The Development of a Software Clone Detector

N. Davey*, P.C. Barson†*, S.D.H. Field†*, R.J. Frank*, D.S.W. Tansley†

†BNR Europe Limited
London Road, Harlow, Essex, UK, CM17 9NA
P.C.Barson@bnr.co.uk, S.D.H.Field@bnr.co.uk,
D.S.W.Tansley@bnr.co.uk

** University of Hertfordshire*
College Lane, Hatfield, Herts, UK, AL10 9AB
N.Davey@herts.ac.uk, R.J.Frank@herts.ac.uk

Abstract: Cloning, the copying and modifying of blocks of code, is the most basic means of software reuse. Code cloning has been very extensively used within the software development design community. Unofficial surveys carried out within large, long term software development projects suggest that 25-30% of the modules in this kind of system may have been cloned. A system to detect clones of procedures in large software systems is described. The system uses a self organising neural net, a SOM, to cluster feature vectors associated with the procedures. We report how a prototype system was developed and subsequently enhanced into a full product quality tool. The limitations of the SOM-based tool are the long training times and the fixed number of clone classes that are created. A second neural net model, the Dynamic Competitive Learning (DCL) net, which overcomes both these limitations is also discussed as a possible component: the results of our initial trials with the SOM-based tool and the DCL-based prototype are given.

1 Introduction

In 1992 a collaboration began between the Software and Systems Engineering department of BNR Europe and the University of Hertfordshire, to investigate the problem of the detection of clones in software systems. Initially a prototype using a self organising neural net was built and, as this proved to be successful, a fully functional, product quality tool has been constructed and tested. The tool is now in use. This project has been unusual as an application of neural computational technology in that the problems of the integration of this type of technology into conventional software systems, with their inherent need for quality interfaces and reliable functionality, has been tackled and overcome.

In this paper we discuss the problem of software cloning and its relation to software reuse. We describe the initial prototype and show how it was modified to produce a full system. During the productisation phase a second type of neural net was investigated and tested in an effort to improve some aspects of the current system. The new network architecture is described here and we give our initial results.

Section 2 discusses software cloning, Section 3 describes how self organising neural nets can be used to identify such clones and Section 4 describes the final system. Section 5 describes the alternative neural network model we are investigating, Section 6 presents an evaluation of the project and Section 7 concludes.

2 Software Cloning

2.1 Large Software Systems

In 1968 the NATO Science Committee convened a meeting of top programmers, computer scientists and captains of industry to discuss the difficulties and concerns with building large software system, and it was at this meeting that the term *software engineering* was coined. In spite of the concerns at that time, over the last 25 years the size and complexity of software has increased tremendously. For instance, the amount of code in most consumer products is doubling every two years - televisions may contain up to 500 kilobytes of software, and even an electric shaver may have 2 kilobytes (Wayt Gibbs, 1994).

A typical digital telephone exchange in the 1970’s would have possessed a software system of a few hundred thousand lines of source code. Today a digital telephone exchange may incorporate tens of millions of lines of code. Such systems have evolved over the years, with the current systems incorporating code from the very first system. The development of these systems began in a time when software development methods were in their infancy. For example, structured analysis and design were not the norm and the sophisticated re-usability of object oriented implementation techniques were not in use outside computer science research laboratories.

Error rates increased as these systems grew in complexity. Errors in real-time software are notoriously difficult to detect because they often occur only under certain conditions. It is often not practical to re-build existing systems using new techniques and technology because of the immense costs involved, with the average cost for well-managed code running at over $100 per line (Wayt Gibbs, 1994).

Improving software product quality, performance and development team productivity has therefore become a primary priority for almost every organisation that relies on computers (Moller & Paulish, 1993). For instance, in Table 1, the US NASA Space Shuttle software system required 25.6 million lines of code, 22,096 staff-years of effort, and cost $1.2 billion. With the large capital investment necessary to develop such software then the management of these systems becomes ever more vital, both during and after the actual development. It is as much a managerial problem as a technical one.

As an example of the maintenance concerns, United Airlines Covia subsidiary spends $120 million each year maintaining and updating software for its Apollo reservation system, and according to KPMG Peat Marwick, this activity typically consumes 80% of most corporate software budgets (Schendler, 1989).

Table 1: Typical Software Application Size and Investment [Moller & Paulish, 1993; Schendler,1989]

Product Application	Size (million lines of code)	Development Cost ($millions)	Manpower Required (staff-years)
Space Shuttle	25.6	1 200	22 096
Operating System	2 – 5	150 – 350	
Mid-sized Communication System	1 – 2	50 – 100	
Citibank Teller Machine	0.78	13.2	150
Lotus 1-2-3 v3	0.4	22	263
IBM Checkout scanner	0.090	3	58
1989 Lincoln Continental	0.083	1.8	35

2.2 Cloning

The copying and modifying of blocks of code constitutes the technique of cloning - it is the most basic means of software reuse. Code cloning has been very extensively used within the software development design community. Unofficial surveys carried out within large, long term software development projects suggest that 25-30% of the modules in this kind of system may have been cloned. In some cases entire modules had been cloned though more usually part of the module was copied and used in another module.

The main advantage of cloning, as a means of software reuse, is that it is very simple to do. It can provide a development team with a quick start, and often a rapid solution to the problem. Copying procedures that have a function close to that required, and then modifying that code, allows customisation without any software ownership negotiations. The designer copying the procedure does not have to discuss with the original designer the procedure interface or function modification required, or wait for the original designer to update the procedure to incorporate the new functionality. It is obviously particularly tempting if contractors are paid by lines of code produced.

