USING NEURAL NETWORKS IN SOFTWARE MODULE IDENTIFICATION

Salwa K. Abd-El-Hafiz

Engineering Mathematics Department, Faculty of Engineering,
Cairo University - Giza, Egypt.

ABSTRACT

This paper presents a general approach for the identification of modules in procedural programs. The approach is based on neural architectures that perform an unsupervised learning of clusters. We describe two of such neural architectures, explain how to use them in identifying modules in software systems and briefly describe a prototype tool, which implements the clustering algorithms. With the aid of several examples, we explain how our approach can identify abstract data types as well as groups of routines which reference a common set of data. The clustering results are compared to the results of many other modularization techniques. Finally, two case studies were performed on existing programs to evaluate the modularization approach. Results concerning the analyzed programs and their generated clusters are discussed.

Keywords: Clustering, Modules, Objects, Abstract Data Types, Neural Networks.

INTRODUCTION

The identification of modules within existing procedural programs can facilitate many maintenance activities. By extracting reusable components from existing software systems, the population of a reuse repository can be cost effective [1, 2]. That is why the extraction of reusable abstract data types has been the focus of many research activities [2-5]. In addition, the automatic identification of modules assists in understanding the relationships among the components of a software system. It, thus, facilitates the recognition and comprehension of the abstractions existing in a given system without getting distracted by implementation details [3, 6 and 7]. Furthermore, modularization enables the re-engineering of procedural programs into functionally comparable object-oriented systems [8-10]. This helps in the modernization of legacy systems that are difficult to maintain due to the lack of a modular style. The re-engineering of such systems into functionally equivalent object-oriented ones makes them much easier to maintain.

Some of the module identification approaches are based on defining a model of the subject system as a graph on which notable sub-graphs and/or patterns are identified [2,3,5,10-12]. Other approaches apply mathematical concept analysis to the modularization problem [9,13-15]. Concept analysis is a branch of lattice theory that can be used to identify similarities among a set of objects based on their attributes [9]. Cluster analysis has also been used to identify modules by, implicitly or explicitly, using a similarity measure among pairs of functions [3, 16-19].

This paper presents a modularization approach that is based on cluster analysis. More specifically, we use clustering neural networks to identify modules in procedural programs. The first section describes two neural architectures which perform an unsupervised learning of clusters. The next section demonstrates how these clustering

neural networks can be used to identify modules in software systems. In that section, we also give the modularization results of many examples and compares them with related approaches. Then we briefly describe an implemented prototype tool and evaluates our approach by applying it on two existing procedural programs. Finally, we summarize the strengths and limitations of the presented approach and gives future research directions.

CLUSTERING NEURAL NETWORKS

Clustering is understood to be the grouping of similar objects and the separation of dissimilar ones. Clustering neural networks learn clusters in the input data without the need of being taught. That is, they perform unsupervised learning of clusters based on data correlations (i.e., similarity measures). The same input pattern is presented to the network several times, and a pattern may move from one cluster to another until the network stabilizes. In this paper, we consider Adaptive Resonance Theory (ART) networks and Kohonen's Self-Organizing Maps (SOM) [20-22].

In order to demonstrate how the two neural architectures perform an unsupervised learning of clusters, we use the data shown in Table 1. This data is for a group of nine animals, each described by its own set of attributes [23]. The group breaks down naturally into three clusters: mammals, reptiles, and birds.

Adaptive Resonance Theory

Adaptive Resonance Theory (ART) models are neural networks that perform clustering. They allow the number of clusters to vary with problem size [21]. The main feature of ART models is that they permit the user to control the degree of similarity between members of the same cluster by means of a user-defined constant called the vigilance parameter, *P*. The ART1 network only accepts binary (0/1) input vectors. As shown in Figure 1, it uses two layers of neurons (or nodes) with feedforward connections (from input to output nodes) as well as feedback connections (from output to input nodes). The input layer contains as many nodes as the size of the input vector. The output layer has a variable number of nodes representing the number of clusters

Table 1 Animals data for unsupervised learning of clusters.

	has hair	has scales	has feathers	Flies	lays eggs
1. Dog	1	0	0	0	0
2. Cat	1	0	0	0	0
3. Bat	1	0	0	1	0
4. Canary	0	0	1	1	1
5. Robin	0	0	1	1	1
6. Ostrich	0	0	1	1	1
7. Snake	0	1	0	0	1
8. Lizard	0	1	0	0	1
9. Alligator	0	1	0	0	1

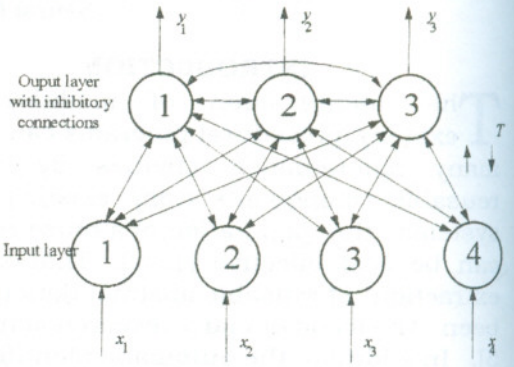

Figure 1 An ART1 network with 4 input and 3 output neurons.

