beginning of the work the implementation is designed to operate on full-mesh networks, which are brains without any additional restrictions. Using the thoughts' relative position in a network, where each thought can be linked to all other thoughts, is too complicated and time consuming.

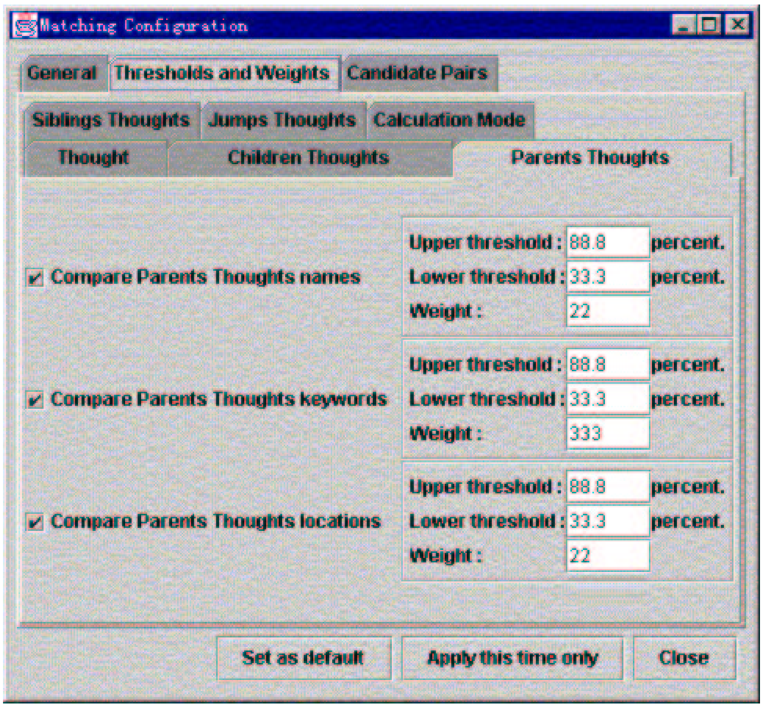
The selection of the active thought's context is similar to the selection of the active thought's content. For each category of neighbouring thoughts, e.g. parent thoughts, the user must define which fields have to be used.

Figure 4.3 shows that a user defines all fields in the parent thoughts to be used in the comparison phase. Figure 4.3 also shows that the similarity of parent thoughts' keywords is much more important than for other fields in the parent thoughts.

In the selection of similarity factors for the identifying candidate-pairs phase, the options are context and content of the active thought. For each neighbouring thought and the active thought user must choose at least one field — either name, or keywords, or location — to be used to search for candidate-pairs.

Figure 4.4 shows the parameter setting for finding candidate-pairs by using the active thought's keywords only. The context of the active thought is not used in this example.

Figure 4.5 shows the parameter setting for finding candidate-pairs by the active thought's name, keywords and location, and also by all surrounding thoughts' name, keywords and location.

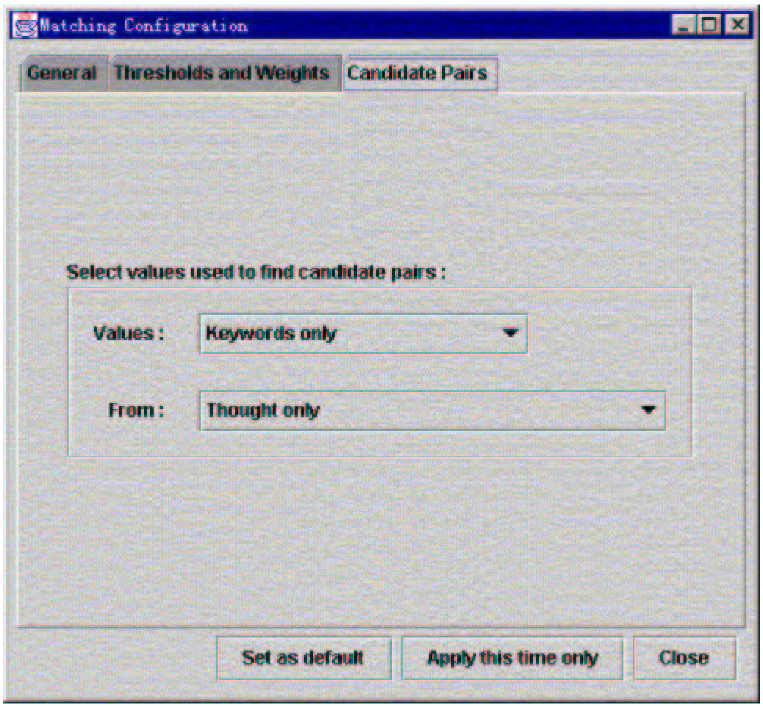


☑ Compare Parents Thoughts names — Upper threshold : 88.8 percent. Lower threshold : 33.3 percent. Weight : 22

☑ Compare Parents Thoughts keywords — Upper threshold : 88.8 percent. Lower threshold : 33.3 percent. Weight : 333

☑ Compare Parents Thoughts locations — Upper threshold : 88.8 percent. Lower threshold : 33.3 percent. Weight : 22