2.3 Problems of Software Systems with Cloned Software.

Software cloning is a major contributor to the steadily increasing complexity of software systems. Whilst it is anticipated that copying working code will produce correct new code, there is a danger that as yet unknown bugs may be copied and that the resulting code may include large numbers of redundant statements. Cloning produces a rapidly expanding library of procedures which have to be maintained; so that if the situation arises where a modification or enhancement is required to be made to an original piece of code then it may also be required in the cloned copies. This propagation is time consuming and may be impossible due to the likely lack of documentation of the cloning process. Additional problems may arise from the cloning process itself: the engineers of cloned software may be unaware of dependencies involved in the original software, causing the new code to be error prone.

Cloned software is therefore a major problem in large software systems that have evolved over a long period of time. The problem is not amenable to a simple solution - to completely re-engineer the system is a possible answer, but one of great cost and potential difficulty. Alternatively an attempt can be made to track down and log potential clones, and in this way manage the problem. It is this approach we are using, and to do so we have built a tool to detect clones.

An additional benefit of such a tool is that software components that have been cloned on several occasions can be identified. These units of code will therefore have demonstrated their reusability and this suggests that populating a software library with such code will provide an effective set of reusable components.

2.4 Types of clone

Some clones are easy to spot, for example, similar names may be used between related procedures, or it may even be noted in the cloning programmer's comments to the code, such as: "Cloned from..."! However, there are many more subtle clones and it is these that a clone detection tool will prove most useful for, as well as for finding the more obvious clones.

In general, clones may be described using the following typology:

Type I	An exactly identical source code clone, i.e. no changes at all.
Type II	An exactly identical source code clone, but with indentation, comments, or identifier (name) changes.
Type III	A functionally identical clone, but with small changes made to the code to tailor it to some new function.
Type IV	A functionally identical clone, developed possibly with the originator unaware that there is a function already available that accomplishes essentially the same function.

These types follow an increasing level of subtlety from type I through IV. Similarly, the level of sophistication required by any process designed to detect such clones, increases from I through IV - with type IV being particularly hard to detect without a great deal of background knowledge about program construction and software design. This scale applies whether the process is automatic or not.

Whilst a neural net approach could be used to address any of these types, currently we are addressing the problem of types I, II and III, which form by far the majority of clones present in existing systems. Future work may address type IV.

3 A Neural Computational Solution

3.1 Potential Neural Network Models

Unsupervised neural nets provide a powerful technique for clustering unlabelled data. They are particularly suited to the kind of problem described here as they scale up well to very large data sets and are a robust and relatively straightforward technology to use. To employ such networks the data set must first be encoded as a set of real valued vectors, which will then constitute the training set. The way in which this was accomplished is

described in section 3.2. After the net is trained the data set will be partitioned into a set of clusters of similar vectors. With most unsupervised neural nets the weight vector of the classifying output unit will be the centroid of its class, so that the network is performing a form of k-means clustering (Butchart et al, 1995a).
Moreover any novel vector can be presented to the trained net, which will identify the class in the training set to which the novel vector belongs, and therefore will identify the likely clones of this software module. More formally the set of N software modules: $\{S_i\}_{i \le N}$ is first converted to a set of vectors: $\{v_i\}_{i \le N}$. An unsupervised neural net consisting of M output units (with M << N), $\{\mathrm{X}_j\}_{j \le M}$ is then trained on these vectors and will produce a classification of the training set. That is a discrete partition of the training set is produced:

$$c:\{v_i\}_{i \le N} \rightarrow \{\mathrm{X}_j\}_{j \le M}$$

so that the set of clusters: $\{c^{-1}(\mathrm{X}_j)\}_{j \le M}$ identifies the potential clones.

For a system design team the major issue to be initially confronted with is the specific choice of neural net architecture. The available models were: simple competitive and its refinements, ART (Carpenter & Grossberg, 1988) type networks and self organising maps (SOMs) (Kohonen 1990). At a later stage a more recent model, the dynamic tree network (DCL) (Racz and Klotz, 1991) became available.

Our analysis of this choice was informed by a study we undertook on a different, albeit, related data set, in which the three unsupervised neural nets were investigated (Field et al, 1995). Here, data representing the complexity structure of software units was clustered and analysed. Our results showed favourable performance from the SOM type model.

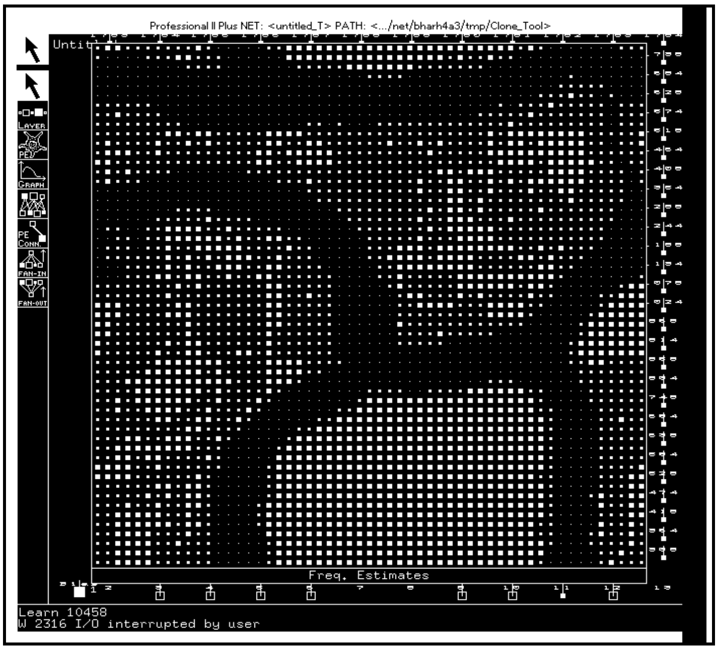
An important feature of the SOM is that the classification, or output, units are arranged in a two dimensional grid: the $\{\mathrm{X}_j\}_{j \le M}$ are each given a Cartesian co-ordinate (x_j, y_j). This is useful in two respects:

- Any global structure in the data may be extracted.
- As can be seen in Figure 1, three distinct areas in the classification space are visible.

- Neighbourhoods of output units, in their Cartesian space, provide additional, similar clusters. So that in looking for clones of software module A, the search can be extended to the neighbours of the unit that classifies A.

3.2 Representation of software units as fixed length vectors

As described above the training set of source code modules had to be represented as a set of feature vectors. Such an encoding is central to the success of an application such as this since it is imperative that similar blocks of code have similar representational vectors. It is important to note here that the notion of 'similar' is different in either case; for the source code similarity relates to the probability of the code being cloned, and for the corresponding vectors that they are close in Euclidean space. More specifically we require that if software unit A is represented by vector $\mathbf{v}_A$ and B by $\mathbf{v}_B$ then if B is a clone of A $\|\mathbf{v}_A - \mathbf{v}_B\|_2$ should be relatively small.



Professional II Plus NET: <untitled_T> PATH: <.../net/bharh4a3/tmp/Clone_Tool>

Freq. Estimates

```
Learn 10458
W 2316 I/O interrupted by user
```

Figure 1. Frequency activation of the nodes contained in a 55 by 55 SOM trained on 10,257 vectors.

In fact the set of source code units can be thought of as a set of indexed taxonomic trees, where the root of each tree corresponds to the first occurrence when a unit was cloned and branching points further down the tree correspond to further instances of cloning — the whole structure is analogous to a set of phylogenetic trees. The index of each tree denotes the degree of divergence of the clone from its parent, see Figure 2.

Due to the equivalence of indexed hierarchical trees and ultrametric (Rannal et al, 1986) spaces it can be seen that the task of detecting clones becomes one of inducing an appropriate ultrametric in the space of representational vectors. Clustering the source code vectors with a SOM provides a similarity metric, whereas our newer model produces the desired ultrametric - the vectors are organised into a tree structure.

3.3 Representation Vector Definition

To represent all the information in a source code unit would not be feasible due to the resulting size of the vector and the need to represent a variable length structure in a fixed length format. Some degree of abstraction is therefore required. Our first simplification was that the user chosen tokens (e.g. identifiers or operators), should be largely ignored; this gives a big reduction to the size of the problem. We were therefore left with the problem of capturing the information in the parse tree of the software unit. We do this in three ways: firstly the frequency of keywords in the unit are accumulated, secondly the indentation pattern is represented and lastly the length of each line is recorded. The method is applicable to any source code language - it has even been suggested that intermediate code may be usefully examined.

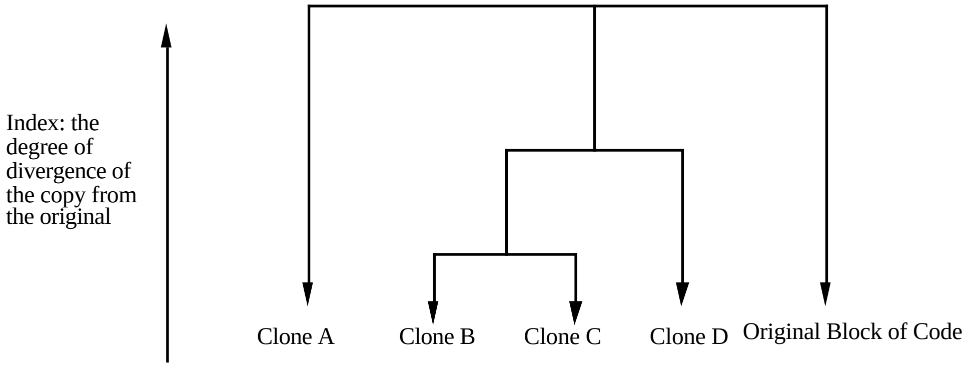

Index: the degree of divergence of the copy from the original
Clone A
Clone B
Clone C
Clone D
Original Block of Code

Figure 2: An indexed taxonomic tree for the clones of a single block of code. It can be seen that A and D were cloned and subsequently B and C were cloned from D.

As all the code is printed in a standard format (or can be easily filtered to a standard format), generated by the programming environment, the indentation pattern is isomorphic to the structure of the parse tree. The problem with representing it is that the number of lines in a unit of software, and therefore the number of indentations, is variable. To map this to a fixed length vector we first took the raw indentation values and viewed them as ordinates on a graph; we then sampled one hundred points from this graph, using linear extrapolation where necessary. This coding method is relatively stable against minor modifications to the source code, such as the addition or removal of a line. See Figure 3.

The line length gives some indication of the user defined tokens; it was coded in a similar way to the indentation pattern, again being mapped to one hundred points across the software unit.

