Available online at www.sciencedirect.com

ScienceDirect

Fuzzy Sets and Systems 161 (2010) 2515–2532

FUZZY sets and systems

www.elsevier.com/locate/fss

A divide and conquer method for learning large Fuzzy Cognitive Maps

Wojciech Stach*, Lukasz Kurgan, Witold Pedrycz

Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada

Received 25 June 2009; received in revised form 26 January 2010; accepted 9 April 2010
Available online 8 May 2010

Abstract

Fuzzy Cognitive Maps (FCMs) are a convenient tool for modeling and simulating dynamic systems. FCMs were applied in a large number of diverse areas and have already gained momentum due to their simplicity and easiness of use. However, these models are usually generated manually, and thus they cannot be applied when dealing with large number of variables. In such cases, their development could be significantly affected by the limited knowledge and skills of the designer. In the past few years we have witnessed the development of several methods that support experts in establishing the FCMs or even replace humans by automating the construction of the maps from data. One of the problems of the existing automated methods is their limited scalability, which results in inability to handle large number of variables. The proposed method applies a divide and conquer strategy to speed up a recently proposed genetic optimization of FCMs. We empirically contrast several different designs, including parallelized genetic algorithms, FCM-specific designs based on sampling of the input data, and existing Hebbian-based methods. The proposed method, which utilizes genetic algorithm to learn and merge multiple FCM models that are computed from subsets of the original data, is shown to be faster than other genetic algorithm-based designs while resulting in the FCMs of comparable quality. We also show that the proposed method generates FCMs of higher quality than those obtained with the use of Hebbian-based methods.
© 2010 Elsevier B.V. All rights reserved.

Keywords: Fuzzy Cognitive Maps; Qualitative modeling; Inductive learning; Genetic algorithms; Parallel genetic algorithms

1. Introduction

Fuzzy Cognitive Maps (FCMs) [19] are a tool for modeling dynamic systems, which combine elements of neural networks and fuzzy logic. They were introduced as an extension of Cognitive Maps [2], and originally applied to problems concerning political science. Their main advantages are flexibility and adaptability to a given domain [1]. In addition, FCMs come with a convenient graph representation, in which concepts are represented as nodes and weighted edges represent knowledge about associations between the concepts. This technique is particularly suitable to qualitative rather than quantitative models. Quantitative modeling often is not suitable to describe complex systems with strong nonlinearities and unknown physical behavior [4]. Being a qualitative approach, FCMs are free from most of the drawbacks that are inseparable with quantitative modeling. FCMs were applied to a significant number of domains such as engineering, medicine, political sciences, earth and environmental sciences, economics and management, etc. see e.g., [1]. Examples of specific applications include medical diagnostics [15,16], medical decision making [26], analysis

* Corresponding author.
E-mail addresses: wstach@ece.ualberta.ca (W. Stach), lkurgan@ece.ualberta.ca (L. Kurgan), pedrycz@ece.ualberta.ca (W. Pedrycz).

0165-0114/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.fss.2010.04.008

of electrical circuits [39], failure modes effects analysis [28], fault management in distributed networks [23], modeling of complex technological systems [41], modeling of software development project [31,36], time-series prediction [38], trust dynamics analysis in virtual enterprises [43], renewable energy system efficiency improvements [21], business process reengineering [44], and many others. The scope and range of the applications demonstrate usefulness of this method and motivate further research in this area.

There are two main groups of approaches to develop FCMs, namely (a) using an expert knowledge about the domain of application (deductive modeling) and (b) using learning algorithms to establish FCMs based on historical data (inductive modeling) [30]. Even though the methods from the first group are well-established, they suffer from several important drawbacks. Firstly, they require domain knowledge that has to be supplemented by expertise in FCM methodology. Also, since the entire model has to be comprehended by a human expert, the application of these methods is limited to relatively simple systems, i.e., the human-developed maps usually consist of 5–10 nodes [34]. Mutual relationships among a large number of concepts are hard to comprehend, analyze, and quantify, which results in substantial difficulties in the construction of the corresponding maps. Secondly, the models obtained from these methods are based entirely on human expertise and relevant tools to represent it in the form of the corresponding map. This results in difficulties to complete an unbiased assessment of the accuracy of these models [30]. The abovementioned drawbacks motivated the development of automated and semi-automated approaches for learning FCM models. We refer to this group of development strategies as computational methods and they are classified as inductive modeling methods.

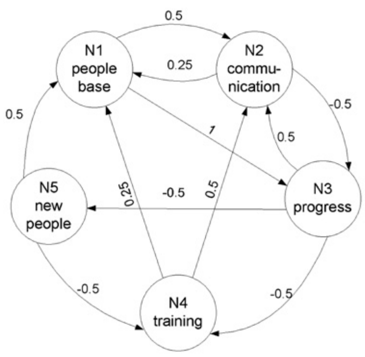
One of the recently investigated methods used in automated learning of FCMs from data is based on real-coded genetic algorithm (RCGA) [34,35]. This method does not require human intervention and provides high quality models that are generated from historical data. Another advantage of this method, which is especially useful in the context of the divide and conquer approach proposed in this paper, is the ability to use various strategies to subdivide the input data to speed up the learning process [34,35]. However, one of the main disadvantages of RCGA is its mediocre scalability as the number of parameters to be established grows quadratically with the size of the FCM model (number of nodes). This is because the genetic optimization applied to this modeling is time consuming especially when dealing with large number of variables. At the same time, in some areas such as systems biology [29], the underlying networks that could be modeled with FCMs are large and consist of at least several dozens of nodes. These issues call for the development of learning approaches that would be fast enough to be applied to larger systems while still assuring generation of high quality models. Therefore, we propose a novel approach to speed-up the learning process based on a divide and conquer strategy. We provide a comprehensive suite of empirical experiments that compare several designs, which include two solutions based on parallelization of the genetic algorithm and two FCM-specific methods that sample the input data, to analyze trade-offs between the learning time and the quality of the generated FCM models. The proposed method, which computes the model by learning and combining multiple FCMs developed from subsets of the input data, is shown to be the fastest among the genetic algorithm based approaches. At the same time, the quality of the generated models is comparable with the quality of models computed when parallelizing the genetic algorithm and when using the sequential RCGA method. We also compare these designs with the existing methods that are based on Hebbian learning. Although the Hebbian based methods scale well to large problems, we show that they generate models of lower quality.

This paper is organized as follows. Section 2 presents some background material on Fuzzy Cognitive Maps including methods supporting their development. In Section 3, we provide motivation and introduce the proposed algorithm. Section 4 presents experimental results along with their analysis. Section 5 provides some conclusions and outlines future research directions.

2. Background

2.1. Fuzzy Cognitive Maps

Proposed in 1986 by Kosko [19], Fuzzy Cognitive Maps (FCMs) are a class of discrete-time artificial neural networks. They represent knowledge in a symbolic manner and relate states, variables, events, outputs and inputs using a cause and effect approach. Each of FCM's edges is associated with a weight value that reflects the strength of the corresponding relation. This value is usually normalized to the interval $[-1,1]$. Positive values describe promoting effect, while negative ones describe inhibiting effect. The value of -1 represents full negative, +1 full positive and 0 denotes neutral relation.



	N1	N2	N3	N4	N5
N1	0	0.5	1	0	0
N2	0.25	0	-0.5	0	0
N3	0	0.5	0	-0.5	-0.5
N4	0.25	0.5	0	0	0
N5	0.5	0	0	-0.5	0

Fig. 1. Example FCM model for software development projects. Left panel shows the FCM graph and right panel shows the corresponding connection matrix.

Other values correspond to different intermediate levels of the causal effect. The graph representation is equivalent to a square matrix, called connection matrix, which stores all weight values of edges between corresponding concepts. Fig. 1 shows a simple example of FCM (graph and connection matrix) that models software development project [31].

In FCMs, each node quantifies a degree to which the corresponding concept in the system becomes active at a particular iteration. This value, called activation level, is a number located in the [0,1] interval where 0 denotes no activation, whereas 1 represents full activation. For a given concept, the activation level is determined by taking into account the activation levels at the previous iteration of all other concepts that exert influence on it:

$$\forall j \in \{1, \ldots, N\}, \quad C_j(t+1) = f\left(\sum_{i=1}^{N} e_{ij} C_i(t)\right) \tag{1}$$

where $C_j(t)$ is the activation level of *j*th concept at iteration *t*; e_{ij} the strength of relation from concept C_i to concept C_j; *f* the transformation function.

The transformation function is used to reduce unbounded weighted sum to a certain range. The normalization hinders quantitative analysis, but, at the same time, it allows for a comparative analysis of activation levels of the concepts.

A snapshot of the activation levels of all nodes at a particular iteration defines the system state. It can be conveniently represented by a vector, called state vector, which consists of the nodes' activation values. Initial state vector refers to the system state at the first iteration. Successive states are calculated by iterating over *t* in (1).