When a new input vector x is presented to the network, it is communicated to the output layer via upward connections carrying bottom-up weights, B. A bottom-to-top processing stage uses the B weights matrix to compute the matching scores, y. These scores reflect the degree of similarity of the present input to the previously encoded clusters. The candidate cluster, j, is the one that best matches the input vector. The similarity

measure used in this network is the hamming distance which is equal to the number of different bit positions between two binary vectors of the same length. Strictly speaking, the hamming distance is proportional to the dissimilarity of vectors [22].

The top-down part of the network performs a vigilance test using the vigilance parameter, P, with $0 < P < 1$. P controls the degree of similarity between the candidate cluster, already encoded in the T weights matrix, and the current input vector. That is, the current input should be in 'resonance' with the encoded candidate cluster. Large P values indicate a more strict similarity requirement than small P values. If the vigilance test succeeds, the input is associated with the winning cluster and the weight matrices are updated accordingly. If the test fails, the node j is deactivated and the search continues until either an appropriate cluster is found or a new one is created. The network is 'adaptive' because it allows the creation of new clusters when deemed necessary.

Figure 2 shows the results of applying the ART1 clustering algorithm to the animals data of Table 1. The input vectors correspond to the rows of Table 1. Starting with small P values ($0.1 \leq P < 0.4$) resulted in the successful identification of the mammals cluster. However, the network could not differentiate between reptiles and birds. Increasing the P values ($0.4 \leq P < 0.6$) resulted in identifying three clusters; mammals, reptiles, and birds. Figure 2 highlights these three clusters by using **bold** rectangles. High P values ($0.6 \leq P \leq 0.9$) further decomposed the mammals cluster based on whether they fly or not. Theoretically, there is an infinite number of possibilities for the P values and, correspondingly, for the clustering results. In practice, however, the possible clustering alternatives are limited because a range of P values can give the same clustering results.

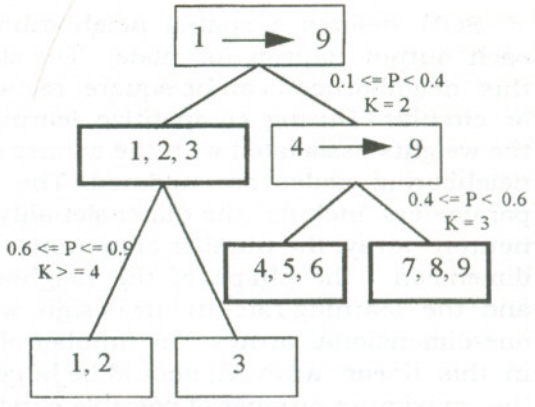

Figure 2 Clustering results for the animals example.

Self-Organizing Maps

Kohonen's Self-Organizing Maps (SOM) have the property of topology preservation. In a topology-preserving mapping, nearby input patterns should activate nearby output units on the map [20-22]. Figure 3 shows the basic network architecture of Kohonen's SOM. It consists of a two-dimensional array of connected neurons. Each neuron is also connected to all n input nodes. Let W_i denote the n-dimensional vector associated with the neuron at location i of the two dimensional array. Each neuron computes the Euclidean distance between the input vector x and the stored weight vector W_i, which is interpreted as a cluster centroid [20].

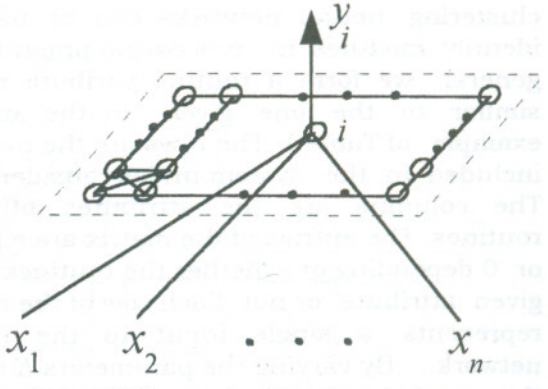

Figure 3 An SOM network with a rectangular array of neurons.

SOM defines a spatial neighborhood for each output neuron (or node). The shape of this neighborhood can be square, rectangular or circular. During competitive learning, all the weights associated with the winner and its neighboring nodes are updated. The design parameters include the dimensionality of the neuron array, the number of neurons in each dimension, the shape of the neighborhood and the learning rate. In our design, we used one-dimensional arrays. The number of nodes in this linear array, K, should be larger than the maximum number of possible clusters for the problem but smaller than the number of input vectors [21]. The shape of the neighborhood we use is circular and the learning rate is exponentially decaying.

The results of applying the SOM clustering algorithm to the animals data of Table 1 is also shown in Figure 2. In this small example, varying K yields the same clustering results as those of the ART1 network. It should be noted that K represents an upper limit on the resulting number of clusters. Making K greater than or equal to four in our example yields the same results because at most four clusters could be identified in the data of Table 1.

MODULE IDENTIFICATION VIA CLUSTERING NEURAL NETWORKS

In this section, we demonstrate how clustering neural networks can be used to identify modules in procedural programs. In general, we form a routine-attribute matrix similar to the one given in the animals example of Table 1. The rows are the routines included in the system under consideration. The columns are the attributes of these routines. The entries of the matrix are either 1 or 0 depending on whether the routines has a given attribute or not. Each row of the matrix represents a single input to the neural network.. By varying the parameters K and P, the neural networks output gives multiple clustering possibilities.