Finally, each field in the vector was normalised, so that each field had roughly the same importance. Each keyword frequency was divided by the maximum frequency for that keyword averaged over a large set of source code units. Each indentation and line-length value was divided by a mean value calculated as before. That is the j’th component of the i’th vector is calculated as: $x'_{ij} = \frac{x_{ij}}{\underset{k}{mean}(x_{ik})}$

The overall structure of the vector is seen in Figure 4.

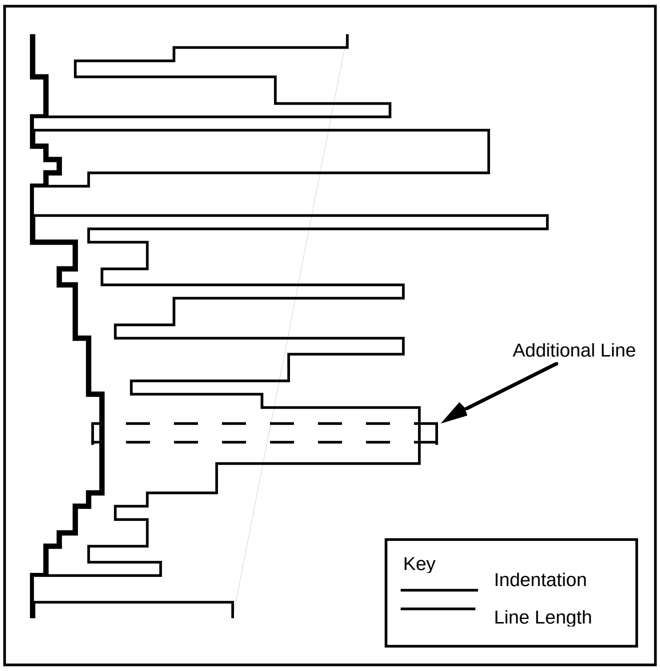


Figure 3: The stability of the sampled indentation pattern when lines are inserted

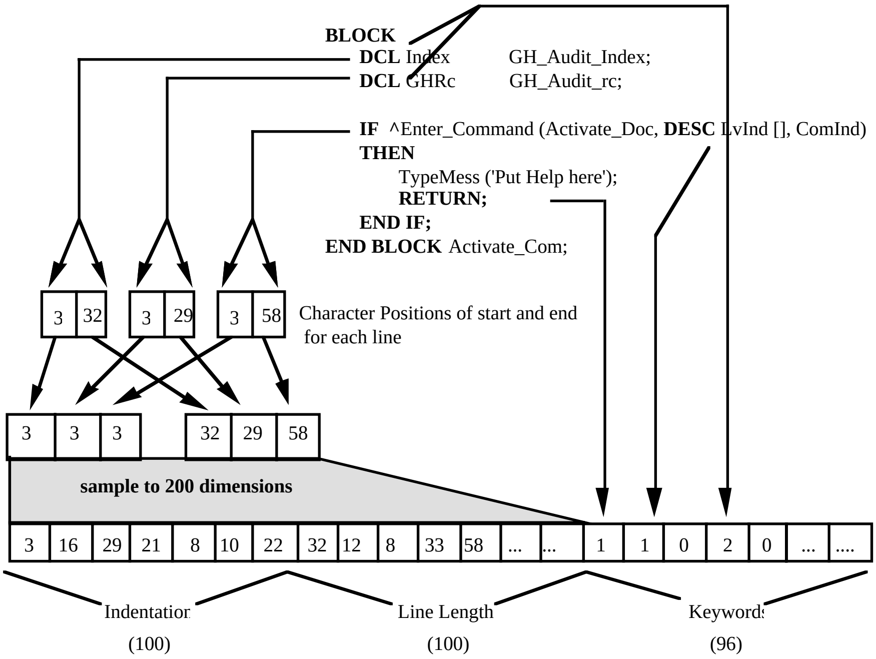


Figure 4: Illustrates how a feature vector is produced to represent a procedure.

4 The Current System

4.1 Introduction

Once the data has been represented as a set of fixed length feature vectors it was then possible to use this to train a neural net to identify clusters in the data. Currently we have used two different types of net, a SOM and subsequently a dynamic tree based net. The former has been incorporated in a full clone detecting tool. This section describes the SOM based system; the following section describes the DCL network.

4.2 Initial Requirements

The system in which the search for clones takes place is organised around specific products, such as a telecommunications switch. This decomposition of the complete system is not disjoint, but product managers will normally only be concerned with identifying clones in their particular product. Typically a product will contain in the order of 10,000 procedures. The initial requirement was therefore to produce a tool that allowed:

1. The selection of all the software from a specific product
2. The encoding of the procedures in this software as feature vectors
3. The training of a SOM based neural net using these vectors
4. The user to identify likely clones and to find clones of a particular procedure

4.3 The Self Organising Map

In a SOM the output units are arranged in a fixed topology, usually a two dimensional grid. In this work we use a grid, wrapped around at the boundaries. Winning units pull their neighbours in the grid with them, towards the input. The neighbourhood of a unit should initially be set to a value which is a large fraction of the output space, and decrease over time. A useful heuristic for setting the neighbourhood is that it should start at roughly half the output space and decrease to unity, to ensure global order.

The key parameter that must be identified when using a SOM is the size of the grid to use. This is dependant on the number of input vectors and the inherent clustering in the data. After extensive experimentation we found that a grid of 55 by 55 units produced a good classification across different product sets. Figure 1 shows how a typical data set is classified.