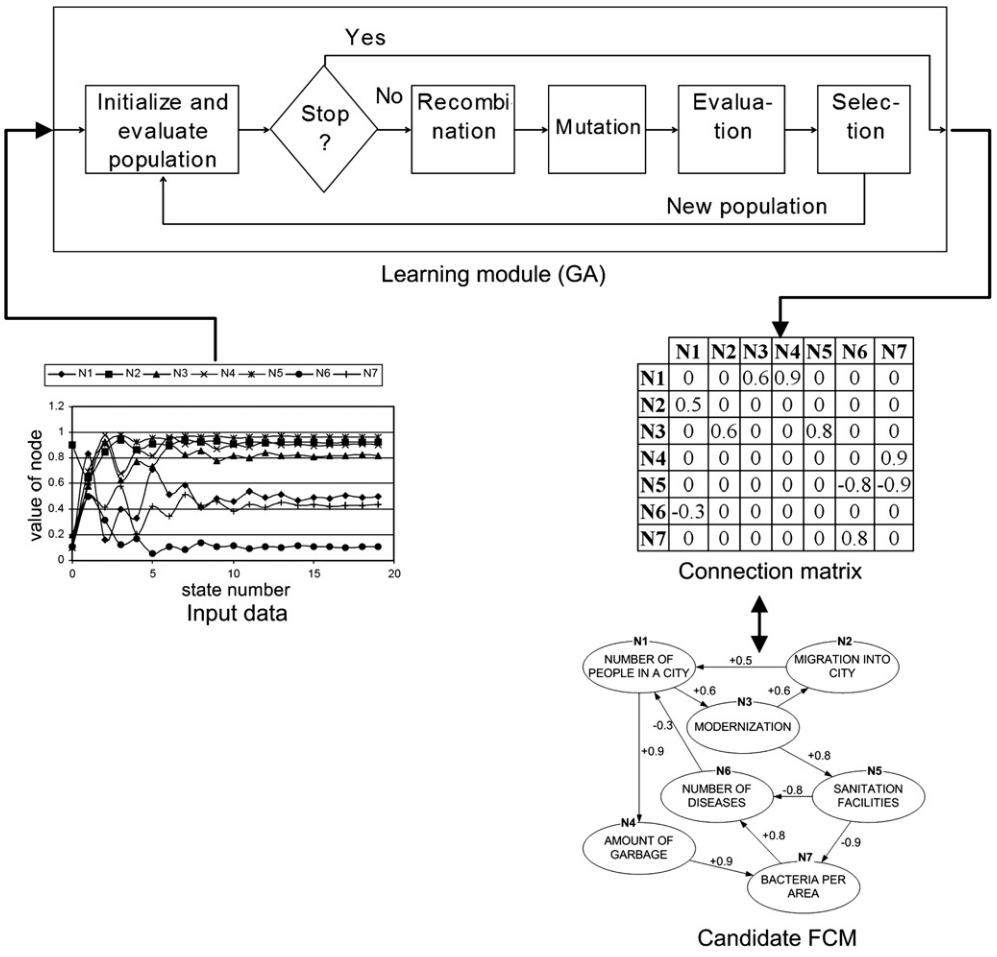
The results produced by the map directly depend on the type of transformation function. Discrete-output functions lead the simulation into either hidden pattern (fixed-point attractor) or limit cycle. The former term refers to a scenario, in which the state vector is kept unchanged after a certain number of iterations. Limit cycle describes a situation, in which the system keeps cycling between certain states. When the transformation function is continuous, the system may produce fixed-point attractor, limit cycle, and chaotic attractor, in which case different state vectors are computed in successive iterations.

The development methods for Fuzzy Cognitive Maps are designed for learning the connection matrix, i.e., casual relationships (edges), and their strength (weights) based on available historical data. In this case, the expert is substituted by historical data and the underlying learning algorithm using which we estimate the entries of the connection matrix. Several algorithms for automated, inductive learning of FCMs have been recently proposed. In general, two learning paradigms are used, that is Hebbian learning and genetic algorithms. Table 1 summarizes the main features of the learning approaches. The comparison is made based on several factors, such as the learning goal, involvement of a domain expert, usage of historical data, and learning strategy type. We observe that all methods, except BDA [14], were designed for FCMs that utilize continuous transfer function, which leads to a more generic solution. Also, one of the learning methods, GS [22], aims at learning an initial vector rather than the connection matrix, which is the goal of the proposed approach. The GS and PSO [27] methods require multiple sequences of state vectors, which may not be available in some domains. They also require longer learning. Finally, most of the methods were developed and tested only with small FCMs that include several concepts. Only the RCGA method was successfully used to generate FCMs when more than 20 concepts were considered. The contents of Table 1 and the above discussion suggest that the two

Table 1
Comparison of inductive methods used in learning FCMs.

Algorithm	Reference	Learning goal	Human intervention	Type of data used[a]	Transformation function	No. of nodes	Learning algorithm
DHL	[9]	Connection matrix	No	Single	N/A	N/A	Hebbian
BDA	[14]	Connection matrix	No	Single	Binary	5,7,9	Modified Hebbian
NHL	[25]	Connection matrix	Yes and No[b]	Single	Continuous	5	Modified Hebbian
DD-NHL	[37]	Connection matrix	Yes and No[b]	Single	Continuous	5	Modified Hebbian
AHL	[24]	Connection matrix	Yes and No[b]	Single	Continuous	8	Modified Hebbian
GS	[22]	Initial vector	No	Multiple	Continuous	7	Genetic
PSO	[27]	Connection matrix	No	Multiple	Continuous	5	Swarm
RCGA	[34]	Connection matrix	No	Single	Continuous	4,6,8,10	Genetic
Parallel RCGA	[33]	Connection Matrix	No	Single	Continuous	10, 20, 40, 80	Parallel genetic
SA	[10]	Connection matrix	No	Single	Continuous	2–15	Simulated annealing
Divide and conquer RCGA	This paper	Connection matrix	No	Single	Continuous	5, 10, 20, 40	Genetic

[a]Single—historical data consisting of one sequence of state vectors, Multiple—historical data consisting of several sequences of state vectors, for different initial conditions.
[b] Initial human intervention is necessary but later when applying the algorithm there is no human intervention needed.



	N1	N2	N3	N4	N5	N6	N7
N1	0	0	0.6	0.9	0	0	0
N2	0.5	0	0	0	0	0	0
N3	0	0.6	0	0	0.8	0	0
N4	0	0	0	0	0	0	0.9
N5	0	0	0	0	0	-0.8	-0.9
N6	-0.3	0	0	0	0	0	0
N7	0	0	0	0	0	0.8	0

Fig. 2. High-level conceptual diagram of RCGA learning method for FCMs.

best choices to implement scalable methods for learning FCMs are RCGA and Hebbian methods; the latter methods are very fast and should scale well to larger sizes of the maps.

Hebbian-based methods use available data and a learning formula, which is based on several modifications of Hebbian law that resulted in different algorithms listed in Table 1, to iteratively adjust FCM weights. The weights are being adjusted until the stopping criterion, which is based on constraints imposed on some or all nodes, is satisfied. As a result, these methods do not aim at finding a model that would mimic the entire available training data, but rather at finding a model that would converge to a given stable state (fixed-point attractor). They are very fast since they do not include computationally expensive calculations. On the other hand, genetic-based learning has been oriented towards finding models that mimic the input data, but they use optimization techniques that are computationally quite demanding. Fig. 2 shows a high-level diagram of the RCGA learning method.

The RCGA method develops an FCM model (called *candidate FCM*) using the input data that are given as time series and that consist of a sequence of state vectors that describe a given system at consecutive discrete time moments (iterations). All data points are normalized to the [0,1] range and correspond to the degree of presence of a given concept at a particular iteration. Since FCM can be fully represented by its connection matrix, the learning goal is to establish N^2 parameters, where N denotes the number of concepts. They reflect weighted values of mutual relations between the concepts. The learning module uses genetic optimization (real-coded genetic algorithms described in the next section) and the input data to find these values. The learning objective is to generate the same state vector sequence from the candidate FCM for the same initial state vector, as it is defined in the input data. At the same time, the candidate FCM generalizes the inter-relations between concept nodes, which are inferred from the input data. Therefore, it allows performing simulations from different initial state vectors and, based on their results, to draw conclusions about the modeled system. A comprehensive overview of the learning methods can be found in [30].

Our main objective is to construct a method of better scalability when compared with scalability of other existing methods while maintaining high quality of the solution provided by it. Among the methods listed in Table 1, we selected RCGA approach as the basis for our design since this method is characterized by generation of high quality FCM models from data [34], and since it can be easily adopted for parallelization [33] and for other approaches that accelerate the learning process. More specifically, we apply an FCM-specific approach to speed-up the learning in which we combine together several FCM models that are computed from subsets of the data to generate the final FCM model. We compare our approach with two representative Hebbian-based methods, NHL [25] and a recently introduced DD-NHL [37], as well as with the parallelized RCGA [33]. Besides the existing parallel implementation of the RCGA, our comparative study also includes a parallelization method based on multiple population parallel GAs.

2.2. Genetic algorithms

Genetic algorithms (GAs) are optimization tools, which origins come from the biological realm, i.e., they guide a population of solutions (chromosomes) that evolves over time using operators inspired by evolutionary biology, such as mutation, selection, and crossover. Each chromosome represents a solution to a given problem and its quality is quantified by a fitness value computed using a problem-dependent fitness function. The GAs usually start from a randomly generated population, which is modified and evolves over generations. In each generation, a new population is formed by using genetic operators and by exploiting fitness values of the chromosomes. The idea is to produce better solutions to the problem over the successive generations. GAs come with several advantages like broad applicability, ease of use, and ability to provide global solutions. The detailed information about GAs can be found in [11,8].

Several extensions to the generic GAs have been introduced. One of them, called real-coded genetic algorithms (RCGA), is used to implement method for learning FCMs. RCGA is a floating-point extension to basic genetic algorithms, which means that RCGA represents chromosomes as floating-point vectors, instead of binary vectors being used in the "classical" GAs. This makes RCGA more effective in tackling optimization problems with continuous variables. Although RCGA requires revising the genetic operators to handle floating-point values, the main principles of this GA technique and the classical GA are the same. A comprehensive review of the RCGA approach is presented in [12].

GAs are slow because of their inherently high computational complexity. However, they are relatively easy to parallelize, and several approaches that address the problem of scaling up GAs through parallelization have been developed. Generally speaking, these approaches can be divided into those that use a single population and those that use several subpopulations. Four main paradigms that are applied to parallelize GAs include [18,6]:

- global single-population master slave—a single population is maintained but the evaluation of fitness is performed by multiple processors;
- single-population fine-grained—a single population is maintained, but selection and crossover are restricted to a small neighborhood, yet some interaction among all the individuals is allowed;
- multiple-population coarse-grained—instead of maintaining a single population, in this approach several subpopulations exists and they occasionally exchange individuals
- hybrid models—combine some elements from the above three paradigms

Among the above four choices, the two most popular are global single-population master slave and multiple-population coarse-grained [5]. The former one, despite being very simple, has been shown to be very efficient [18,6]. The implementation usually follows the master–slave framework, where the master stores the population and the slaves evaluate the fitness of possible solutions. This is performed by assigning a fraction of the population to each of the available processors, see Fig. 3. The master stores the population, executes the GA operations, and distributes individuals to the slaves. The slaves evaluate the fitness of the individuals. Communication between processes occurs only when slaves receive a subset of individuals to evaluate and when they return the fitness values to the master process. In this approach, the underlying computer architecture is not constrained by any special requirements. This approach also does not affect the behavior of the genetic algorithm since no additional restrictions are imposed on the genetic operators such as crossover or selection. The second approach, which is based on multiple populations uses of a few relatively large subpopulations that exchange individuals (chromosomes), which is called migration, see Fig. 4. Each

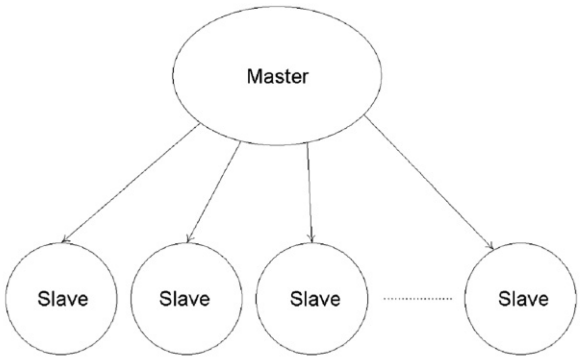


Fig. 3. Master–slave parallel GA architecture. Arrows describe assignment of chromosomes to slave processes for fitness function calculation.