Related literature adopted several strategies for picking up the attributes. Attributes used before include: usage of common global variables [2,3,5, 10,12-15], dataflow information [17], usage of user-defined data types [4], in general, and usage of record (structure) data types, in specific [9, 10]. Similar to Siff and Reps approach [9], our approach is very flexible and general when it comes to the choice of the attributes. Any set of attributes, that may be useful in some instances, can be used in our approach. In our examples and case studies, using the following attributes, either separately or jointly, yielded good modularization results:

- Usage of global variables. An attribute might be of the form 'uses global variable x'.
- Usage of record (structure) and enumeration data types. An attribute might be of the form 'uses fields of struct stack', 'has argument of type struct stack *', or 'return type is struct stack *'.
- Disjunction of attributes related to similar user-defined types or similar global variables. For instance, if T_1 and T_2 are two similar data types, the disjunction 'uses fields of T_1 or uses fields of T_2' can improve the modularization results [9].
- Usage of data files and/or usage of read/write statements. In some cases, such attributes identify the modules which interact with the user.

Since the neural networks can generate different clustering results at different parameter values, we form a clustering tree, similar to those shown in Figures 2 and 5, to facilitate the visualization and analysis of clustering results. In this tree, the root node represents all routines in the program. Whenever the neural network generates partitions of an existing tree node, we create the corresponding sub-nodes which represent the resulting partitions.

To further explain our clustering techniques and to facilitate the comparison with related modularization techniques, we

use several examples adapted from Canfora *et. al.* [3], and Siff and Reps [9]. Despite the fact that our techniques apply to any procedural programming language, the examples in this section are in C.

Figure 4 shows a specific C implementation of stacks and queues [9]. Queues are represented by two stack; one for the front and one for the back. Information is shifted from the front stack to the back stack when the back stack is empty. The queue functions make indirect use of the stack fields by calling the stack functions. We would like to identify the two modules representing the two given abstract data types. Using the functions of Figure 4 and the attributes of Table 2, we formed the routine-attribute matrix of Figure 5. This matrix represents the input to the two clustering neural networks under consideration. We varied P between 0.1 and 0.9 with a step of 0.1 and gave K values that are greater than or equal to 2. ART1 and SOM gave the same clustering tree, which is depicted in Figure 5. As shown by the two **bold** rectangles, the two abstract data types are correctly identified.

```
struct stack { int *base, *sp, size; };
struct queue {struct stack *front, *back; };

/* 1 */  struct stack *initStack(int sz)          {/* uses fields of struct stack */}
/* 2 */  struct queue *initQ( )                   {/* uses fields of struct queue */}
/* 3 */  int isEmptyStack(struct stack* s)        {/* uses fields of struct stack */}
/* 4 */  int isEmptyQ(struct queue *q )           {/* uses fields of struct queue */}
/* 5 */  void push(struct stack* s, int i)        {/* uses fields of struct stack */}
/* 6 */  void enq(struct queue *q, int i)         {/* uses fields of struct queue */}
/* 7 */  int pop(struct stack* s)                 {/* uses fields of struct stack */}
/* 8 */  int deq(struct queue *q)                 {/* uses fields of struct queue */}
```

Figure 4 A sample C-like code for a stack and a queue (adapted from [9]).

Table 2 Attributes for the stack-queue example.

#	Attribute
A_1	argument or return type is struct stack*
A_2	argument or return type is struct queue*
A_3	uses fields of struct stack
A_4	uses fields of struct queue

	A_1	A_2	A_3	A_4
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0
4	0	1	0	1
5	1	0	1	0
6	0	1	0	1
7	1	0	1	0
8	0	1	0	1

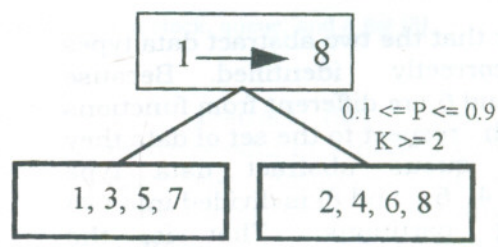

Figure 5 The routine-attribute matrix and its clustering results for the stack-queue example

Module identification in the presence of undesired links

The results generated by clustering neural networks, in the examples considered thus far, are similar to the results produced by many other techniques in literature (see for example [9 and 10]). To demonstrate the full power of clustering neural networks, we now consider their application to real-life systems. In such systems, there can be some routines which cause undesirable clustering of functions. Canfora *et al.* [3] describe two different types of undesired links: coincidental links and spurious links. A coincidental link results from a routine that actually includes implementations of several routines, each logically belonging to a different module. Spurious links are created by routines that access the supporting data structures of more than one module in order to implement system specific operations. Many modularization approaches do not yield good results when applied to examples that exhibit undesired links (see for example References 5, 10, 12 and 24).

As an example of spurious links, Siff and Reps [9] consider the following modification of the stack-queue example given in Figure 4.

```
/* 4 */   int isEmptyQ(struct queue *q)    {/* uses fields of struct
                                            stack and struct queue
                                            */}
/* 6 */   void enq(struct queue *q, int i) {/* uses fields of struct
                                            stack and struct queue
                                            */}
```

Although such a modification may be more efficient, it causes some queue routines to access the supporting data structure of the stack routines. Figure 6 shows the routine-attribute matrix and the clustering tree after performing this modification.