Figure 4.3: Matching configuration GUI, context setting for the comparison phase

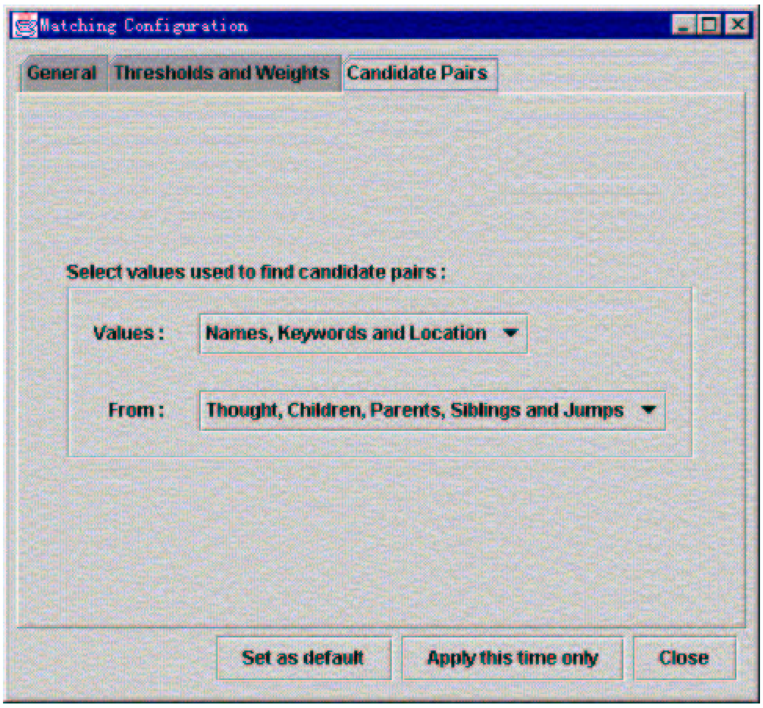


Select values used to find candidate pairs :

Values : Keywords only

From : Thought only

Figure 4.4: Matching configuration GUI, parameter setting for the identifying candidate-pairs phase (a simple example)



Select values used to find candidate pairs :

Values : Names, Keywords and Location

From : Thought, Children, Parents, Siblings and Jumps

Figure 4.5: Matching configuration GUI, parameter setting for the identifying candidate-pairs phase (a complex example)

4.2.4 Type of Thought

As stated in table 4.1, and in [2], there is no differentiation between thoughts. But when restrictions are added to the brain in order to form a tree-like Network of Content, some differentiation of thoughts can be made. For instance, an indicator in the keywords of each thought can be added, indicating whether the respective thought is a content node or a classifier node. Or it can be detected if the respective thought does not have any children (see also section 4.1), in this case the respective thought is a content node. Otherwise, it is a classifier node.

Although it is possible to make the differentiation, the physical data structure is still exactly the same between the two types of nodes. Besides, if the user really wants to make differentiation between classifier nodes and content nodes, it does not make any sense to propose a content node and a classifier node as a candidate pair, because their roles in the network are different.

The implemented matching process does not differentiate types of thoughts. For now a simpler approach has been included in the implementation of the matching process (shown in figure 4.6). The brains are divided into two parts based on the type of thoughts (classifier thoughts and content thoughts). The system runs two threads, each executing a separate matching process. One of the matching processes gets the two sub brains which contain only classifier thoughts as input, the other matching process gets the two sub brains which contain only content thoughts as input.

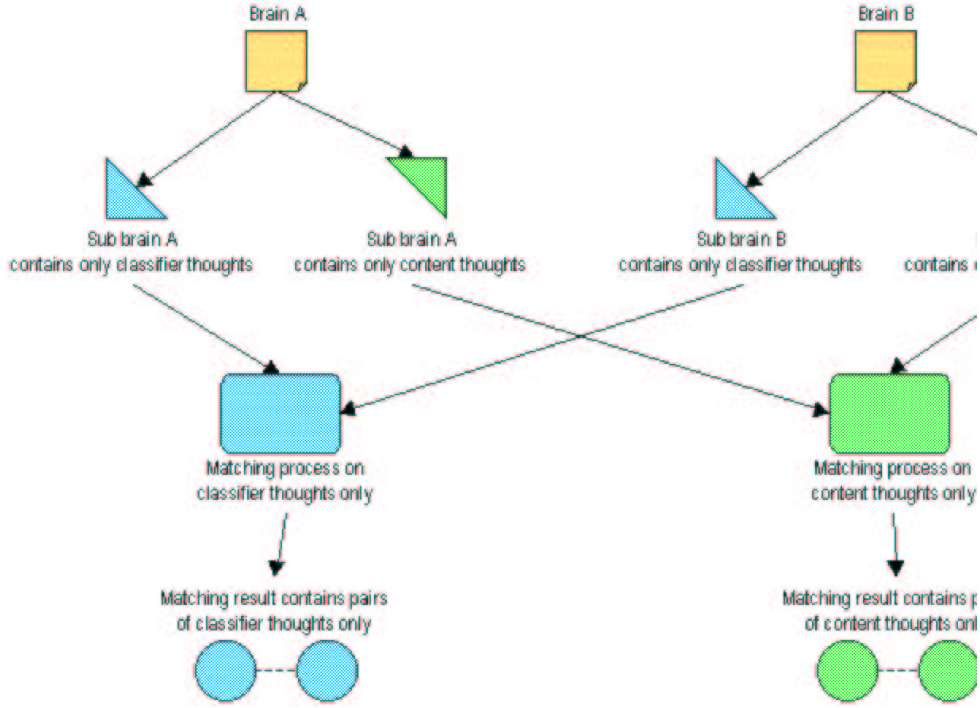

Brain A
Brain B
Sub brain A
contains only classifier thoughts
Sub brain A
contains only content thoughts
Sub brain B
contains only classifier thoughts
contains
Matching process on
classifier thoughts only
Matching process on
content thoughts only
Matching result contains pairs
of classifier thoughts only
Matching result contains p
of content thoughts onl

Figure 4.6: Processing different types of thoughts in brains

4.3 Identifying Candidate-Pairs

The identifying candidate-pairs phase uses three similarity factors to find candidates pairs, which will be explained below.