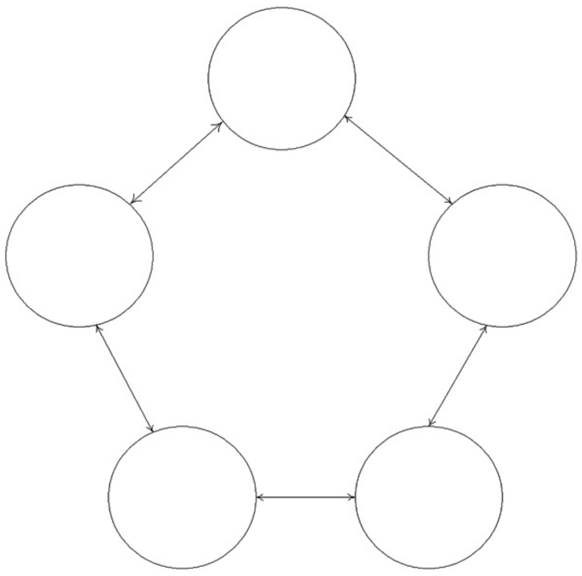


Fig. 4. Multiple-population parallel GA. Nodes correspond to subpopulations, whereas arrows show how the subpopulations exchange chromosomes.

subpopulation, denoted here by a circle, is executed as a standard GA and there is (infrequent) communication between the populations. In this example, the populations are arranged in a ring, but a few other communication topologies have been used [18,6].

3. Motivation and proposed approach

Although the RCGA method [34] was reported to be effective in terms of the quality of generated FCM models, the learning process, which utilizes genetic optimization, is time consuming, especially when applied to large maps that have dozen or more concepts. The reason behind it is the quadratic growth of the number of parameters that have to be optimized when compared with the system's size expressed by the number of concepts. Given a system that consists of N concepts, the corresponding FCM would be defined by N^2 parameters which express mutual relationships between all possible pairs of concepts. In addition, genetic optimization is time consuming when applied to problems with large number of variables. As a result, using a desktop PC the RCGA method could be used to learn FCMs which consist of up to 10 concepts, which limits the applicability of this learning method. Moreover, recently proposed extension to generic FCMs, the higher-order fuzzy cognitive maps [32], requires even more parameters to be established during the learning process. One of the straightforward solutions that would provide the speed-up is to parallelize the RCGA algorithm,

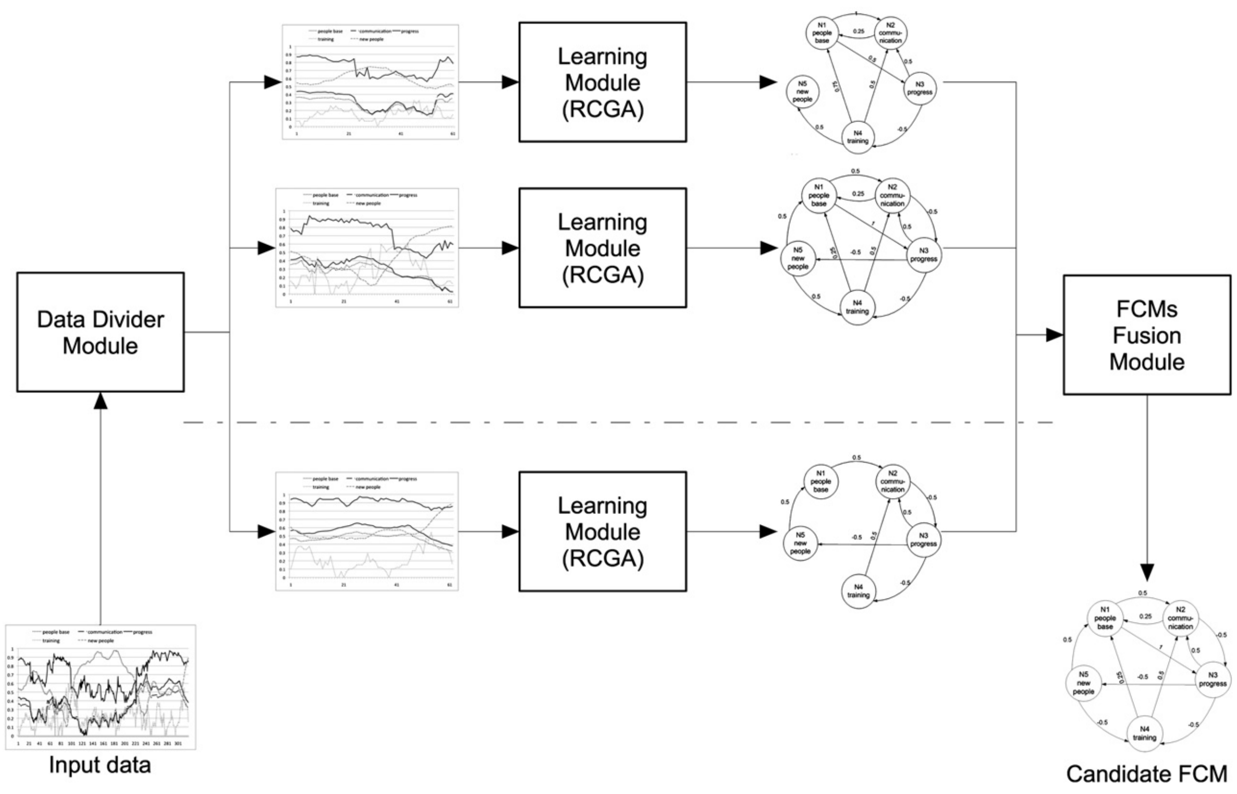


Fig. 5. High level diagram of the proposed method.

which was attempted in [33]. While conceptually simple, such approach does not take advantage of properties of FCMs that could potentially provide even further improvements. To this end, the following two goals are addressed:

- We propose a time-efficient method for learning FCMs based on RCGA and a divide and conquer strategy. The proposed method utilizes FCM-specific architecture in which we divide the input data into subsets, learn an FCM model from each subset and, finally, merge these models to generate the solution.
- We test and compare the new RCGA learning method with implementations based on straightforward parallelization of RCGA algorithm. We analyze the trade-off between the speed-up and the quality of the generated maps based on experiments with synthetic FCMs that consist of 5, 10, 20, and 40 concepts, as well as with a relatively large real-world FCMs that consists of 8, 12, 13, and 24 concepts.

In our recent work we used global single-population master slave method to parallelize the RCGA algorithms [33]. The reported results showed that learning of FCMs on eight processors was four times faster than the sequential learning, i.e., 2:1 ratio between the speed up and the number of processors used was achieved. In this work, we investigate whether an FCM-specific design could provide more significant improvements. The motivation to divide the input data into subsets comes from the linear relation between the number of calculations required to evaluate the fitness function, which is the most time-consuming part of the RCGA optimization, and the length of the input data. Therefore, reduction of the length of the input data could provide a linear decrease in the execution time when using a multiprocessor architecture (each sub-problem would be solved on a different processor). This could potentially provide a better speed-up than the abovementioned improvement observed when parallelizing the RCGA algorithm. Fig. 5 presents a high-level diagram of the proposed method. The two core modules are *Data Divider* and *FCMs fusion*.

Data Divider takes the input data, given as discrete time-series, and splits it into subsets that are used to learn the submodels. Various strategies could be implemented by taking advantage of inherent properties of FCMs. Firstly, the data may be divided into S non-overlapping contiguous subsets (intervals) where S is the number of available processors. Secondly, the split may be done as above except that the subsets would overlap to assure better similarity (and potentially better quality) between the submodels. Thirdly, as the RCGA method can use any two subsequent data points to learn FCMs, the subsets can be obtained by random sampling of the data point pairs. Finally, the random sampling with replacement could be used. The motivation for the latter case is the same as for the second strategy. While these strategies are computationally equivalent (strategies 1 and 3, and 2 and 4 would use the same number of input data point pairs), the first two strategies may result in generating submodels that overfit their corresponding input intervals. On the contrary, the latter two random sampling-based strategies include data points that cover the entire input time-series. Their submodels should better generalize to describe the entire input time series and this is why we concentrate on these two strategies. We empirically investigate the impact of the amount of overlap between the subsets (when allowing for the replacement) on the trade-off between the amount of computations and the quality of the learned FCM model.