	A_1	A_2	A_3	A_4
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0
4	0	1	1	1
5	1	0	1	0
6	0	1	1	1
7	1	0	1	0
8	0	1	0	1

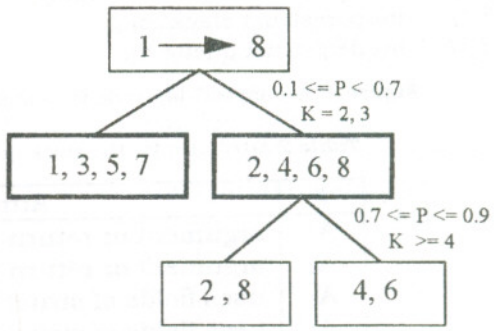

Figure 6 The routine-attribute matrix and its clustering results after modifying the stack-queue example

It is clear that the two abstract data types are still correctly identified. Because functions 4 and 6 are different from functions 2 and 8, with respect to the set of data they access, the queue abstract data type (functions 2, 4, 6, and 8) is divided into two corresponding partitions. That is, the clustering technique provides additional information about similarities among the functions of a selected module. Compared to the concept analysis technique presented by Siff and Reps [9], we do not have to add a complementary attribute of the form 'does not use fields of struct queue' to correctly identify the two abstract data types.

In order to discuss the effect of both spurious and coincidental links, Canfora *et al.* [3] use the example of Figure 7. This example gives a sample C-like code which uses a stack, a queue, and a list. The function global-init (function # 20) is an example of a coincidental connection, while functions 14-

19 exemplify spurious connections. In this example, we use six attributes corresponding to the six global variables defined in the code. Each attribute has the form: 'uses global variable *x*'. For more details on this example and on its routine-attribute matrix, refer to [3].

```
ELEM_T  stack_struct[MAXDIM];
int     stack_point;

ELEM_T  queue_struct[MAXDIM];
int     queue_head,
        queue_tail,
        queue_num_elem;

struct  list_struct {ELEM_T node_content;
                     struct list_struct * next_node;
                    } list;
main ( )
{/* this program exploits a stack, a queue, and a list of items of type ELEL_T */}

 /* 1 */    void stack_push ( el )          {/* uses stack_point and stack_struct */}
 /* 2 */    ELEM_T stack_pop ( )            {/* uses stack_point and stack_struct */}
 /* 3 */    ELEM_T stack_top ( )            {/* uses stack_point and stack_struct */}
 /* 4 */    BOOL stack_empty ( )            {/* uses stack_point */}
 /* 5 */    BOOL stack_full ( )             {/* uses stack_point */}

 /* 6 */    void queue_insert ( el )        {/* uses queue_struct, queue_head and queue_num_elem */}
 /* 7 */    ELEM_T queue_extract ( )        {/* uses queue_struct, queue_tail and queue_num_elem */}
 /* 8 */    BOOL queue_empty ( )            {/* uses queue_num_elem */}
 /* 9 */    BOOL queue_full ( )             {/* uses queue_num_elem */}

 /* 10 */   void list_add ( el )            {/* uses list */}
 /* 11 */   void list_elim ( el )           {/* uses list */}
 /* 12 */   BOOL list_is_in ( el )          {/* uses list */}
 /* 13 */   BOOL list_empty ( )             {/* uses list */}

 /* 14 */   void stack_to_list ( )          {/* uses stack_point, stack_struct and list */}
 /* 15 */   void stack_to_queue ( )         {/* uses stack_point, stack_struct, queue_struct, queue_head and
                                             queue_num_elem */}
 /* 16 */   void queue_to_stack ( )         {/* uses stack_point, stack_struct, queue_struct, queue_tail and
                                             queue_num_elem */}
 /* 17 */   void queue_to_list ( )          {/* uses queue_struct, queue_tail, queue_num_elem and list */}
 /* 18 */   void list_to_stack ( )          {/* uses stack_point, stack_struct and list */}
 /* 19 */   void list_to_queue ( )          {/* uses queue_struct, queue_head, queue_num_elem and list */}
 /* 20 */   void global_init ( )            {/* uses stack_point, queue_head, queue_tail, queue_num_elelm and
                                             list */}
```

Figure 7 A sample C-like code for a stack, queue, and a list [3].

The results of applying ART1 and SOM are shown in Figures 8 and 9, respectively. We varied *P* between 0.1 and 0.9 with a step of 0.1 and gave *K* values that are greater than or equal to 2. Only *P* and *K* values which trigger a partitioning of an existing cluster are shown on these figures. ART1 succeeds in identifying the list (functions 10-13) and stack (functions 1-5) abstract data types. However, it is unsuccessful in separating the queue abstract data type (functions 6-9). On the other hand, SOM successfully identify all the three abstract data types. That is, SOM give results that are comparable to those of Canfora *et al.* [3] and better than those of [5, 12]. Additionally, SOM provide the information

that functions 14-20 can be grouped into three clusters: (15,16), (14, 18), and (17, 19, 20). As pointed out by Canfora *et al.* [3], program slicing [25] can overcome the coincidental connection introduced by routine number 20.