4.4 The System Architecture

The finished system can be viewed as consisting of two logical components, that support administrative tasks and user tasks. The administration task is to produce the classification data for a set of source code and the user task is to support the interrogation of the resulting classification in the search for clones. The whole process can be seen in Figure 5.

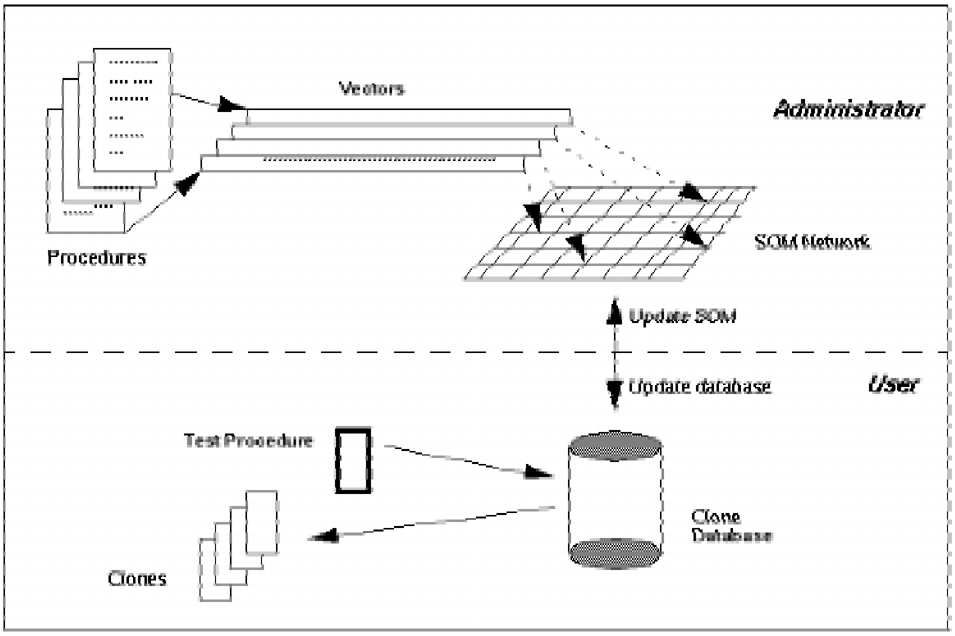


Figure 5: The clone detector tool process

4.4.1 The System Administrator

This main function of this part of the system is to produce a database of clone information. Initially a collection of source code procedures, which will be used to populate the database is down loaded and pre-processed to produce a set of feature vectors, as described in 3.2. This is a non-trivial task and involves a partial source code parser built using the UNIX tools byacc, flex and newyacc, to extract the structure of the code, and then a C program to put together the feature vectors. These vectors are then used as a training set for an encapsulated SOM, which was developed using Neuralware's neural network tool NeuralWorks Professional II. Due to the number of vectors in the training set the network requires several hours of training time.

Once the SOM is trained the training set is used in a test phase to associate a grid reference with all the source procedures. This information is stored in the clone database, and for these procedures no further reference is needed to the SOM. However, if a search is required for clones of an unseen set of procedures an additional test phase is required; when this occurs the clone database is updated with the new information. The tool currently runs on both a HP9000 7xx and 4xx series workstation.

4.4.2 The User View and the Graphical User Interface (GUI)

Once a database has been generated, the navigation and interrogation of the results produced by the neural network is controlled through a GUI. The GUI was developed using GNU C++ and Neuron Data's Open Interface.

The main window of the GUI is shown in Figure 6. There are five different areas to the main window. The first area contains the source procedure (top left). The second contains all the possible clones of that procedure (top right), and the three areas below display the actual code and where the differences between the two procedures occur. Once a potential clone has been identified, the user is then able to view the code not only of the source procedure, but also that of the potential clone. The two sections of code are displayed along side each other in the bottom half of the main window. The user is then able to view the code and see exactly how the sections differ and verify if that clone identified is actually a clone of the source procedure. In figure 6, the only difference between the source procedure and the clone which has been identified is that the procedure name and one of the identifiers has been changed. The cloniness value is shown as 100% because the differences are so minor as to make the vector representation of the two procedures identical.