Each of the data subsets is used to generate a separate submodel. This process involves running, in parallel, S independent experiments using *RCGA learning module*. Therefore, the learning modules do not exchange chromosomes in our method—there is no migration. Each experiment takes a given data subset and establishes the corresponding submodel using the RCGA learning module. This module exploits RCGA to develop a model which, when simulated, mimics the input data subset. This corresponding optimization problem requires establishing N^2 parameters for a system that is compounded of N concepts. Consequently, the chromosome structure is defined $\hat{\mathbf{E}} = [e_{11}, e_{12}, e_{13}, \dots, e_{1N}, e_{21}, e_{22}, e_{23}, \dots, e_{2N}, \dots, e_{NN}]^T$ where e_{ij} is the relationships strength from ith to jth concept. The RCGA maintains the population of chromosomes that represent solutions to the learning problem and which evolve with time by employing genetic operations, such as selection, crossover, and mutation [12]. Fitness function (or fitness, for short) evaluates chromosome's quality, and is defined by taking advantage of an inherent property of FCM execution model. More specifically, the state of a given FCM at each iteration, except for the initial one, depends only on the immediately preceding state. In other words, FCM does not have memory. This observation allows breaking the input data into pairs {initial vector, system response}, in which the former is a system state at given iteration t, and the latter stands for system state at immediately succeeding iteration $t + 1$. The fitness function is expressed as [35]

$$f = \alpha \frac{1}{\beta \cdot \sum_{t=1}^{K-1} \sum_{n=1}^{N} (C_n(t) - \hat{C}_n(t))^2 + 1} \tag{2}$$

where $\mathbf{C}(t) = [C_1(t), C_2(t), \dots, C_N(t)]$ is the given system response for $\mathbf{C}(t-1)$ initial vector; $\hat{\mathbf{C}}(t) = [\hat{C}_1(t), \hat{C}_2(t), \dots, \hat{C}_N(t)]$ the candidate FCM response for $\mathbf{C}(t-1)$ initial vector; K the number of input data points (observations); N the number of concepts; α, β the positive scaling constants.

The fitness function exploits $K - 1$ single-step simulations of the candidate FCM, and compares each result with the corresponding given data.

FCMs fusion module merges the submodels. The subject of combining multiple FCMs into a single model is described in literature in the context of deductive modeling [1]. The simplest method is to establish the candidate FCM by averaging the corresponding relationship values (weights) across all the submodels. Using the FCM matrix representation, this is carried out by calculating averages of the corresponding cells across all the submodels. This method can be augmented by adding credibility (confidence) coefficient to each submodel, which results in computing weighted average of the weights. The coefficients can be calculated using in-sample error value, i.e., difference between the actual data and the data simulated using a given submodel. The in-sample error values quantify the quality of the submodels. Consequently, higher credibility is assigned to submodels with lower in-sample error value, and vice versa. Specifically, given S submodels and their in-sample values, the fusion is carried out using the following expression:

$$e_{ij} = \frac{\sum_{s=1}^{S} cr_s \cdot e_{ij}^s}{\sum_{s=1}^{S} cr_s} \tag{3}$$

where cr_s is the credibility of sth submodel (calculated as 1-in-sample error of the submodel); e_{ij}^s the weight between concept C_i and C_j in sth submodel.

We decided to use the in-sample error since the number of points for learning of each submodels decreases along with increasing the number of submodels and since an insufficient size of input data may result in poor quality of learning using RCGA [35]. Therefore, we used the entire input data instead of splitting them into the training and validation parts.

The trade-offs between using different parallelization paradigms and different methods for learning from data subsets are investigated using a set of empirical tests.

4. Experimental setup

The hardware used to execute the experiments is a state-of-the-art 12-way IBM p570 server powered by POWER5 processors. The code has been written in C++ using OpenMP, which is an application programming interface that supports multi-platform shared memory multiprocessing.

4.1. Evaluation criteria

There are three evaluation measures used in the experiments:

- *Execution time* evaluates a given learning algorithm in terms of the time that is needed to complete learning which is expressed in seconds. This measure does not include the time to load the input data and to perform model evaluation.
- *In-sample error* evaluates capability of the learned model to reconstruct the input data used for learning. In other words, this criterion measures the performance of the model on the previously seen data. It is defined as the difference between the input data and data generated by simulating the candidate FCM from the initial state vector for the input data. The criterion is defined as a normalized average absolute error between corresponding concept values at each iteration between the two sequences of state vectors [34]

$$in\text{-}sample\ error = \frac{1}{(K-1)\cdot N}\sum_{t=1}^{K-1}\sum_{n=1}^{N}|C_n(t)-\hat{C}_n(t)| \tag{4}$$

  where $C_n(t)$ is the value of node n at iteration t in the input data; $\hat{C}_n(t)$ the value of node n at iteration t from simulation of the candidate FCM; K the number of input data points; N the number of nodes.
- *Out-of-sample error* evaluates generalization capabilities of the candidate FCM. To compute this criterion, both the input model and the candidate FCMs are simulated from 10 randomly chosen initial state vectors. Therefore, the candidate FCM is evaluated on new data that were not used for learning. This measure, similarly to the in-sample error, is normalized and expressed as an average error per concept per iteration. In particular, (3) is used for each simulation to compare state vector sequences generated by the input and the candidate FCM, and an average of these values is computed [34].

$$out\text{-}of\text{-}sample\ error = \frac{1}{P\cdot(K-1)\cdot N}\sum_{p=1}^{P}\sum_{t=1}^{K-1}\sum_{n=1}^{N}|C_n^p(t)-\hat{C}_n^p(t)| \tag{5}$$

  where $C_n^p(t)$ is the value of a node n at iteration t for data generated by input FCM started from pth initial state vector; $\hat{C}_n^p(t)$ the value of a node n at iteration t for data generated by candidate FCM started from pth initial state vector; K the number of input data points; N the number of nodes; P the number of different initial state vectors.

The first criterion is used to test the speed-up in execution time, whereas the other two are used to evaluate the quality of simulations provided by the developed FCMs. They are consistent with the criteria reported in [34].

4.2. Data sets

We have used both synthetic and real-world data. In the first scenario, the data for each experiment were obtained by simulating randomly generated FCM models starting from a random initial vector. We grouped our experiments based on FCM's size that includes 5, 10, 20, and 40 concepts. Each test was executed in sequential fashion, as well as in parallel using 2, 4, and 8 processors. Additionally, we generated five independent input data for every setup where each input data includes 40 points. The in-sample results were averaged over the five inputs and for each of the four FCM sizes, four parallelization architectures, and each learning method under consideration. For each setup the out-of-sample data have been generated by simulating the models from 10 random initial vectors.

In the second scenario, four real-world maps reported in literature were used. We concentrated on relatively large maps considering that a recent survey shows that an average size map does not exceed a dozen of concepts [34]. We selected four maps that describe a model of a control process system (eight concepts) [40], a model of deforestation in the Brazilian Amazon (12 concepts) [17], factors affecting slurry rheology (13 concepts) [3], and factors in the adoption of educational software in schools (24 concepts) [13]. These models were simulated from initial vectors suggested in the original references or from a random initial vector in case the initial vector was not provided. Similarly to experiments with the synthetic data, the out-of-sample error was computed by simulating the original and the learned models from 10 randomly chosen initial vectors.

4.3. Learning methods under consideration

We have compared six learning methods:

- *Single population*: This is a parallelized version of RCGA learning method [33], in which the learning module has been implemented using global single-population master slave GAs. Each slave process executes on a separate processor and evaluates fitness function of a given subset of individuals.
- *Divide and conquer*: In this approach, the Data Divider module divides the input data without replacement. Therefore, if the input data length is *K* and the number of available processors (the same as the number of submodels) is *S*, then each experiment uses *K*/*S* data points to learn a given submodel.
- *Divide and conquer with oversampling*: In this case the Data Divider module selects the input data pairs with replacement. We allow *O*% (oversampling coefficient) data points to be used twice to learn different submodels. Therefore, if the input data length is *K*, the number of submodels is *N*, and the oversampling coefficient is *O*, then each experiment uses (*K*+*O*%**K*)/*N* data points to learn corresponding submodel. In our experiments we used different values of *O*, which include 25, 50, and 75, to analyze its influence on the evaluation measures.
- *Multiple population*: This is a parallelized version of RCGA learning method, in which the learning module has been implemented using multiple-population coarse-grained GAs. The number of populations is equal to the number of available processors, and each population is maintained on a designated processor. We used the design from Fig. 4 to perform migration among subpopulations involving top 10% of chromosomes after each 1000 iterations. These values were selected experimentally by taking into account convergence to final solution and the learning time.
- *NHL*: This is one of the Hebbian-based learning methods [25]. This learning algorithm takes the initial FCM and initial values of all concepts and iteratively updates the model using modified Hebbian law until the desired map is found. The initial FCM is generated randomly. This method outperforms all GA-based in terms of the execution time and thus it does not require parallelization.
- *DD-NHL*: This is the most recently introduced extension to Hebbian-based learning approach [37]. It extends NHL method by using historical data of the input concepts to provide improved quality of the learned FCMs. Similarly as NHL, DD-NHL outperforms all GA-based methods in terms of the execution time and thus it was not parallelized. Although when compared with NHL, DD-NHL improves the quality of the produced maps [37], our results show that the quality is still substantially lower than the quality of maps generated with the RCGA method.

For the proposed "divide and conquer" and "divide and conquer with oversampling" methods, we report results for experiments with random divisions of input data in the Data Divider module as they gave better results than the results obtained when dividing the input data into contiguous intervals. Also, slightly better results were obtained when the model merging in the FCMs Fusion module was carried out using weighted average based on in-sample error for each submodel (when compared with using the average), see Section 3 for details. In the remaining four approaches we do not split the input data into subsets during the learning.

5. Results

Table 2 summarizes the experimental results and presents the three quality criteria for the six learning methods run with varying number of processors. Columns with 5, 10, 20, and 40 nodes correspond to experiments with synthetic data, whereas columns with 8, 12, 13 and 24 nodes to experiments with the real-life model. The columns with execution time and in-sample error are averaged over five independent experiments performed with each setup, whereas the

Table 2
Summary of the experimental results.