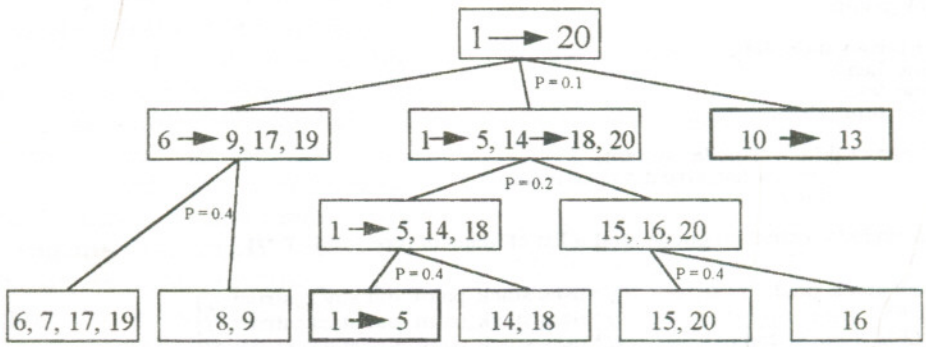

Figure 8 ART1 clustering results for the stack-queue-list example

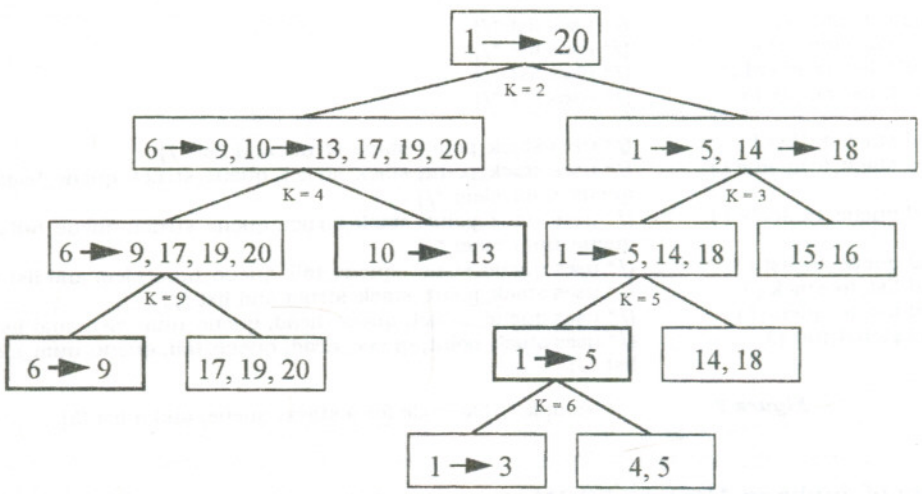

Figure 9 SOM clustering results for the stack-queue-list example.

IMPLEMENTATION AND EVALUATION

A prototype tool, which supports the clustering approach presented in this paper, has been implemented. The tool accepts as input the routine-attribute matrix. The routine-attribute matrix is constructed using simple static analysis of the code. The user decides which neural network model to use and its corresponding parameter. The clustering results are provided using a simple text-based interface. Currently, the tool is not provided with an advanced user interface. A graphical display of the clustering results, as shown in this paper, would certainly be very advantageous.

To evaluate our approach, we used the prototype tool to identify modules in two existing procedural programs. In the following two subsections, we present and discuss the modularization results of these two programs. Despite the fact that the examples presented so far focus on the identification of abstract data types, we demonstrate that our approach is also appropriate for the identification of other groups of routines which reference a common set of data (e.g., object instances) [10].

First Case Study: The Counting Program

This program performs different counts for C source files [26]. It provides the number of commentary source lines, the number of non-commentary source lines and comment-to-code ratio for C source files. These counts are reported for each function, for lines external to functions, and for the source file as a whole.

The program consists of 800 lines of C code. It has 17 functions, which are shown in Table 3. The designer of the program divided the program into 7 modules. In Table 3, each of these modules is enclosed between two dashed lines. The attributes we use for this program are given in Table 4. Because there is a small number of global and data type definitions in this program, we use a combination of all possible attribute categories. The attributes include the only structure defined in the program (count-struct). The three defined enumeration types are also included. We consider a disjunction of the two similar enumeration types, char_class and token_type. In addition, usage of two global definitions, data files, and read/write statements is taken into account

Table 3 Functions for the counting program

#	Function	#	Function
1	main	10	report_metrics
2	check_options	11	create_node
3	clean_command_line	12	destroy_node
4	get_parameters	13	is_empty_list
5	count_lines	14	create_list
6	start_tokenizer	15	append_element
7	get_token	16	delete_element
8	find_function_name	17	error
9	classify_line		

Table 4 Attributes for the counting program

#	Attribute
A_1	argument /return type is struct count_struct *.
A_2	uses fields of struct count_struct.
A_3	argument/return type is token_type or char_class.
A_4	uses elements of token_type or char_class.
A_5	argument /return type is error_type.
A_6	uses elements of error_type.
A_7	uses max_line.
A_8	uses max_ident.
A_9	uses a file data type.
A_{10}	uses read/write statements.

Because ART1 and SOM gave similar clustering results for this program, we only show the results of ART1 in Figure 10. The designers view of the program modularization is correctly identified for 5 modules. These 5 modules are drawn in **bold** rectangles. The first two designer-defined modules were joined together in one module (functions 1-4). The reason for joining these four functions together is that they possess non of the considered set of attributes. That is, they represent a collection of a driver and miscellaneous service routines. Functions 6-9

are the ones that parse lines of code and classify them. Functions 11-16 implement an abstract data type for lists of line counts. Functions 13, 14 are more similar to each other than the rest of the abstract data type functions because they do not use fields of the count-struct structure. Functions 5 and 10 are considered similar to this abstract data type because they have argument /return types of struct count_struct *. Function 17 generates the error messages.