4.4.3 Finding Cloned Procedures

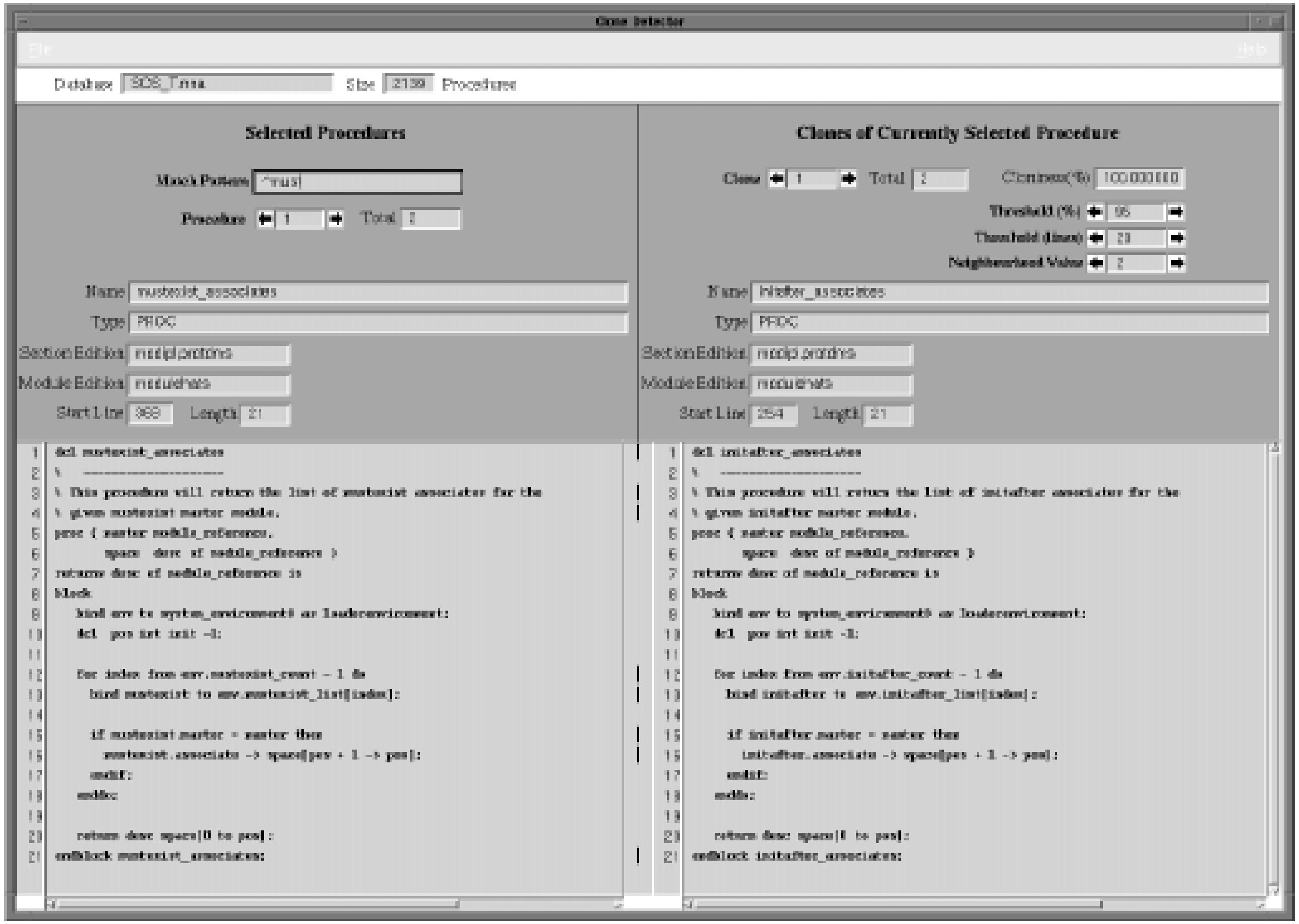
Once a database has been selected and loaded into the GUI the users must then identify a source procedure, this is done by using area 1 of the main window. The tool then automatically generates a list of possible clones. The criteria to which this list must conform to is identified by the attributes in the top right hand corner of the main window.

The attributes which determine the search criteria are contained in the top right hand side of the window. They are:

- Cloniness - A simple similarity value. It uses the cosine measure of similarity of the feature vectors.
- Threshold% - The minimum similarity that the reported clones have to match. This acts as a tolerance measure, a high value indicates that a high degree of accuracy is required.
- Threshold (lines) - The minimum number of lines which the potential clones must have.
- Neighbourhood Value - The size of the area of the neural network were clones might be found. A high proportion of all clones for a particular procedure will be within the immediately surrounding area.

Neighbourhood defines the search area, in the neural network, with the selected procedure belonging to the neural network class at the centre of the search area. Closely matching neural network classes are situated close together in the network. If the user wants to increase the search area to bring in more potential clones then the neighbourhood search size can be increased. Increasing the search area will introduce more potential clones, but these will have a lower cloniness value.

Rather than searching for clones for each procedure independently the tool allows for a list of all the clones in any particular database to be produced. However, as the size of the database grows the time to perform the operation grows exponentially.



Database SDS_Trma Size 2139 Procedures

Selected Procedures

Match Pattern ^must

Procedure 1 Total 2

Name mustexist_associates
Type PROC
Section Edition mcdlpl.protdnrs
Module Edition modulehats
Start Line 368 Length 21

```
dcl mustexist_associates
%   --------------------
% This procedure will return the list of mustexist associates for the
% given mustexist marker module.
proc ( master module_reference,
       space  desc of module_reference )
returns desc of module_reference is
block
   bind env to system_environment0 as loaderenvironment:
   dcl  pos int init -1:

   for index from env.mustexist_count - 1 do
     bind mustexist to env.mustexist_list[index]:

     if mustexist.marker = master then
       mustexist.associate -> space[pos + 1 -> pos]:
     endif:
   endfor:

   return desc space[0 to pos]:
endblock mustexist_associates:
```

Clones of Currently Selected Procedure

Clone 1 Total 2 Cloneness(%) 100.000000

Threshold (%) 95

Threshold (lines) 20

Neighbourhood Value 2

Name initafter_associates
Type PROC
Section Edition mcdlpl.protdnrs
Module Edition modulehats
Start Line 254 Length 21

```
dcl initafter_associates
%   --------------------
% This procedure will return the list of initafter associates for the
% given initafter marker module.
proc ( master module_reference,
       space  desc of module_reference )
returns desc of module_reference is
block
   bind env to system_environment0 as loaderenvironment:
   dcl  pos int init -1:

   for index from env.initafter_count - 1 do
     bind initafter to env.initafter_list[index]:

     if initafter.marker = master then
       initafter.associate -> space[pos + 1 -> pos]:
     endif:
   endfor:

   return desc space[0 to pos]:
endblock initafter_associates:
```

Figure 6: The user interface of the productised clone detector.