# proc.	Execution time (s)								In-sample error								Out-of-sample error							
	5 nodes	8 nodes	10 nodes	12 nodes	13 nodes	20 nodes	24 nodes	40 nodes	5 nodes	8 nodes	10 nodes	12 nodes	13 nodes	20 nodes	24 nodes	40 nodes	5 nodes	8 nodes	10 nodes	12 nodes	13 nodes	20 nodes	24 nodes	40 nodes
Single population																								
1	564 ± 28.61	739 ± 42.34	1218 ± 35.22	1498 ± 41.91	1904 ± 40.22	3476 ± 47.22	4901 ± 50.22	11242 ± 58.88	0.0029 ± 0.0018	0.0032 ± 0.0021	0.0042 ± 0.0032	0.0052 ± 0.0036	0.0045 ± 0.0049	0.0064 ± 0.0043	0.0086 ± 0.0060	0.0199 ± 0.0210	0.1050 ± 0.1220	0.1061 ± 0.1293	0.1350 ± 0.1440	0.1431 ± 0.1483	0.1410 ± 0.1480	0.1510 ± 0.1490	0.1594 ± 0.1967	0.1630 ± 0.1870
2	290 ± 14.23	383 ± 19.64	634 ± 29.22	761 ± 33.90	1042 ± 32.22	1970 ± 38.22	2719 ± 41.60	5902 ± 51.22	0.0031 ± 0.0041	0.0036 ± 0.0046	0.0049 ± 0.0028	0.0048 ± 0.0033	0.0047 ± 0.0039	0.0065 ± 0.0048	0.0085 ± 0.0064	0.0195 ± 0.0212	0.1128 ± 0.1434	0.1139 ± 0.1411	0.1371 ± 0.1560	0.1370 ± 0.1529	0.1466 ± 0.1542	0.1476 ± 0.1380	0.1539 ± 0.1608	0.1658 ± 0.1842
4	190 ± 10.11	277 ± 14.26	474 ± 28.11	588 ± 34.58	710 ± 35.01	1316 ± 38.22	1895 ± 42.74	5008 ± 49.18	0.0035 ± 0.0038	0.0037 ± 0.0044	0.0045 ± 0.0051	0.0049 ± 0.0059	0.0051 ± 0.0065	0.0067 ± 0.0079	0.0092 ± 0.0117	0.0197 ± 0.0128	0.1152 ± 0.1013	0.1185 ± 0.1071	0.1428 ± 0.1332	0.1447 ± 0.1411	0.1498 ± 0.1280	0.1572 ± 0.1720	0.1607 ± 0.1721	0.1632 ± 0.1763
8	174 ± 9.83	252 ± 14.25	386 ± 22.88	444 ± 29.06	476 ± 29.28	916 ± 31.23	1246 ± 42.79	3784 ± 49.28	0.0032 ± 0.0033	0.0043 ± 0.004	0.0041 ± 0.0043	0.0049 ± 0.0051	0.0051 ± 0.0047	0.0061 ± 0.0071	0.0088 ± 0.0097	0.0204 ± 0.0199	0.1102 ± 0.1020	0.1144 ± 0.1071	0.1446 ± 0.1360	0.1454 ± 0.1414	0.1410 ± 0.1520	0.1582 ± 0.1660	0.1614 ± 0.1774	0.1655 ± 0.1881
Divide and conquer																								
1	564 ± 28.61	739 ± 42.34	1218 ± 35.22	1498 ± 41.91	1904 ± 40.22	3476 ± 47.22	4901 ± 50.22	11242 ± 58.88	0.0029 ± 0.0018	0.0032 ± 0.0021	0.0042 ± 0.0032	0.0052 ± 0.0036	0.0045 ± 0.0049	0.0064 ± 0.0043	0.0086 ± 0.0060	0.0199 ± 0.0210	0.1050 ± 0.1220	0.1061 ± 0.1293	0.1350 ± 0.1440	0.1431 ± 0.1483	0.1410 ± 0.1480	0.1510 ± 0.1490	0.1594 ± 0.1567	0.1630 ± 0.1870
2	294 ± 18.21	388 ± 24.22	618 ± 27.45	791 ± 33.49	958 ± 34.34	1770 ± 40.30	2501 ± 46.60	5634 ± 51.22	0.0843 ± 0.0780	0.0924 ± 0.0881	0.1128 ± 0.0860	0.1138 ± 0.0972	0.1212 ± 0.1220	0.1280 ± 0.1256	0.1315 ± 0.1369	0.1245 ± 0.1063	0.1169 ± 0.1202	0.1204 ± 0.1320	0.1463 ± 0.1385	0.1471 ± 0.1408	0.1501 ± 0.1210	0.1649 ± 0.1445	0.1825 ± 0.1586	0.1849 ± 0.1802
4	160 ± 10.74	212 ± 14.82	324 ± 23.80	411 ± 25.94	482 ± 26.33	902 ± 28.88	1290 ± 39.57	2836 ± 42.28	0.0956 ± 0.0860	0.1074 ± 0.0998	0.1223 ± 0.1730	0.1299 ± 0.1938	0.1328 ± 0.0980	0.1368 ± 0.1420	0.1402 ± 0.2216	0.1356 ± 0.1850	0.1299 ± 0.1580	0.1278 ± 0.1691	0.1598 ± 0.1623	0.1622 ± 0.1782	0.1621 ± 0.1330	0.1674 ± 0.1498	0.1799 ± 0.1466	0.1828 ± 0.1881
8	82 ± 6.34	108 ± 8.81	170 ± 8.23	218 ± 9.96	250 ± 13.12	454 ± 23.34	667 ± 32.91	1424 ± 34.44	0.1114 ± 0.1040	0.1221 ± 0.1206	0.1307 ± 0.1100	0.1313 ± 0.1210	0.1356 ± 0.1110	0.1482 ± 0.1540	0.1449 ± 0.2002	0.1472 ± 0.1860	0.1348 ± 0.1440	0.1392 ± 0.1469	0.1617 ± 0.1580	0.1658 ± 0.1612	0.1664 ± 0.1442	0.1797 ± 0.1568	0.1835 ± 0.1548	0.1839 ± 0.1792
Divide and conquer with oversampling 25%																								
1	564 ± 28.61	739 ± 42.34	1218 ± 35.22	1498 ± 41.91	1904 ± 40.22	3476 ± 47.22	4901 ± 50.22	11242 ± 58.88	0.0029 ± 0.0018	0.0032 ± 0.0021	0.0042 ± 0.0032	0.0052 ± 0.0036	0.0045 ± 0.0049	0.0064 ± 0.0043	0.0086 ± 0.0060	0.0199 ± 0.0210	0.1050 ± 0.1220	0.1061 ± 0.1293	0.1350 ± 0.1440	0.1431 ± 0.1483	0.1410 ± 0.1480	0.1510 ± 0.1490	0.1594 ± 0.1567	0.1630 ± 0.1870
2	371 ± 19.55	494 ± 28.15	748 ± 29.56	948 ± 33.35	1175 ± 28.20	2185 ± 34.15	3168 ± 39.86	6392 ± 41.57	0.0869 ± 0.1224	0.1022 ± 0.1144	0.1046 ± 0.0854	0.1045 ± 0.1093	0.1033 ± 0.0867	0.1071 ± 0.1027	0.1058 ± 0.1177	0.1089 ± 0.1306	0.1145 ± 0.1176	0.1172 ± 0.1247	0.1442 ± 0.1106	0.1436 ± 0.1084	0.1463 ± 0.1504	0.1519 ± 0.1826	0.1688 ± 0.1720	0.1736 ± 0.1828
4	207 ± 17.21	271 ± 25.64	389 ± 18.07	512 ± 22.59	605 ± 25.34	1145 ± 26.37	1626 ± 30.24	3428 ± 38.08	0.0958 ± 0.0880	0.1133 ± 0.1003	0.1128 ± 0.1483	0.1164 ± 0.1676	0.1173 ± 0.1722	0.1250 ± 0.1416	0.1225 ± 0.1410	0.1237 ± 0.1269	0.1168 ± 0.1380	0.1189 ± 0.1449	0.1548 ± 0.1716	0.1619 ± 0.1699	0.1589 ± 0.1607	0.1694 ± 0.1833	0.1799 ± 0.1848	0.1828 ± 0.1877
8	109 ± 6.18	139 ± 8.15	216 ± 15.77	268 ± 19.79	325 ± 17.49	602 ± 22.53	846 ± 24.32	1766 ± 26.78	0.1050 ± 0.1444	0.1176 ± 0.1098	0.1258 ± 0.1342	0.1241 ± 0.1557	0.1277 ± 0.1463	0.1325 ± 0.1426	0.1355 ± 0.1353	0.1397 ± 0.1290	0.1343 ± 0.1409	0.1368 ± 0.1543	0.1579 ± 0.1495	0.1629 ± 0.1599	0.1624 ± 0.1411	0.1752 ± 0.1628	0.1794 ± 0.1682	0.1836 ± 0.2051
Divide and conquer with oversampling 50%																								
1	564 ± 28.61	739 ± 42.34	1218 ± 35.22	1498 ± 41.91	1904 ± 40.22	3476 ± 47.22	4901 ± 50.22	11242 ± 58.88	0.0029 ± 0.0018	0.0032 ± 0.0021	0.0042 ± 0.0032	0.0052 ± 0.0036	0.0045 ± 0.0049	0.0064 ± 0.0043	0.0086 ± 0.0060	0.0199 ± 0.0210	0.1050 ± 0.1220	0.1061 ± 0.1293	0.1350 ± 0.1440	0.1431 ± 0.1483	0.1410 ± 0.1480	0.1510 ± 0.1490	0.1594 ± 0.1967	0.1630 ± 0.1870
2	444 ± 24.44	595 ± 35.19	906 ± 34.45	1069 ± 34.37	1424 ± 37.11	2648 ± 45.53	3654 ± 52.38	8450 ± 55.43	0.0846 ± 0.1102	0.0985 ± 0.1323	0.1021 ± 0.0880	0.1023 ± 0.1082	0.1028 ± 0.0540	0.1042 ± 0.1240	0.1064 ± 0.1810	0.1078 ± 0.1280	0.1128 ± 0.1200	0.1169 ± 0.1308	0.1439 ± 0.1142	0.1458 ± 0.1129	0.1421 ± 0.1554	0.1522 ± 0.1796	0.1638 ± 0.1752	0.1686 ± 0.1882
4	244 ± 21.78	349 ± 22.23	466 ± 23.47	555 ± 29.57	724 ± 25.12	1352 ± 33.38	2014 ± 41.73	4246 ± 48.82	0.0921 ± 0.0880	0.1041 ± 0.0968	0.1107 ± 0.1440	0.1140 ± 0.1829	0.1142 ± 0.1782	0.1190 ± 0.1430	0.1205 ± 0.2045	0.1225 ± 0.1220	0.1148 ± 0.1380	0.1211 ± 0.1435	0.1489 ± 0.1560	0.1502 ± 0.1638	0.1538 ± 0.1644	0.1669 ± 0.1785	0.1784 ± 0.1617	0.1802 ± 0.1882
8	122 ± 7.72	182 ± 11.04	258 ± 27.33	328 ± 35.53	374 ± 21.86	684 ± 28.89	1022 ± 33.34	2140 ± 33.47	0.0985 ± 0.1140	0.1105 ± 0.1084	0.1119 ± 0.1220	0.1188 ± 0.1501	0.1210 ± 0.1330	0.1250 ± 0.1440	0.1275 ± 0.1287	0.1351 ± 0.1240	0.1285 ± 0.1290	0.1298 ± 0.1371	0.1566 ± 0.1480	0.1569 ± 0.1584	0.1587 ± 0.1442	0.1705 ± 0.1480	0.1792 ± 0.1898	0.1815 ± 0.1911
Divide and conquer with oversampling 75%																								
1	564 ± 28.61	739 ± 42.34	1218 ± 35.22	1498 ± 41.91	1904 ± 40.22	3476 ± 47.22	4901 ± 50.22	11242 ± 58.88	0.0029 ± 0.0018	0.0032 ± 0.0021	0.0042 ± 0.0032	0.0052 ± 0.0036	0.0045 ± 0.0049	0.0064 ± 0.0043	0.0086 ± 0.0060	0.0199 ± 0.0210	0.1050 ± 0.1220	0.1061 ± 0.1293	0.1350 ± 0.1440	0.1431 ± 0.1483	0.1410 ± 0.1480	0.1510 ± 0.1490	0.1594 ± 0.1967	0.1630 ± 0.1870
2	512 ± 31.53	680 ± 46.03	1078 ± 44.44	1398 ± 44.36	1639 ± 47.50	3129 ± 59.64	4279 ± 61.12	10013 ± 72.61	0.0821 ± 0.1200	0.0879 ± 0.132	0.0958 ± 0.0862	0.1014 ± 0.0975	0.1041 ± 0.0535	0.1029 ± 0.1079	0.1066 ± 0.1021	0.1089 ± 0.1242	0.1125 ± 0.1032	0.1158 ± 0.1053	0.1424 ± 0.1117	0.1432 ± 0.1218	0.1424 ± 0.1612	0.1528 ± 0.1557	0.1608 ± 0.1680	0.1678 ± 0.1747
4	275 ± 28.75	378 ± 31.69	562 ± 29.81	712 ± 35.47	841 ± 38.65	1599 ± 42.39	2239 ± 51.05	5011 ± 62.49	0.0901 ± 0.0880	0.0988 ± 0.1012	0.1011 ± 0.1426	0.1083 ± 0.1597	0.1076 ± 0.1122	0.1166 ± 0.1502	0.1189 ± 0.1192	0.1198 ± 0.1147	0.1139 ± 0.1325	0.1167 ± 0.1338	0.1458 ± 0.1607	0.1512 ± 0.1575	0.1478 ± 0.1460	0.1602 ± 0.1769	0.1759 ± 0.1785	0.1754 v0.1808
8	142 ± 10.27	193 ± 15.09	295 ± 34.71	383 ± 31.65	441 ± 29.29	800 ± 36.11	1160 ± 42.36	2512 ± 43.51	0.0925 ± 0.0396	0.1074 ± 0.044	0.1069 ± 0.1220	0.1152 ± 0.1342	0.1198 ± 0.1290	0.1205 ± 0.1440	0.1224 ± 0.1344	0.1272 ± 0.1128	0.1193 ± 0.1082	0.1239 ± 0.1266	0.1562 ± 0.1539	0.1551 ± 0.1693	0.1582 ± 0.1498	0.1692 ± 0.1802	0.1748 ± 0.1888	0.1801 ± 0.1889
Multiple population																								
1	564 ± 28.61	739 ± 42.34	1218 ± 35.22	1498 ± 41.91	1904 ± 40.22	3476 ± 47.22	4901 ± 50.22	11242 ± 58.88	0.0029 ± 0.0018	0.0032 ± 0.0021	0.0042 ± 0.0032	0.0052 ± 0.0036	0.0045 ± 0.0049	0.0064 ± 0.0043	0.0086 ± 0.0060	0.0199 ± 0.0210	0.1050 ± 0.1220	0.1061 ± 0.1293	0.1350 ± 0.1440	0.1431 ± 0.1483	0.1410 ± 0.1480	0.1510 ± 0.1490	0.1594 ± 0.1567	0.1630 ± 0.1870
2	284 ± 22.38	392 ± 31.33	618 ± 36.73	791 ± 36.49	1022 ± 35.33	1888 ± 40.28	2605 ± 44.38	5830 ± 56.84	0.0022 ± 0.0023	0.0026 ± 0.0026	0.0043 ± 0.0048	0.0054 ± 0.0055	0.0053 ± 0.0048	0.0069 ± 0.0058	0.0090 ± 0.0077	0.0195 ± 0.0222	0.1095 ± 0.1280	0.1125 ± 0.1344	0.1387 ± 0.1335	0.1352 ± 0.1317	0.1383 ± 0.1189	0.1433 ± 0.1389	0.1497 ± 0.1418	0.1591 ± 0.1582
4	184 ± 11.21	270 ± 15.02	480 ± 22.11	581 ± 26.09	682 ± 26.42	1176 ± 39.65	1752 ± 43.13	4998 ± 52.34	0.0023 ± 0.0022	0.0026 ± 0.0024	0.0045 ± 0.0058	0.0055 ± 0.0069	0.0045 ± 0.0041	0.0069 ± 0.0048	0.0093 ± 0.0062	0.0194 ± 0.0166	0.1097 ± 0.1380	0.1145 ± 0.1408	0.1419 ± 0.1660	0.1466 ± 0.1726	0.1391 ± 0.1660	0.1513 ± 0.1484	0.1605 ± 0.1731	0.1635 ± 0.1801
8	170 ± 12.15	235 ± 17.13	370 ± 25.27	455 ± 28.56	460 ± 29.32	892 ± 30.12	1276 ± 43.67	3676 ± 49.32	0.0021 ± 0.0038	0.0024 ± 0.0044	0.0041 ± 0.0037	0.0051 ± 0.0041	0.0051 ± 0.0048	0.0068 ± 0.0058	0.0097 ± 0.0075	0.0205 ± 0.0188	0.1073 ± 0.1220	0.1113 ± 0.1281	0.1427 ± 0.1330	0.1434 ± 0.1383	0.1427 ± 0.1503	0.1528 ± 0.1667	0.1574 ± 0.1824	0.1603 ± 0.1742
DD-NHL																								
1	15 ± 1.12	20 ± 1.65	42 ± 2.32	47 ± 2.90	45 ± 2.87	96 ± 3.43	130 ± 3.53	216 ± 3.88	0.1893 ± 0.1880	0.2174 ± 0.2068	0.2012 ± 0.1860	0.2257 ± 0.2046	0.2008 ± 0.1980	0.2037 ± 0.2332	0.2125 ± 0.2138	0.2048 ± 0.2176	0.2103 ± 0.2010	0.2173 ± 0.2131	0.2021 ± 0.1803	0.2213 ± 0.1854	0.2013 ± 0.2105	0.2056 ± 0.2249	0.2081 ± 0.2096	0.2385 ± 0.2033
NHL																								
1	16 ± 1.44	21 ± 2.12	41 ± 1.87	53 ± 2.28	46 ± 2.92	94 ± 2.78	128 ± 3.81	214 ± 4.01	0.1925 ± 0.1870	0.2196 ± 0.2200	0.2101 ± 0.2010	0.2304 ± 0.2412	0.2117 ± 0.1952	0.2112 ± 0.1881	0.2198 ± 0.2144	0.1854 ± 0.1868	0.2131 ± 0.1890	0.2213 ± 0.2079	0.2102 ± 0.2260	0.2316 ± 0.2237	0.2151 ± 0.1880	0.2135 ± 0.2180	0.2113 ± 0.2033	0.2449 ± 0.1902