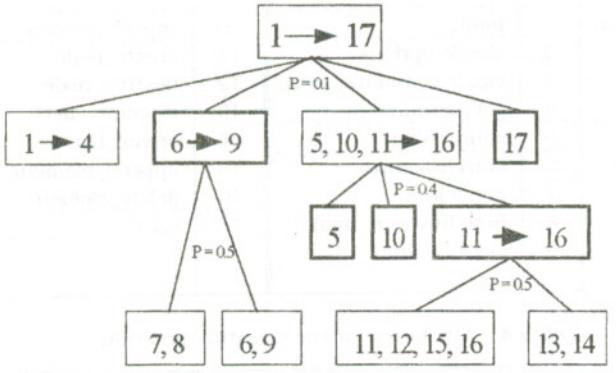

Figure 10 ART1 clustering results for the counting program.

Second Case Study: The Scheduling Program

This program schedules a set of courses offered by a computer science department [27]. It takes two input files. File 1 contains information about the courses in the department catalog, rooms in the building, and valid lecture times. The second file contains, for each course offered in a given semester, the course number, its enrollment and a set of time preferences given by the instructor of the course. The goal is to schedule these courses in the rooms such that a set of constraints given in a requirements document is satisfied.

The program consists of 1450 lines of Pascal code. It has 39 procedures/functions that are shown in Table 5. Table 6 shows some of the 23 attributes we used for this program. We used all the user-defined record data types. We only considered the usage of global variables of the array and file types (10 out of 21) because there are too many global variables in the program. Since read/write statements are used throughout the whole program, they are not considered in the attributes list. The program is divided by the designer into the four modules that are enclosed between dashed lines in Table 5. In addition to the driver module, there are modules to validate the input, perform the scheduling, and print the output. As shown in Tables 5 and 6, the program has a highly nested structure and uses global variables excessively. It also does not define any abstract data types. We would like to identify groups of procedures/functions (modules) which reference a common set of data.

Table 5 Procedures and functions for the scheduling program

#	Procedure/Function	#	Procedure/Function	#	Procedure/Function
1	main	14	chk_fmt_course_no	27	get_room_index
2	get_validated_input	15	chk_dup	28	prep_template
3	lengthof	16	Validate_lec_times	29	sched_pg_pref
4	get_next_line	17	chk_fmt_time_slot	30	sched_ug_pref
5	get_token	18	chk_dup	31	initialize_pg_reserve
6	string_to_int	19	validate_file_2	32	safe_allotment
7	validate_file_1	20	get_course_index	33	update_pg_reserve
8	validate_classrooms	21	get_pref_valid	34	sched_pg_no_pref
9	chk_fmt_rm_no	22	form_course_rec	35	sched_ug_no_pref
10	chk_range_cap	23	form_pref_list	36	print_output
11	sort_rooms	24	Duplicate_course	37	print_time_table
12	chk_dup	25	separate_courses	38	print_explanation
13	validate_dept_courses	26	Schedule	39	print_conflict

Table 6 Some attributes for the scheduling program.

#	Attribute
A_1	uses fields of record p_node_i.
A_2	argument /return type is ^p_node_i.
⋮	⋮
A_{12}	uses fields of record r_node.
A_{13}	argument /return type is ^r_node.
A_{14}	uses file1
A_{15}	uses file2
A_{16}	uses classroom_db
⋮	⋮
A_{23}	uses expl_list

The clustering results using ART1 and SOM are shown in Figures 11 and 12, respectively. Because of the large number of routines, we only form the clustering trees at three P values (0.1, 0.4, and 0.7) and three K values (7, 14, and 21). The SOM architecture gave slightly better results than the ART1 architecture. Thus, we focus on the description of the SOM results and only point out the deficiencies of the ART1 results. In describing the SOM results, the author's view of one correct way to decompose the program into modules is given. However, there can be several other modularization views based on the required level of granularity.

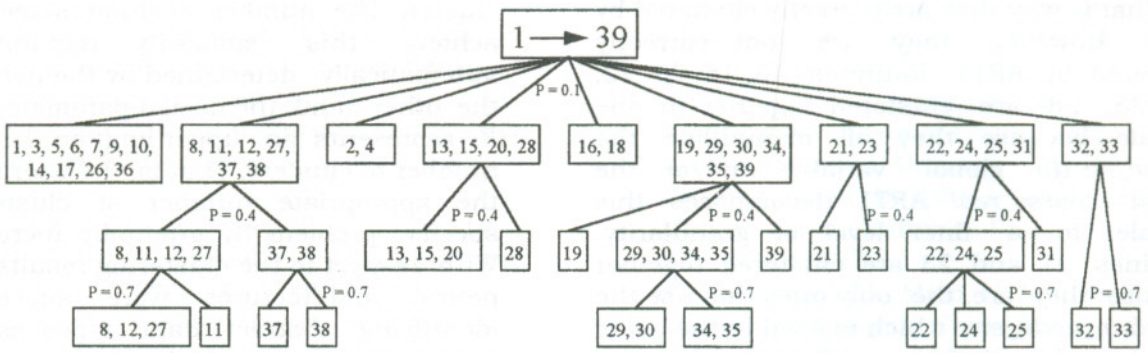

Figure 11 ART1 clustering results for the scheduling program.