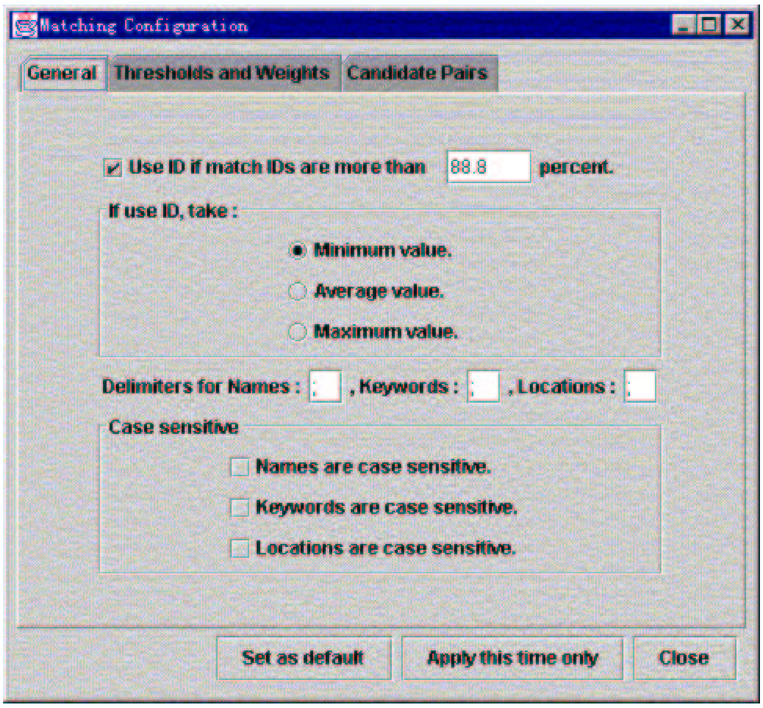
4.3.1 Using the Identifier to Identify Candidate-Pairs

This method of finding candidate-pairs can be activated and deactivated by the user in the matching configuration phase, see figure 4.7.

Considering the instability of thought identifiers in the brain, the system must detect whether the thought identifiers between the two brains are stable or not, if the user decides to use the identifiers of thoughts to find pairs. The detection and the candidate-pairs finder are both combined in one algorithm. It works as follows.

The system first tries to find thoughts from the two brains which have the same *identifier* and *name*. If two thoughts have the same **both** *identifier* **and** *name*, they are proposed as a candidate-pair. After the system has gone through all thoughts, it counts how many candidate-pairs it has found, and compares the number of candidate-pairs to the total number of thoughts in either brain. From that the system gets two percentage values, because each brain might have a different number of thoughts. Only one out of two percentage values is taken into account. The user decides whether to take the minimum, maximum or average of the two available percentage values. The options of minimum, maximum, and average can be seen in figure 4.7.

The resulting percentage value then will be compared to a threshold value, which is entered by the user in the GUI in figure 4.7. If the resulting percentage value is equal or larger



☑ Use ID if match IDs are more than 88.8 percent.

If use ID, take :

● Minimum value.
○ Average value.
○ Maximum value.

Delimiters for Names : ; , Keywords : ; , Locations : ;

Case sensitive

□ Names are case sensitive.
□ Keywords are case sensitive.
□ Locations are case sensitive.

Figure 4.7: Matching configuration GUI, general settings

than the threshold, the system decides that the thought identifiers between both brains are stable. Otherwise, they are not stable. In this case, the candidate-pairs which have been obtained will be discarded, and the system will have to find candidate-pairs through other methods (explained in section 4.3.2) from the beginning again. In case the system decides that the thought identifiers between both brains are stable, the candidate-pairs which have been obtained will be kept, and the system will have to find candidate-pairs only for the rest of the thoughts which are not in a pair yet.

4.3.2 Using Content and Context to Identify Candidate-Pairs

As has been explained in section 4.2.3, the implementation of the identifying candidate-pairs phase uses the context of the active thought to find candidate-pairs for it. The implemented method of finding candidate-pairs can be explained with the following example.

As in figure 4.4, a user defines that the system uses only the active thought's name to find candidate-pairs. The system takes the second brain, and goes through all thoughts in that brain. While going through the second brain, the system fills a sorted hash table. The keys of the hash table are the thought names from the second brain. Keys are unique. The values of the hash table are the identifiers of thoughts, which have name the same as the respective key.

Table 4.2 shows a sample brain, which is used as the second brain. The hash table created for this brain is shown in table 4.3. In the second brain there are two thoughts which have the same name 'Management'. In the hash table, the two thoughts (with identifier 3 and 5) are represented by a single key ('Management'), because their names are the same.

Table 4.4 shows the first brain. The system goes through the first brain, takes one thought

Thought identifier	Thought name
1	Market
2	Finance
3	Management
4	Marketing
5	Management
6	Technical
7	Human Resource

Table 4.2: A simple brain, second brain

Name	Set of identifiers
Finance	2
Human Resource	7
Management	3, 5
Market	1
Marketing	4
Technical	6

Table 4.3: The hash table from the second brain

Thought identifier	Thought name
1	Finance
2	Market
3	E-Business
4	E-Marketing
5	Management
6	Technical
7	Human Resource

Table 4.4: A simple brain, first brain

at a time, extracts the thought's name, and looks up to the hash table of the second brain to see if the name exists as a key in the hash table. If the name exists in the second brain's hash table, the system will get the value. The value, together with the thought identifier from the first brain, is filled into another hash table, called candidate-pairs list.