Rows show different learning methods, whereas columns correspond to different evaluation criteria. Each cell includes the average value followed by the standard deviation shown below using smaller fonts.

out-of-sample error was additionally averaged over 10 independent experiments performed with different initial vectors. Each cell includes two values, the average and the corresponding standard deviation.

5.1. Execution time

The fastest methods are these based on the Hebbian learning, i.e., NHL and DD-NHL. They learn the connection matrix weights based on a simple formula that performs local adjustments that quickly converge into the final solution. From among the four other methods which use genetic optimization the divide and conquer RCGA approach outperforms both single and multiple population GAs. When compared with the sequential learning on a single processor, the gain of the computational time is, on average, 86% and 79% for the divide and conquer RCGA without and with 50% oversampling, respectively, when using eight processors. In the case of the two other methods, i.e., the single and the multiple populations, the gain between the sequential implementation and when eight processors are used is approximately 70%. We also observe that doubling the number of processors results in almost 50% (between 46% and 50%) decrease of the execution time for the divide and conquer methods. On the other hand, we observe decreasing returns along with increasing the number of processors for both the single and multiple population GAs. For instance the decrease of the execution time for the single population GA when doubling the number of processors for 40-nodes FCM is 43% for two processors, 33% for four processors, and 30% for eight processors. This is due to the fact that the execution time that corresponds to the sequential constraints of performing genetic operations on population of chromosomes becomes a more significant part of the total execution time. The above overhead is caused by the situation where the fitness function evaluations performed in parallel are completed faster on larger number of processors [33]. For the multiple population GA, these numbers are 45%, 37%, and 24%, respectively. The decreasing return in this case stems from the fact that there is a nonlinear relationship between the population size and the execution time [7], as well as the migration among subpopulations [6].

Overall, excluding the Hebbian-based methods, the two methods based on the divide and conquer approach to learn FCMs are better than the methods that parallelize genetic algorithms using either single or multiple population. Comparison of the execution time of the divide and conquer method without oversampling with the single population method on eight processors reveals that we can almost double the size of FCMs that are learned using the former method within the same amount of time. For instance, using the former method, learning of 40-nodes FCM takes 1424 s, whereas time needed to learn 24-nodes FCM using the latter method is 1246 s. The execution time increases by $O\%$ on average for $O\%$ oversampling when compared to the divide and conquer approach; this observation is consistent over all considered setups.