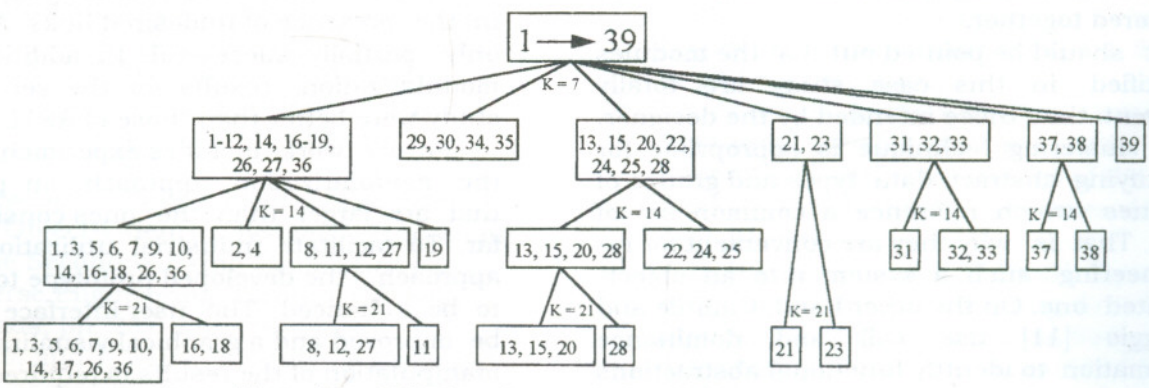

Figure 12 SOM clustering results for the scheduling program.

Routines 1, 3, 5, 6, 7, 9, 10, 14, 17, 26, and 36 are included in a single module because they do not possess any of the considered attributes. Routines 16 and 18 are clustered together because they are related to the validation of lecture times. Routines 2 and 4 are grouped together because they are the only ones that use the input data files. Routines 8, 11, 12, and 27 are clustered together because they manipulate the *classroom_db* global variable and/or the record *room_rec*. Routines 29, 30, 34, and 35 are included in single module because they represent the four main scheduling functions. Routines 31-33 represent the components of one main scheduling routine, routine number 30. That is why they are correctly clustered by SOM. However, they are not correctly clustered by ART1. Routines 13, 15, 20, 22, 24, 25, 28 are clustered together in one module because they all manipulate the *course_no_db* global variable and/or the record *course_rec*. ART1 decomposes this module to a finer level of granularity. Routines 21 and 23 are clustered together because they are the only ones that use the record *p_node_str*, which is used in the list of actual student preferences. Because the print routines (37-39) access a lot of global variables and record types, they are not correctly clustered by both neural architectures. Only routines 37 and 38 are clustered together.

It should be pointed out that the modules identified in this case study are totally different than those intended by the designer. Our clustering technique is appropriate for identifying abstract data types and groups of routines which reference a common set of data. That is why they are convenient for re-engineering such a system into an object-oriented one. On the other hand, Cimitile and Visaggio [11] use call and dominance information to identify functional abstractions and to introduce hierarchical nesting to the resulting modules. In this case study, such an approach would offer another modularization alternative which is close to the one followed by the system designer.

CONCLUSIONS

We have presented an approach for identifying modules in procedural programs. This approach is based on clustering neural networks. It is very flexible and general when it comes to the choice of the attributes on which the modularization is based. By controlling the design parameters of the two considered neural architectures, we obtain multiple clustering possibilities. The design parameter of ART1, P, controls the degree of similarity between elements of the same cluster. The number of clusters necessary to achieve this similarity requirement is automatically determined by the network. On the other hand, the design parameter of SOM, K, represents an upper limit on the required number of clusters. That is, the user identifies the appropriate number of clusters for a specific problem by gradually increasing K. With respect to the clustering results, the two neural architectures were successful in identifying abstract data types as well as groups of routines which reference a common set of data. However, the examples and case studies showed that the SOM architecture is slightly better than the ART1 architecture. While SOM succeeded in identifying modules in the presence of undesired links, ART1 was only partially successful. In addition, SOM modularization results for the second case study were better than those of ART1.

Future work includes experimenting with the modularization approach on programs that are larger than the ones considered so far. To facilitate industrial utilization of our approach, the developed prototype tool needs to be enhanced. The user interface needs to be improved and a graphical visualization and manipulation of the results is required.