If e.g. the system takes the second thought in the first brain, it gets the name, which is 'Market'. It looks up in the second brain's hash table, whether the string 'Market' exists here as a key. It finds that the string exists, and gets the value of the key 'Market' which is identifier number 1. The system then inserts a new entry into the candidate-pairs list, with identifier number 2 (the thought's ID in first brain) as key, and identifier number 1 (the thought's ID in second brain) as value. Table 4.5 shows the complete list of candidate-pairs. Table 4.6 shows the thought identifiers which do not have a pair. They are called singles.

Thought identifier from the first brain	**Thought identifiers from the second brain**
1	2
2	1
5	3, 5
6	6
7	7

Table 4.5: Candidate-pairs list

Single thought identifiers from the first brain	**Single thought identifiers from the second brain**
3, 4	4

Table 4.6: Singles list

If the user defines more complex parameter settings for the identifying candidate-pairs phase, which includes the context of the active thought (e.g. like in figure 4.5), the process of finding candidate-pairs remains principally the same, the system only works on a different set of data.

Instead of considering only the thought name, the system can also consider e.g. parent thoughts' names during the whole procedure. The key of second brain's hash table is not only derived from active thought name, but also includes the parents' thought names. When the system goes through the first brain, it analogously extracts not only the active thought name, but also the parents' thought names, and uses each string to look up the second brain's hash table. In consequence, the second brain's hash table, and the candidate-pairs list get longer the more similarity factors are included in the process of finding candidate-pairs. The computation also takes longer, because the system must consider more things.

4.4 Comparison, Computation of Similarity

In the comparison phase, the system uses the candidate-pairs list, the result from previous phase. The system goes through the list pair by pair. If there are more than one thought identifier from the second brain, the system takes the identifiers from the second brain one by one to be paired to the identifier from the first brain. Take the third row of table 4.5 as an example where the system will create two candidate-pairs, which are (5, 3) and (5, 5). For each pair, the system will compare the similarity factors of the active thought, according to the user settings in the matching configuration phase.

4.5 Results Evaluation

For the merging implementation based on this work, the result of the matching process is preferred to be one pair per thought. Therefore, the matching process implements several strategies for filtering out the best pair (see also section 3.7).

The matching process implements a threshold strategy, so that every pair proposed by system satisfies a certain level of user expectation. The threshold strategy divides the result into three categories. Each category represents a certain level of system confidence regarding the pairs in this category. The categories are:

Matched Pairs. These pairs are couples of thoughts which are similar with a high probability. In the subsequent merging process, matched pairs are not examined. Instead, the user can choose a matched pair and explicitly splits it if he/she feels that the nodes are not similar after all.

Proposed Matched Pairs. These pairs are couples of thoughts which are similar with a lower probability. In the subsequent merging process, proposed matched pairs have to be explicitly confirmed by the user.

Singles are thoughts which do not have any partner from the other brain. When merging, a user can match single nodes by hand or treat them as newly inserted thoughts.

For example, the user defines that the comparison phase uses the active thought's name, parents' names and children's names. In order to be qualified as *matched pair*, each couple of thoughts must have the same name, the similarity of the parents' names must be above the upper threshold, and the similarity of the children's names must be above the upper threshold. In order to be qualified as *proposed matched pair*, each couple of thoughts must have the same name, and the similarity of the parents' names must be above the lower threshold, and the similarity of the children's names must be above the lower threshold. If neither of those conditions is satisfied, each thought in the compared couple is listed as *single*.

If there is more than one pair per thought qualified into a category, the system needs a way to evaluate which of those pairs is best. Here weights play an important role. By using the weights, each pair will have a weighted similarity. The computation of the weighted similarity is also done in the comparison phase. By evaluating the pairs weighted similarity the system can decide which pair is the best. But there is a possibility that two pairs have the same weighted similarity. In this case the user has to decide which pair is the best.

4.6 Diagrams

Figure 4.8 shows a class diagram of the matching process implementation. Only a few important classes are included in the figure. The whole classes are packed into one Java library called *BrainMatching* package.

The main class in the implementation is the class *SubBrainMatching*. An object of this class represents a matching process as a whole. From this object, an interface for configuring the whole matching process can be obtained, which is the *MatchingConfiguration* interface. After the configuration is finished, the system invokes the *execute* method in the *SubBrainMatching* object. The *getMatchingResultSet* method will return another interface from which the details of the result can be obtained.