5.2. Learning quality

A few interesting conclusions are drawn when analyzing the in-sample errors. Firstly, both Hebbian-based methods perform relatively poorly when compared with all genetic-based approaches. The average in-sample error values over all experiments for these methods are 0.210 for the NHL method and 0.207 for the DD-NHL, while the average error equals 0.006 for the genetic-based methods. The error values for the Hebbian-based methods are similar across different map sizes suggesting that their in-sample error does not depend on the map size. Secondly, for the two methods based on the parallelization of GAs, the in-sample error increases along with increasing the map size. This is a non-linear, exponential type of relationship, which is consistent with conclusions on the RCGA learning reported in literature [33]. On the other hand, the in-sample error of these two methods does not depend on the number of processors, i.e., the maximal standard deviation across experiments with a fixed map size is 0.0005. This is also consistent with results reported in [33]. Finally, for the divide and conquer RCGA approaches without and with oversampling the in-sample error increases with increasing the map size. This error is substantially higher than the error obtained through parallelization of GAs even when considering results for two processors. On the other hand, the error does not grow as rapidly as in the case of the two methods that parallelize GAs. In contrast, when increasing the number of nodes from 5 to 10, the in-sample error increases by approximately 20%, and when doubling the map size subsequently, this ratio decreases to 3% for 40 nodes. This stems from the fact that models developed for larger maps are of lower quality compared to the baseline calculated as performance of a randomly chosen FCM, which does not depend on the map size [35]. Consequently, given any or zero improvement when compared to a random map the error rates are upper bounded by this value and the abovementioned ratio decreases towards the value of zero for very large maps. When

Table 3
Results of the *t*-test at 95% confidence level that compares the out-of-sample errors of all methods with the errors of the best performing multiple population method.

Learning method	# processors	FCM size							
		5	8	10	12	13	20	24	40
Single population	2	=	=	=	=	=	=	=	=
	4	=	=	=	=	=	=	=	=
	8	=	=	=	=	=	=	=	=
Divide and conquer	2	=	=	=	=	=	=	+	+
	4	+	+	+	+	+	+	+	+
	8	+	+	+	+	+	+	+	+
Divide and conquer with oversampling 25%	2	=	=	=	=	=	=	+	=
	4	=	=	+	+	+	+	+	+
	8	+	+	+	+	+	+	+	+
Divide and conquer with oversampling 50%	2	=	=	=	=	=	=	+	=
	4	=	=	=	=	=	+	+	+
	8	+	+	+	+	+	+	+	+
Divide and conquer with oversampling 75%	2	=	=	=	=	=	=	=	=
	4	=	=	=	=	=	=	+	=
	8	+	+	+	+	+	+	+	+
DD-NHL	1	+	+	+	+	+	+	+	+
NHL	1	+	+	+	+	+	+	+	+

Equality (=) represents statistically insignificant differences, whereas plus (+) denotes that multiple population method has led to statistically significantly better results.

comparing the in-sample error for the divide and conquer approaches with either single or multiple population methods for the same map size, we note that the in-sample error is substantially higher and the difference increases along with increasing the number of processors. However, even for 40 nodes and eight processors, this value is significantly lower than the in-sample error for both Hebbian-based methods, i.e., 0.147 for method without oversampling and 0.135 with 50% oversampling vs. 0.185 for NHL and 0.205 for DD-NHL.

In order to facilitate analysis of the out-of-sample error, the statistical paired *t*-test analysis for all tested methods has been reported in Table 3. The purpose was to compare all other methods to the one that performed the best, i.e., multiple population, and analyze whether the obtained differences are statistically significant.

The differences between the single and the multiple population methods are shown to be insignificant across all experiments that include various numbers of processors and sizes of the maps. Divide and conquer approach without oversampling performs similarly to the best method for small maps and only in the case when two submodels are used, i.e., when two processors are utilized. Increasing the amount of the oversampling leads to improved results. When using the 50% oversampling the quality improves and the results are similar to the best results for all map sizes when using two processors and for smaller maps when using four processors. By increasing the oversampling to 75% the differences are statistically insignificant for all setups except one when using up to four processors. We observe that RCGA method provides higher quality models when more data points are available, which results in similar performance of the methods based on the parallelization of GA and the divide and conquer strategies.

For instance, when comparing the divide and conquer strategy with 50% oversampling to multiple population method for four processors and 10 nodes, the quality is comparable and the execution times equal 466 and 480 s, respectively. Comparison of the divide and conquer strategy without oversampling and the multiple population method for two processors and 20 nodes shows that the quality is comparable and the learning time is 1770 and 1888 s, respectively.

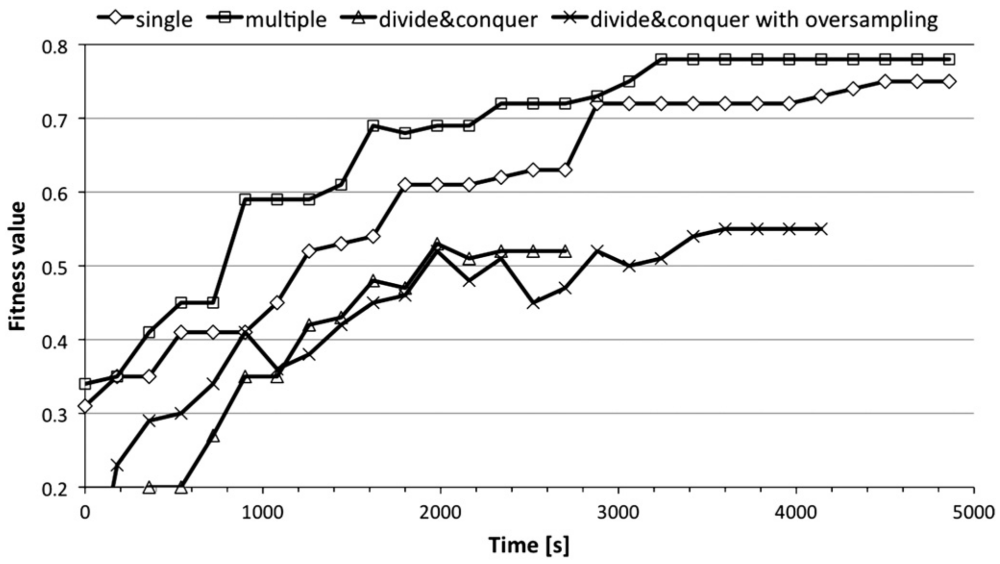


Fig. 6. Fitness value vs. time for the four methods that use genetic optimization. The experiments concern learning 40-nodes maps on four processors with the maximum number of iterations limited to 30,000.

Splitting the data into more processors results in a fewer input data points used to learn each submodel. This is particularly important in the context of learning maps with larger size, as in this case more data points is needed for learning in order to obtain high quality solution. As reported in our previous work [35], learning quality depends on the map size and the number of the available data points. Given a desired maximal in-sample error value, more data are needed for larger maps; detailed discussion is included in [35]. At the same time, the lower quality of the learned model is traded for the faster learning time. For example, the execution times equal 3784, 3676, 2140, and 1424 s when learning maps with 40 concepts on eight processors using single population, multiple population, divide and conquer with 50% oversampling, and divide and conquer without oversampling methods, respectively. Both Hebbian-based methods perform statistically significantly worse than the multiple population method (see Table 3) for all setups. When compared to the methods based on the genetic optimization the Hebbian-based method provide solutions with higher values of both in-sample and out-of-sample errors (see Table 2).

The out-of-sample error value decreases when the oversampling is used in the divide and conquer method. On average across all setups, adding extra 25% oversampling leads to 2–6% reduction in the out-of-sample error. The error reduction is within this range for all considered oversampling levels, i.e., 25%, 50%, and 75%. Also, it does not depend on the number of processors and, on average, it varies by as little as 2% when comparing configurations with two and eight processors.

5.3. Convergence

We also investigate differences with respect to the convergence to a final solution as a function of time. We analyze how fast the fitness value increases before reaching a plateau. Fig. 6 shows a sample plot that illustrates how the fitness value, which is inversely proportional to the in-sample error of the best-found solution, changes over the time of learning. Four methods were compared including single population, multiple population, divide and conquer without oversampling, and divide and conquer with 50% oversampling. This plot concerns experiments run on four processors and a map consisting of 40 nodes. The simulations were stopped by reaching the maximum number of iteration, which was equal to 30,000.

When comparing the single and the multi-population methods, the latter one converges faster and provides a better final solution as well as better solutions during the simulation, i.e., at the corresponding time points. Since these two methods always keep the best solution for the next iteration of the genetic algorithm evolution, the fitness function does not decrease over the time. Although the proposed methods based on the divide and conquer approach obtain

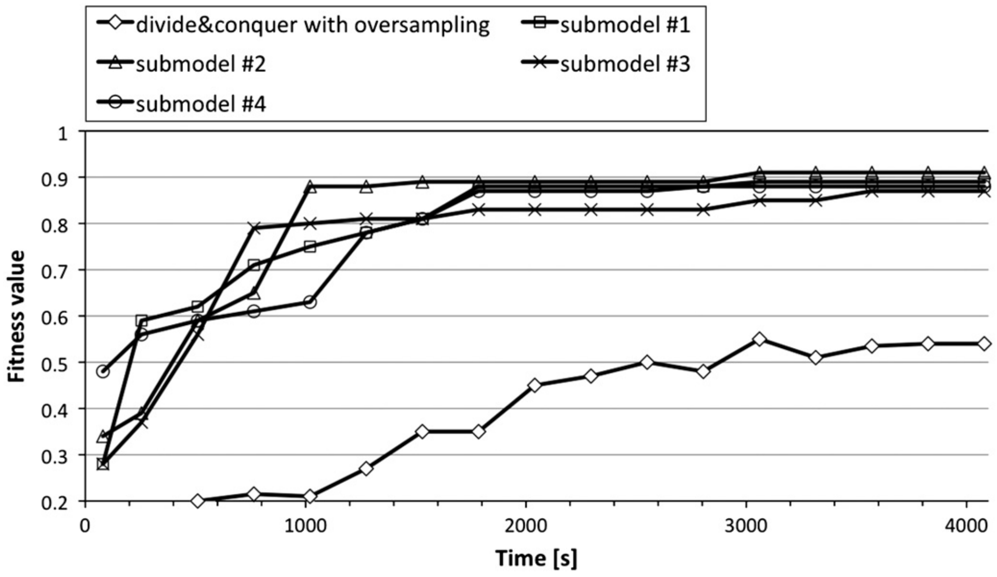


Fig. 7. Fitness value vs. time for the divide and conquer method. Included are the fitness values of the solution FCM model and the individual submodels for the experiment that was shown in Fig. 6.

lower value of the fitness function calculated for the entire input data set (inversely proportional to the in-sample error), they provide models of similar quality in terms of the out-of-sample error, see Tables 1 and 2. This suggests that these models are capable of generalizing the solution.