REFERENCES

1. S. K. Abd-El-Hafiz, V. R. Basili, G. and Caldiera, "Towards Automated Support for Extraction of Reusable Components", Proceedings of the Conference on Software Maintenance, Sorrento, Italy, pp. 212-219, (1991).
2. G. Canfora, A.., Cimitile, M. and Munro, "An Improved Algorithm for Identifying Objects in Code", Software Practice and Experience, Vol. 26, No. 1, pp. 25-48, (1996).
3. G. Canfora, A.., Cimitile, M. and Munro, "An Improved Algorithm for Identifying Objects in Code", Software Practice and Experience, Vol. 26, No. 1, pp. 25-48, (1996).
4. G. Canfora, A. Cimitile, and M. A Munro, "Reverse Engineering Method for Identifying Reusable Abstract Data Types", Proceeding of the First Working Conference on Reverse Engineering, Baltimore, Maryland, pp. 73-82, (1993).
5. M.F. Dunn, and J.C. Knight, "Automating the Detection of Reusable Parts in Existing Software", Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, pp. 381-390, (1993).
6. S. K. Abd-El-Hafiz, "Effects of Decomposition Techniques on Knowledge-Based Program Understanding", Proceedings of the International Conference on Software Maintenance, Bari, Italy, pp. 21-30, (1997).
7. S. K. Abd-El-Hafiz, and V. R. A Basili, "Knowledge-Based approach to Program Understanding", Kluwer Academic Publishers, (1995).
8. P. Newcomb, "Reengineering Procedural Into Object-Oriented Systems", Proceeding of the Second Working Conference on Reverse Engineering, Toronto, Ontario, Canada, pp. 237-249, (1995).
9. M. Siff, and T. Reps, "Identifying Modules Via Concept Analysis", Proceedings of the International Conference on Software Maintenance, Bari, Italy, pp. 170-179, (1997).
10. A. Yeh, D. R. Harris, and H.B. Reubenstein, "Recovering Abstract data Types and Object Instances from a Conventional Procedural language", Proceeding of the Second Working Conference on Reverse Engineering, Toronto, Ontario, Canada, pp 227-236, (1995).
11. A. Cimitile, G. and Visaggio, "Software Salvaging and the Call Dominance Tree", The Journal of Systems and Software, Vol. 28, No. 2, pp.117-127, (1995).
12. S. Liu, N. and Wilde, "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery", Proceedings of the Conference on Software Maintenance, San Diego, California, pp. 266-271, (1990).
13. C. Lindig, and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis", Proceedings of the 19th International Conference on Software Engineering, pp. 349-359, (1997).
14. H.A. Sahraoui, W. Melo, H. Lounis, F. Dumont, "Applying Concept Formation Methods to Object Identification in Procedural Code", Technical Report CRIM-97/05-77, CRIM, (1997).
15. G. Snelting, "Reengineering of Configurations Based on Mathematical Concept analysis", ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 2, pp.146-189, (1996).
16. B.L. Achee, D. L. and Carver, "A Greedy Approach to Object Identification in Imperative Code", Proceedings of the Third Workshop on Program Comprehension, pp. 4-11, (1994).
17. D. H. Hutchens, and V.R. Basili, "System Structure Analysis: Clustering With Data Binding", IEEE Transaction on Software Engineering, SE-11, No. 8, pp.749-757, (1985).
18. R. Ibba, D. Natale, P. Benedusi and R. Naddei, "Structure-Based Clustering of

Components for Software Reuse", Proceedings of the International Conference on Software Maintenance, Montreal, Quebec, Canada, pp. 210-215, (1993).
19. T. Kunz, "Evaluating Process Clusters to Support Automatic Program Understanding", Proceedings of the Fourth Workshop on Program Comprehension, pp. 198-207, (1996).
20. A. K. Jain, J. Mao and K. M. Mohiuddin, "Artificial Neural Networks: a Tutorial", IEEE COMPUTER, Vol. 29 No. 3, 31-44, (1996).
21. K. Mehrotra, C.K. Mohan, and S. Ranka, "Elements of Artificial Neural Networks", The MIT Press, (1997).
22. J. Zurada, "Introduction to Artificial Neural Systems", West Publishing Company, (1992).
23. K. Knight, "Connectionist Ideas and Algorithms", Communications of the ACM, Vol. 33, No. 11, 59-74, (1990).
24. P. E. Livadas, and T. Johnson, "A New Approach to Finding Objects in Programs", Software Maintenance: Research and Practice, Vol. 6 pp. 249-260, (1994).
25. M. Weiser, "Program Slicing", IEEE Trans. on Software Engineering, SE-10, No. 4, pp. 352-357, (1984).
26. W.B. Frakes, C. J. Fox, and B.A. Nejmeh, "Software Engineering in the UNIX/C Environment", Prentice Hall, (1991).
27. P. Jalote, "An Integrated Approach to Software Engineering", Springer-Verlag, (1991).

Received June 8, 1998
Accepted March 25,1999

استخدام الشبكات العصبية فى التعرف على الوحدات البرمجية

سلوى كمال عبد الحفيظ
قسم الرياضيات والفيزيقا الهندسية - جامعة القاهرة

ملخص البحث

يقدم هذا البحث طريقة عامة للتعرف على الوحدات البرمجية. وتعتمد هذه الطريقة على المعمار العصبى القادر على استنباط الكتل المتماثلة بدون الحاجة لأى اشراف . ويتم وصف معمارين عصبيين وشرح كيفية استخدامهم فى التعرف على الوحدات البرمجية فى نظم البرمجيات . كما يتم أيضا وصف نموذج أداه لاستنباط الكتل المتماثلة . وعن طريق عدة أمثلة يشرح البحث كيفية التعرف على أنواع البيانات المطلقة "abstract data types" ومجاميع البرمجيات التى تستخدم بيانات مشتركة . تتم مقارنة نتيجة التكتيل "clustering" بنتائج أخرى للتعرف على الوحدات البرمجية . ولتقييم الطريقة المقدمة تطبق على اثنين من البرامج وتعرض وتناقش نتائج تطبيقها.