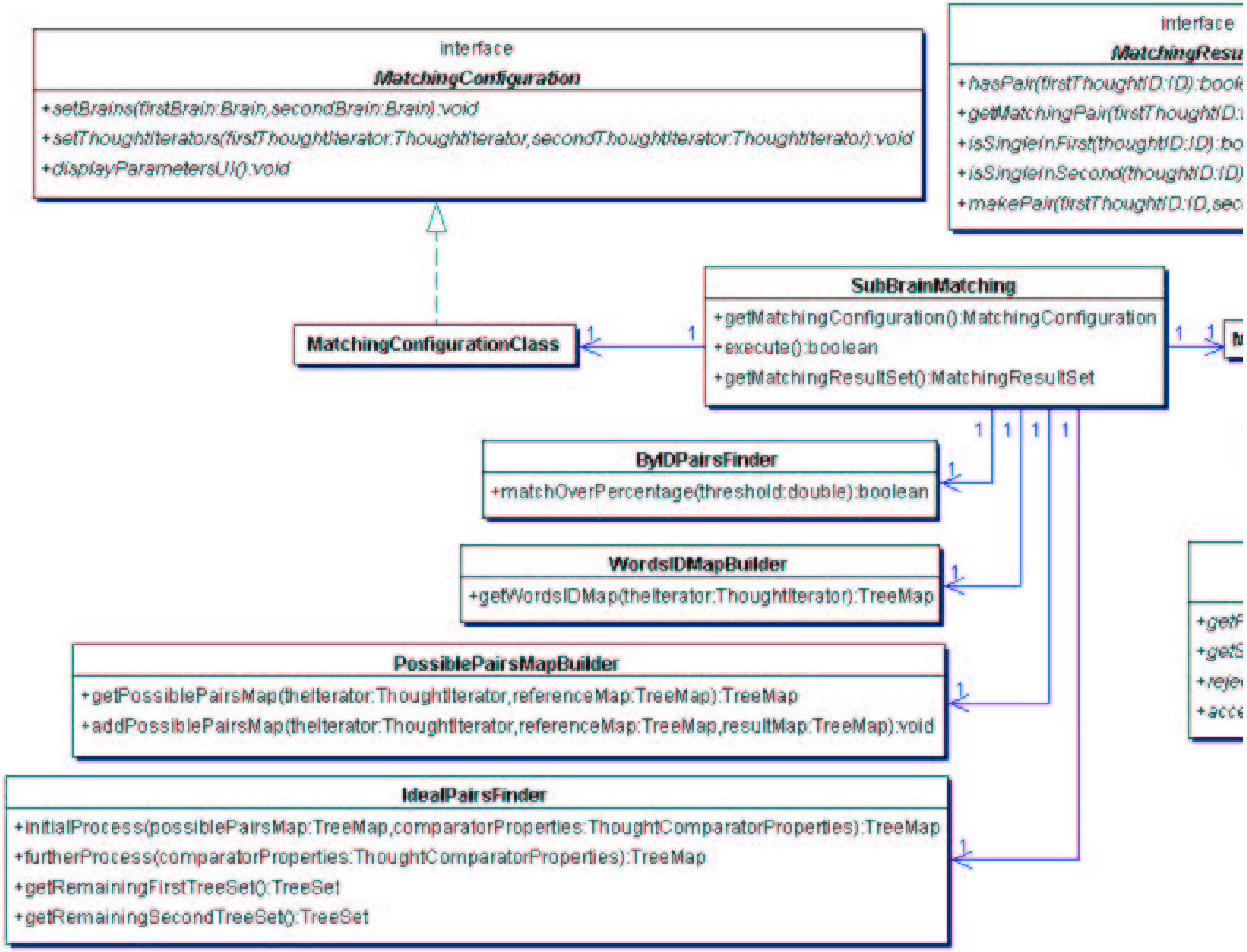

interface
MatchingConfiguration
+setBrains(firstBrain:Brain,secondBrain:Brain):void
+setThoughtIterators(firstThoughtIterator:ThoughtIterator,secondThoughtIterator:ThoughtIterator):void
+displayParametersUI():void
interface
MatchingResu
+hasPair(firstThoughtID:ID):boole
+getMatchingPair(firstThoughtID:
+isSingleInFirst(thoughtID:ID):bo
+isSingleInSecond(thoughtID:ID)
+makePair(firstThoughtID:ID,sec
MatchingConfigurationClass
1
1
SubBrainMatching
+getMatchingConfiguration():MatchingConfiguration
+execute():boolean
+getMatchingResultSet():MatchingResultSet
1
1
M
1
1
1
1
ByIDPairsFinder
+matchOverPercentage(threshold:double):boolean
1
WordsIDMapBuilder
+getWordsIDMap(theIterator:ThoughtIterator):TreeMap
1
PossiblePairsMapBuilder
+getPossiblePairsMap(theIterator:ThoughtIterator,referenceMap:TreeMap):TreeMap
+addPossiblePairsMap(theIterator:ThoughtIterator,referenceMap:TreeMap,resultMap:TreeMap):void
1
IdealPairsFinder
+initialProcess(possiblePairsMap:TreeMap,comparatorProperties:ThoughtComparatorProperties):TreeMap
+furtherProcess(comparatorProperties:ThoughtComparatorProperties):TreeMap
+getRemainingFirstTreeSet():TreeSet
+getRemainingSecondTreeSet():TreeSet
1
+getF
+getS
+reje
+acce

Figure 4.8: Class diagram of the matching process implementation

The matching process itself is executed by several classes. *ByIDPairsFinder* detects whether thought identifiers between two brains are stable or not. If the identifiers are stable this class builds a candidate-pairs list (e.g. table 4.5) based on thought identifiers (see also section 4.3.1). *WordsIDMapBuilder* builds the second brain's hash table (e.g. table 4.3) discussed in section 4.3.2. *PossiblePairsMapBuilder* builds the candidate-pairs list (e.g. table 4.5) also discussed in section 4.3.2. The *IdealPairsFinder* controls the comparison phase and evaluates the resulting pairs, as has been discussed in section 4.4 and 4.5.

The result of the matching process is returned as a composition of pairs, encapsulated in the *MatchingResultSetClass*. Each pair is represented by a *MatchingPairClass* object. From the *MatchingPairClass* details of the pair's information, e.g. the level of the thoughts' similarity, can be obtained.