4.5 SOM Results

Results were produced using input vectors from 1775 procedures extracted from a file of approximately 5Mbytes of arbitrarily selected source code. A SOM was trained using these vectors. At the end of training the output from the SOM was further processed to populate a clone database. Experiments using various input vector representations were carried out, and these results showed that the Keyword/Indentation representation was sufficiently rich to classify procedures. Clones were also deliberately created and seeded into the source code. The detector was able to find almost all of these.

5 Dynamic Competitive Learning (DCL) Model

5.1 Introduction

The limitations of the SOM-based tool are the long training times and the fixed number of clone classes that are created. Using the same input vector representation we have implemented a prototype tree based dynamic neural network.

As described in section 3 the ideal classification structure for the type of data used here is a tree structure, rather than the flat structure induced by a SOM. As a simple example consider the tree in Figure 7.

Here the classification space has been divided into two clusters at the top level: one containing E only and the other, which is further subdivided, the other four vectors. The advantage of this structure is twofold:

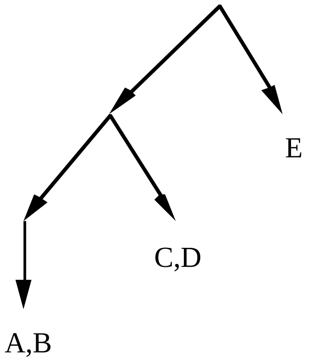


Figure 7: A simple tree structure representing the similarity of five vectors A…E.

- The depth in the tree of the nearest common ancestor of two vectors is a measure of their similarity. This is useful in the test phase - it is easy to specify the type of clone required, for example: look as far as second cousins.

- A search through such a data structure is significantly reduced in comparison to a search through a flat data structure. Specifically, to find an item in the tree takes $O(\log_b N)$, where N is the depth of item in the tree and b the average branching factor, whereas a search in a linear structure is $O(N)$. This has implications for both the training and test phases - both are reduced in time.

To produce such a tree structure we used a recently proposed unsupervised architecture, the Dynamic Competitive Learning model (Racz and Klotz, 1991). For an evaluation of the DCL network and other related models see (Butchart 1995b).

5.2 The DCL algorithm

The nodes in the DCL network are arranged in a tree structure with a dummy node as root. When an input is presented to the network it is passed to the first layer of the tree, where the standard competition takes place between nodes at this level. If the winner is not sufficiently close to the input, that is the distance is greater than a *quality* value, a new node is created at this level to classify the input. This process is analogous to the use of a vigilance parameter in ART. The quality value determines the radius of the hypersphere, that is the classification volume of a node; this value must therefore decrease as the tree grows deeper, so that lower levels provide a finer classification than higher levels.

If a new node is not required then the winner moves towards the input, mediated by a learning rate, as in standard competitive learning. The learning process is then recursively applied to the children of the winner, until a leaf node of the tree is reached.

At any stage of the learning process a winning node without children may procreate. This occurs if the relative frequency of wins of the node against its parent exceeds a *threshold* value. In order to prevent unbounded growth in the tree the threshold value is increased for lower levels in the tree.

More formally the training algorithm is:

```
Initialise learning rate, r, threshold, t , quality q and Tree to be the root node
Repeat until convergence criterion met
        Set x to the next input vector
        Select the child of Tree that is closest to x, w say
        If | w - x | < q then
                move w towards x, according to $w' = w + r(x - w)$
                Increment the win count of w
                If w has children then set Tree to w, increment t, decrement q and recurse
                else if the ratio of the win count of w to the win count of Tree > t
                        generate for w a child with weight vector = x
        Else
                Create a new child of Tree that has weight vector identical to x
End Repeat
```

The shape of the resulting tree is determined by the input vectors and the choice of learning parameters. The following factors must be specified:

- The quality value of the top level nodes
- The rate at which the quality factor decreases for successive layers, as a proportion of the previous value
- The threshold value of the top level nodes
- The rate at which the threshold value increases for successive layers, as a proportion of the previous value
- The schedule for the learning rate - which should decrease after each epoch to promote stability as a proportion of the previous value

The Quality Factor was arrived at with the aim of producing a reasonably large initial cluster. With this setting the initial cluster contains 20 nodes with 4 developing a sub classification. So the initial space is divided up into 16 outlier clusters and 4 larger clusters which are sub-divided.

The Quality Reduction rate was chosen so that the network developed the appropriate amount of decomposition and clustering. The very low initial value of the threshold allows the tree to rapidly deepen, but with an increment rate of 1.15, the threshold quickly becomes large: at ten ply the threshold is 0.12, so that a unit must account for 12% of its parents wins before it can gain a successor.

In the test phase the node closest to the input vector is found by a straightforward search of the tree:

- If x is the input set current-node to the root of the classification tree.
- Find the child of current-node that is closest to x. If this node is a leaf then return it, else repeat with current-node set to this child.

5.3 Training

With these parameters the network converged rapidly; with 10,257 296–ary vectors only two or three epochs were required for reasonable convergence. The training time was less than forty minutes on an HP 712/80.

5.4 Results

A typical classification tree produced by the DCL is described below:

No of nodes	1500
Maximum Ply	19
Average Branching Factor	2.84
Widest point	298 nodes, at depth 5

The leaf nodes of the tree classified, on average, about 20 input vectors. A fraction of the tree is shown in Figure 8.

The quality of the classification was similar to the SOM, but this net has some advantages. The training and recall time is significantly better than the SOM, and the tree structure aids in the search for clones more distant than those in the immediate neighbourhood, as discussed in 5.1.

6 Evaluation

6.1 Productising the prototype

The major challenge that we overcame in this project was moving from the original small scale prototype to a full product quality tool. The task here was not simply scaling up the prototype, but more of improving the overall quality of the system. It soon became apparent that all the components of the system would have to be rebuilt, and it is worth noting that in the final system the neural net component is a relatively small part.

6.2 Use

The tool has been extensively trialed by a number of product groups throughout BNR world-wide. The response was favourable with some groups finding surprising clones in their source code.

One product group were quite adamant that no clones existed within their code. However, this proved to be incorrect: in fact over thirty clones were found in their database of 2139 procedures. The source displayed in Figure 6 is one of several sets of clones that were discovered.

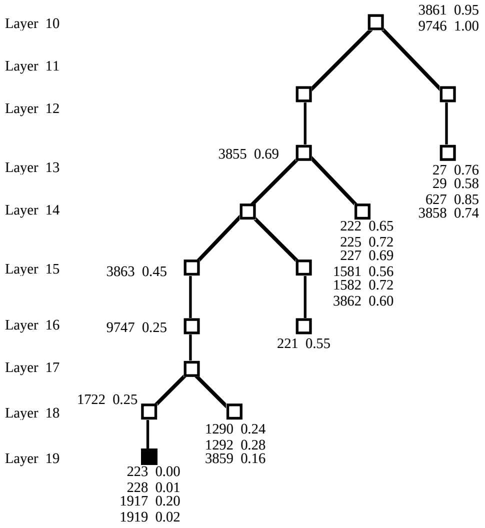


Figure 8: Classification produced by the DCL net, shows the input (left) and the Euclidean distance (right) from a given test input. For the test input 223 procedure 228 is the closest.

6.3 Future Developments

As identified above, more work must be done in selecting a representative set of data for a complete switch (which may contain tens of millions of lines of code and hundreds of thousands of procedures).

In order to improve performance, in terms of reducing training times and network complexity, experiments are being carried out using the Dynamic Competitive Learning network described in section 5. The prototype has already demonstrated the potential benefits of using this approach, but before productisation the problems of representing a large tree based structure with a GUI to the user must be solved.

7 Conclusions

Cloned software is prevalent in large software systems and this project has shown how it can be managed without complete re-engineering of the code. It has also been shown how neural computation can be used and integrated into fully functional tools. In fact one of appeals of using neural nets is their relative ease of use when compared with other sophisticated techniques.

Over the period of this work, however, it has become apparent that the task of moving from a successful, neural net based prototype to a full system should not be underestimated. All neural network applications depend heavily on the appropriate pre-processing of the input data and post-processing of the output data. A major part of our work has been concerned with the pre-processor, the user interface and the overall quality of the system. The lessons learnt from the production of this clone detector software tool are being incorporated into a neural network application development method (Field et al, 1995).

Acknowledgements

We would like to acknowledge the financial assistance of the Department of Trade and Industry of the UK Government, who have partly funded the placement of Paul Barson and Simon Field on a Teaching Company Scheme at BNR Europe Limited, through the Teaching Company Directorate organisation, under grant no. TCS–1326.

References

Barson, P. , Davey, N. , Field, S. , Frank, R. and Tansley, D.S.W. (1995). "Dynamic Competitive Learning Applied to the Clone Detection Problem", in *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*.

Butchart, K. , Davey, N., and Adams, R. G. (1995a). "A Comparative Study of Three Neural Networks that use Soft Competition ", in *Proceedings of International Workshop on the Applications of Neural Networks*.

Butchart, K. , Davey, N., and Adams, R. G. (1995b) "A Comparative Study of two Self Organising and Structurally Adaptive Dynamic Neural Tree Networks", in *Proceedings of Applied Decision Technologies*

Carpenter, G. & Grossberg, S. (1988), "The Art of Adaptive Pattern Recognition by a Self-Organising Neural Network", *IEEE Computer* 21(3), 77-88.

Carter, S. , Frank, R.J. and Tansley, D.S.W. (1993), "Clone Detection in Telecommunication Software Systems: A Neural Net Approach", in *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications* pp. 273-280.

Field, S. , Davey, N. and Frank, R. (1995), "Using Neural Networks to Analyse Software Complexity", in *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications* 1995.

Kohonen, T. (1990) "The Self-Organising Map", P*roceedings of the IEEE*, Vol. 78, No 9.

Möller, K.H. & Paulish, D.J. (1993), *Software Metrics*, London: Chapman & Hall.

Racz, J. & Klotz, T. (1991), "The Dynamic Competitive Learning Method", *Computers in Industry*, **17**, 155-158.

Rannal, R., Toulouse, G. & Virasoro, M.A. (1986), "Ultrametrics for Physicists", *rev. mod. phy.*, **58**, 765.

Schendler, B.R. (1989), "How to Break the Software Logjam", *Fortune*, pp. 72-76, 25 September 1989.

Wayt Gibbs, W. (1994), "Software's Chronic Crisis", *Scientific American* pp 72-81, September 1994.

Trademarks

NeuralWorks Professional II is a trademark of Neuralware inc.
HP is a trademark of Hewlett Packard.
Open Interface is a trademark of Neuron Data.