A user can interact with the system's result set, with regards to the user decision strategy discussed in section 4.5. The interface *MatchingResultSet* and the interface *MatchingPair* also provide a method for the user to *reject* a pair and a method to *make* a pair based on his/her own judgements. If the user rejects a pair, the thoughts in the pair will be listed as single thoughts. The user can also take two thoughts, each from a different brain, and make them a pair. These two methods support the interactive mode of the matching process.

Figure 4.9 shows the same classes as in figure 4.8 in activity diagram, involved in the flow just described above.

4.7 Evaluation of the Implementation, and Future Work

The implemented results evaluation phase filters out a candidate-pair when one of the pair's similarity factors falls below the respective threshold. When a candidate-pair is filtered out, the pair is broken by the system. The two thoughts in the pair are listed as singles.

If there are two pairs or more for one thought after the first screening of the candidate-pairs, the weights are used to select the best pair. This means that the first screening process filters out candidate-pairs only by the level of similarity between the thoughts. The importance of similarity factors, which is also represented by weights, does not have any influence on the first screening of candidate-pairs. For future work, it is better to filter out candidate-pairs based on both the level of similarity between thoughts as well as the importance of the similarity factors themselves. Existing approaches from fuzzy logic [11] can be used here.

The following is an example for the current implementation and a possible improvement. In a matching process, two similarity factors are considered for the comparison.

◇ One similarity factor, e.g. the active thought's keywords, has an upper threshold of 50%, and a weight of 3.

◇ The other similarity factor, e.g. the parents' keywords, has an upper threshold of 25% and a weight of 1.

Now, there are three pairs, lets call them pair C, pair D, and pair E.

◇ Pair C has a 49% similarity for its active thought's keywords, and a 100% similarity for its parents' keywords.

◇ Pair D has a 51% similarity for its active thought's keywords, and a 33% similarity for its parents' keywords.

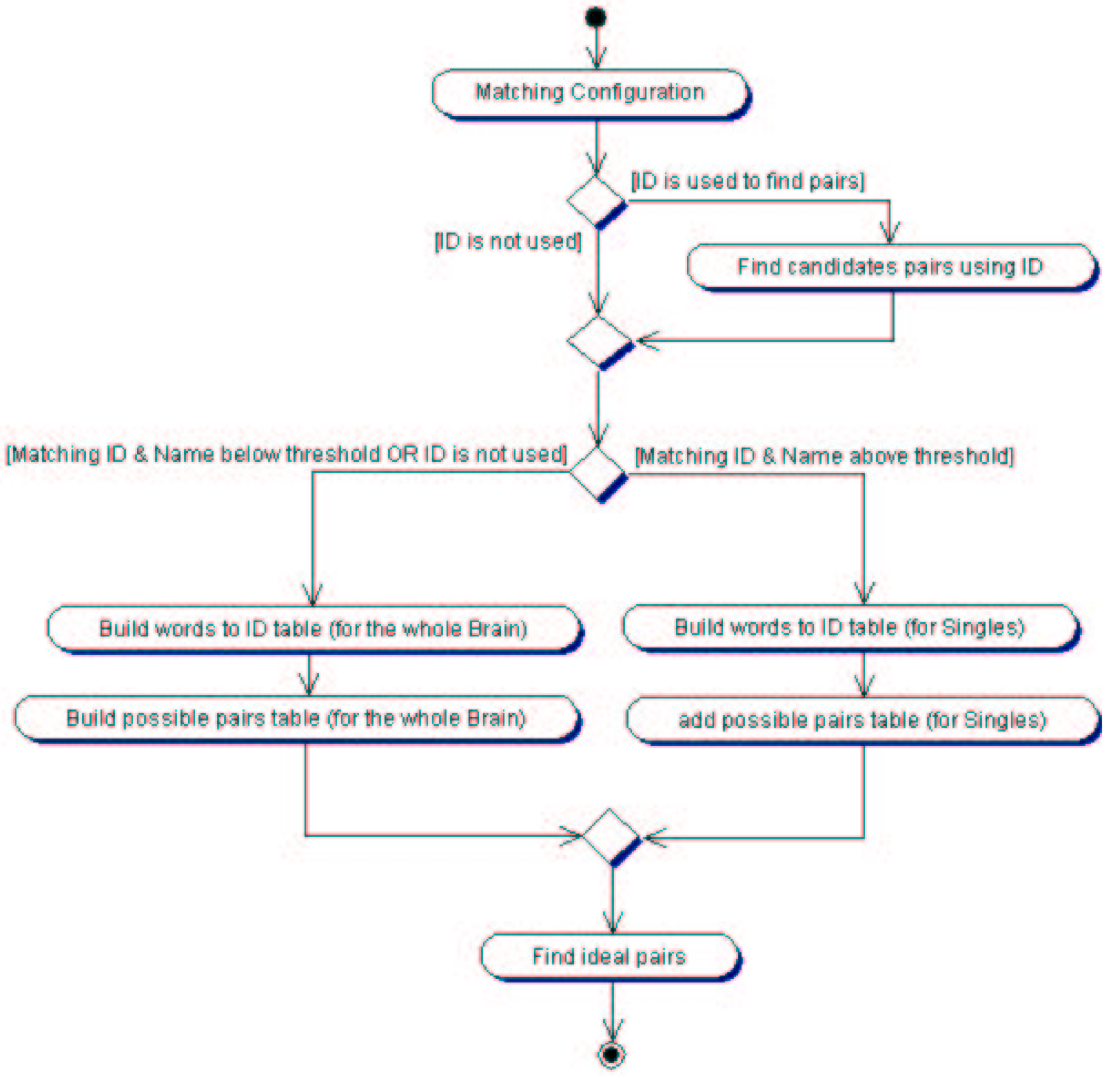

Matching Configuration
[ID is used to find pairs]
[ID is not used]
Find candidates pairs using ID
[Matching ID & Name below threshold OR ID is not used]
[Matching ID & Name above threshold]
Build words to ID table (for the whole Brain)
Build words to ID table (for Singles)
Build possible pairs table (for the whole Brain)
add possible pairs table (for Singles)
Find ideal pairs

Figure 4.9: Activity diagram of the matching process implementation

- ◇ Pair E has a 52% similarity for its active thought's keywords, and a 40% similarity for its parents' keywords.

The total *weighted similarity* of the pairs — a similarity which also considers the importance of each similarity factor — can be calculated as follows.

- ◇ Pair C:

$$\frac{(49\% \times 3) + (100\% \times 1)}{3 + 1} = 61.75\%$$

- ◇ Pair D:

$$\frac{(51\% \times 3) + (33\% \times 1)}{3 + 1} = 46.5\%$$

◇ Pair E:

$$\frac{(52\% \times 3) + (40\% \times 1)}{3 + 1} = 49\%$$

In the current evaluation phase, pair C will be filtered out because its active thought's similarity is below the threshold, and pair E is selected as the best pair.

The values of name, keywords and location of thoughts are strings. Therefore a good string matching is required for this work. However, this work is not about string matching, but it is about Network of Content matching. The current matching process implementation uses a simple string matching, included in the standard Java library. Better string matching algorithms can be used instead of just a simple string matching, for instance, string matching algorithms that anticipate typos, string matching algorithms that match words with similar meanings, string matching algorithms that apply stemming, or multilingual matching algorithms.

The implemented matching process computes pairs for all thoughts at once. As the consequence, when the brains being processed have thousands of thoughts to be compared, the processing time becomes intolerable. This work is used in [16], a merging process. The merging process implemented for [16] has an interactive mode. In this case it should be possible to compute pairs only for the current active thought seen by the user. In this way, the compute time will be less noticeable, because the system is practically idle while waiting for user interaction, or in between user interactions. This idle time can be used to compute the pairs. Processing time could also be speeded up by preprocessing algorithms which can disburden the comparison phase.

The behaviour of the matching process and its optimal result depends tightly on the parameters set in the matching configuration phase. Therefore, the parameters' setting is an important issue for the user. Unfortunately, to set the parameters correctly is not a trivial task. Some sets of default values of the parameters, which have been tested in some general context, should be provided for each typical application of the matching process. For instance, a set of values that have been tested on a number of brains for consolidation merging or for reconciliation merging could be put up in advance. A selection in the matching configuration GUI should be added for the user. The user can choose whether he/she wants to use the pre-set values or likes to define his/her own parameter set. It should also be possible for the user to store his/her own parameters' setting in a file, and reload it later. In this way a user can store many sets of parameters, which are customised to his/her own needs.

Appendix A

Matching Configuration manual

The following is a guide for a new user of the implemented matching process. Since the number of parameters are numerous and they all depend on what kind of matching scenario this implementation is used for, and also on what kind of brains the implementation is working on, the exact parameters' setting can not be determined by numbers here. Experience on the usage is the best guidance. The following should help to get a first idea of how to set up the parameters.

A.1 Matching brains which have a common predecessor

In this scenario of matching, the identifiers of thoughts might be used. The percentage value depends on the degree to which the brains have been updated. If e.g. only 10% of the brain have been updated, the match identifier threshold value is just slightly lower than 90% (see also figure 4.7).

This scenario can be further categorised based on what kind of update happens between the versions of the brain.

Moving thoughts around in the brain. In this working scenario, the surrounding thoughts might change severely. Therefore it is better to rely only on the content of the active thought itself to find candidate-pairs and to compare the thoughts. Therefore, the keywords threshold and the weights of the active thought's name, keywords and location are preferred to be much larger than what is set for the surrounding thoughts.

The threshold value of the active thought's keywords depends on how many keywords each thought has on the average, and on how many must match for similar thoughts. The weights are relative values in a sense that their absolute size entered by the user does not matter. What is important is the relation between all of weights' values. So, it is the same whether a user enters: 10, 20, 30 or a user enters: 1, 2, 3.

For example a classifier thought might have many child thoughts, and the user updates the brain by splitting this classifier thought into two (or more) new classifier thoughts and distributes the child thoughts between the new classifier thoughts. In this case, the old classifier thought is deleted from the brain, and the two new classifier thoughts are added to the brain. Since most of the thoughts will still have the same child thoughts, the user can use these child thoughts together with the active thought's content. Most of the thoughts still have the same parents, too, but since the number of parent thoughts in a tree-like Network of Content is far less than the number of children, it is better to use the child thoughts. In order to find candidate-pairs and compute the similarity between the thoughts more accurately, more samples for the similarity factors are needed.

If there is no update of thought content at all, then the thresholds of the active thought's keywords can be set up to 100%, with weights larger than child thought weights. If the user wants the old classifier thought (the one which is replaced by the new classifier thoughts) to be paired with one of the new classifier thoughts, the thresholds are preferred to be small percentage values from 0% up to the maximum percentage of child thoughts the new classifier thought gets from the old classifier thought. The smaller the value the better the result, although this takes more memory. Other surrounding thoughts are better ignored (uncheck the options for comparison of the respective similarity factors). For finding candidate-pairs set the options to 'Names and Keywords' and 'Thought and Children'.

In case that the new classifier thoughts are inserted beneath the old classifier, instead of replacing the old classifier, then user can do the same thing as stated above, but the thresholds for child thoughts is set to 0%, in order to make sure that the old classifier thoughts are matched correctly.

Editing thoughts in the brain. If the update involves changing some of the thoughts' names or locations without the intention of changing the thought entirely, the respective active thought's name or location must not be used in the comparison phase. But the name must be used in finding candidate-pairs, since not all thought names are changed.

As a general suggestion, it not a good idea to use the location of thoughts as a similarity factor, since the string representing the location of the actual content is only present in content thoughts. By involving a thought's location in the comparison phase, the system is wasting resources when comparing classifier thoughts. Further more the location string is usually pretty long compared to a thought's name or keywords. Therefore, the system will need more memory.

It is assumed that most of the editing does not change the corresponding thought entirely, rather a slight change is made on the thoughts. Only a few thoughts are changed drastically. This fact also represents the common practice of cooperative developing of a Network of Content. If the majority of the thoughts is changed drastically, the parameters for separately developed brains can be used instead.

With this assumption, the following conclusion can be made: Most of the surrounding thoughts are still similar. Therefore, all surrounding thoughts can be used.

The threshold value of the active thought's keywords is preferred to be smaller compared to keywords from child thoughts. This is because more keywords exist in all child thoughts compared to the number of keywords in the active thought. If the number of parents is smaller than the number of children, than the threshold values for the parent thoughts are preferred to be smaller, too. This also applies to thought names.

Weight values can be set according to the importance of the respective similarity factor. Preferably, the active thoughts' weight values are the largest. The second largest is the weight for parent, child and sibling thoughts.

A.2 Matching brains which do not have a common predecessor

For this kind of application, it is assumed that most of the differences between brains are overlapping. The structure of classification itself is not guaranteed to be similar. Therefore, the threshold values for all thoughts are preferred to be small. The identifiers of thoughts are not recommended to be used. The same applies to the location of thoughts.

The names of thoughts, which represent the same concept, most probably are different. It is better not to compare them. But use the name of thoughts for finding candidate-pairs. Keywords fields have more values compared to the name fields, therefore it is better to use the keywords only. Jump thoughts are better excluded. It is better to do the comparison based by hierarchical structure only. Cross-references like jump thoughts will most likely only make things more complicated without any promising result. For finding candidate-pairs set the options to 'Names and Keywords' and 'Thought, Children, Parents and Siblings'. For a faster computation sibling thoughts can be excluded also.

Weight values can be set according to the importance of the respective similarity factor. The weight values of the active thought are preferred to be the largest. The second largest is the weight for parent, child and sibling thoughts.

Bibliography

[1] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montréal, Québec, June 1996.

[2] TheBrain Technologies Corporation. *BrainSDK Documentation.* TheBrain Technologies Corporation, Santa Monica, CA, BrainSDK version 2.1 edition, October 2000.

[3] Vladimír Dančík. *Expected Length of Longest Common Subsequences.* PhD thesis, Department of Computer Science, University of Warwick, September 1994.

[4] L. Dempsey and R. Heery. Metadata a Current Review of Practice and Issues. *Journal of Documentation*, pages 145–172, 1998.

[5] Anne J. Gilliland-Swetland. Defining Metadata. In Murtha Baca, editor, *Metadata: Pathways to Digital Information*, pages 1–8. Getty Information Institute, Los Angeles, 1998.

[6] Yahoo! Inc. http://www.yahoo.com. February 2001.

[7] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland, November 1992. Report A-1992-6.

[8] Torsten Lenk. Flexible Versionierung semistrukturierter Dokumente im Kontext von Autonomie und Kooperation. Diplomarbeit, Fachbereich Elektrotechnik, Technische Universität Hamburg-Harburg, Hamburg, Germany, January 2001.

[9] Natrificial LLC. *The Brain™ User Manual.* Natrificial LLC., www.thebrain.com, June 1998. Companion to Version 1.5 of The Brain.

[10] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files : diff, diff3, sdiff, cmp, and patch*, 1.3 edition, September 1993. For diff 2.5 and patch 2.1.

[11] Florian Matthes and Ulrike Steffens. PIA - A Generic Model and System for Interactive Product and Service Catalogs. In Serge Abiteboul and Anne-Marie Vercoustre, editors, *Research and Advanced Technology for Digital Libraries, Proceedings of the Third European Conference, ECDL '99, Paris, France, September 1999*, volume LNCS 1696 of *Lecture Notes in Computer Science*, pages 403–422. Springer-Verlag, September 1999.

[12] Jonathan P. Munson and Prasun Dewan. A Flexible Object Merging Framework. In *Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative Work, October 1994, Chapel Hill, NC*, pages 231–242. ACM-Press, October 1994.

[13] NetMind. User Help for Mind-it. http://mindit.netmind.com/help.shtml, February 2001.

[14] Claudia Niederée. *E-MergeNC : A Tool for Merging Networks of Content.* Working draft, Department of Information and Communication Technology, Technische Universität Hamburg-Harburg, Hamburg, Germany, November 2000.

[15] Christoph Spreen. Change Management für Metadaten in Digitalen Bibliotheken. Diplomarbeit, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Germany, Februar 2000.

[16] Siripong Treetasanatavorn. Semi-automatic Merging of Content Networks: Policy-based Customization. Studienarbeit, Technische Universität Hamburg-Harburg, Hamburg, Germany, 2001.