We observe that although the fitness function value may fluctuate (decrease and increase) over the time when using the divide and conquer approaches, the overall convergence trend is clearly visible. The main reason for the fluctuations is the fact that the solution generated by the divide and conquer method is based on averaging several submodels. Fig. 7 shows how the convergence of individual submodels and the solution FCM change over the time. The fitness value of each submodel is higher than the fitness of the merged model because it is calculated only for a given subset of data that is used to learn this submodel. As the simulation proceeds and the fitness values of individual submodels become higher, the fitness value of the merged solution increases. Although the fitness value of individual submodels does not decrease over the time, the fitness value of the solution may decrease due to the averaging, as can be observed after approximately 3100 s.

6. Conclusions

We have proposed and tested a new, scalable approach to learn high quality Fuzzy Cognitive Maps from experimental data. Our goal is to propose the approach that would substantially speed up the learning process for large systems while maintaining high quality of the solutions.

The proposed method, which is based on a divide and conquer strategy, involves dividing input data into subsets, performing independent, parallel learning of submodels using RCGA method, and merging the submodels into a final model. The two different scenarios based on this approach have been investigated. In the first one the entire dataset is divided without oversampling, whereas in the second one oversampling is used to increase quality of the generated models as a trade-off of the increased execution time. The amount of the introduced oversampling could be used to control the trade-off between the learning time and quality of the generated model.

Overall, the quality of the maps generated by the divide and conquer method decreases with the increasing size of the maps and the number of processors used. The time to compute the maps increases with the increased size and it decreases when using more processors. Increasing the oversampling lowers the rate with which the error grows, but it also lowers the savings in the computational time. When compared with the single and the multiple population-based approaches, the divide and conquer method provides larger speed ups when using more processors and when considering increasing map sizes as a trade-off for higher error rates.

The experimental results performed on eight processors show that the method without oversampling is up to seven times faster than the sequential learning. With 50% oversampling, the proposed method is up to five times faster than the sequential learning. We also compared the proposed solution with other commonly used methods, such as single and multiple population parallelizations of the RCGA method. The divide and conquer strategy without oversampling is shown to be up to three times faster than both parallelization methods.

The quality of the FCM model, which is measured using out-of-sample error, learned using the proposed method decreases along with increasing the number of processors. Statistical significance tests that compare quality of the FCM models obtained with the divide and conquer methods and the single and multiple population parallelization methods reveal that the method without oversampling provides comparable solutions for maps of up to 20-nodes and using two processors when compared with the best performing multiple populations based method. The oversampling at 50% results in improving the quality of the generated maps, i.e., their quality is comparable to the quality of the models generated by the best performing method for maps including up to 40-nodes when using two processors and up to 10-nodes for four processors. The oversampling at 75% results in comparable results across virtually all setups for up to four processors. The tests also demonstrate that the proposed method generates FCM models of quality that is significantly better that the quality of models computed using Hebbian-based methods.

References

[1] J. Aguilar, A survey about fuzzy cognitive maps papers, International Journal of Computational Cognition 3 (2) (2005) 27–33.

[2] R. Axelrod, Structure of Decision: The Cognitive Maps of Political Elites, Princeton University Press, Princeton, NJ, 1976.

[3] G.A. Banini, R.A. Bearman, Application of fuzzy cognitive maps to factors affecting slurry rheology, International Journal of Mineral Processing 52 (4) (1998) 233–244.

[4] R.J.G.B. Campello, W.C. Amaral, Towards true linguistic modelling through optimal numerical solutions, International Journal of Systems Science 34 (2) (2003) 139–157.

[5] E. Cantú-Paz, Parameter setting in parallel genetic algorithms, in: F. Lobo, C. Lima, Z. Michalewicz (Eds.), Parameter Setting in Evolutionary Algorithms. Studies in Computational Intelligence, Springer, 2007, pp. 259–276.

[6] E. Cantú–Paz, A survey of parallel genetic algorithms, Calculateurs Parallèles 10 (2) (1998) 141–171.

[7] E. Cantú-Paz, D. Goldberg, On the scalability of parallel genetic algorithms, Evolutionary Computation 7 (4) (1999) 429–449.

[8] K. Deb, An introduction to genetic algorithms, Sadhana 24 (4) (1999) 293–315.

[9] J.A. Dickerson, B. Kosko, Virtual worlds as fuzzy cognitive maps, Presence 3 (2) (1994) 173–189.

[10] M. Ghazanfari, S. Alizadeh, M. Fathian, D.E. Koulouriotis, Comparing simulated annealing and genetic algorithm in learning FCM, Applied Mathematics and Computation 192 (1) (2007) 56–68.

[11] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

[12] F. Herrera, M. Lozano, J.L. Verdegay, Tackling real-coded genetic algorithms: operators and tools for behavioural analysis, Artificial Intelligence Review 12 (4) (1998) 265–319.

[13] S. Hossain, L. Brooks, Fuzzy cognitive map modelling educational software adoption, Computers & Education 51 (4) (2008) 1569–1588.

[14] A.V. Huerga, A balanced differential learning algorithm in fuzzy cognitive maps, in: Proc. 16th Internat. Workshop on Qualitative Reasoning, 2002, poster.

[15] P.R. Innocent, R.I. John, Computer aided fuzzy medical diagnosis, Information Sciences 162 (2) (2004) 81–104.

[16] R.I. John, P.R. Innocent, Modeling uncertainty in clinical diagnosis using fuzzy logic, IEEE Transactions on Systems, Man, and Cybernetics Part B 35 (6) (2005) 1340–1350.

[17] K. Kok, The potential of fuzzy cognitive maps for semi-quantitative scenario development with an example from Brazil, Global Environmental Change 19 (1) (2009) 122–133.

[18] Z. Konfrst, Parallel genetic algorithms: advances, computing trends, applications and perspectives, Parallel and Distributed Processing Symposium (2004) 162–166.

[19] B. Kosko, Fuzzy cognitive maps, International Journal of Man-Machine Studies 24 (1986) 65–75.

[21] T.L. Kottas, Y.S. Boutalis, A.D. Karlis, New maximum power point tracker for PV arrays using fuzzy controller in close cooperation with fuzzy cognitive networks, IEEE Transactions on Energy Conversion 21 (3) (2006) 793–803.

[22] D.E. Koulouriotis, I.E. Diakoulakis, D.M. Emiris, Learning fuzzy cognitive maps using evolution strategies: a novel schema for modeling and simulating high-level behavior, in: Proc. IEEE Congr. on Evolutionary Computation, 2001, pp. 364–371.

[23] T.D. Ndousse, T. Okuda, Computational intelligence for distributed fault management in networks using fuzzy cognitive maps, in: Proc. IEEE Internat. Conf. on Communications Converging Technologies for Tomorrow's Application, Vol. 3, 1996, pp. 1558–1562.

[24] E. Papageorgiou, C.D. Stylios, P.P. Groumpos, Active Hebbian learning algorithm to train fuzzy cognitive maps, International Journal of Approximate Reasoning 37 (3) (2004) 219–249.

[25] E. Papageorgiou, C.D. Stylios, P.P. Groumpos, Fuzzy cognitive map learning based on nonlinear Hebbian rule, in: Proc. Australian Conf. on Artificial Intelligence, 2003, pp. 256–268.

[26] E.I. Papageorgiou, C.D. Stylios, P.P. Groumpos, An integrated two-level hierarchical system for decision making in radiation therapy based on fuzzy cognitive maps, IEEE Transactions on Biomedical Engineering 50 (12) (2003) 1326–1339.

[27] K.E. Parsopoulos, E.I. Papageorgiou, P.P. Groumpos, M.N. Vrahatis, A first study of fuzzy cognitive maps learning using particle swarm optimization, in: Proc. IEEE Congr. on Evolutionary Computation, 2003, pp. 1440–1447.

[28] C.E. Pelaez, J.B. Bowles, Applying fuzzy cognitive maps knowledge representation to failure modes effects analysis, in: Proc. IEEE Annu. Symp. on Reliability and Maintainability, 1995, pp. 450–456.

[29] R. Serra, M. Villani, A. Graudenzi, S.A. Kauffman, Why a simple model of genetic regulatory networks describes the distribution of avalanches in gene expression data, Journal of Theoretical Biology 246 (3) (2007) 449–460.

[30] W. Stach, L.A. Kurgan, W. Pedrycz, A survey of fuzzy cognitive map learning methods, in: P. Grzegorzewski, M. Krawczak, S. Zadrozny (Eds.), Issues in Soft Computing: Theory and Applications, Exit, 2005, pp. 71–84.

[31] W. Stach, L. Kurgan, Modeling software development project using fuzzy cognitive maps, in: Proc. of the 4th ASERC Workshop on Quantitative and Soft Software Engineering, 2004, pp. 55–60.

[32] W. Stach, L. Kurgan, W. Pedrycz, Higher-order fuzzy cognitive maps, in: Proc. of the North American Fuzzy Information Processing Society Conf., 2006, pp. 166–171.

[33] W. Stach, L. Kurgan, W. Pedrycz, Parallel learning of large fuzzy cognitive maps, in: Proc. Internat. Joint Conf. on Neural Networks, 2007, pp. 1584–1589.

[34] W. Stach, L. Kurgan, W. Pedrycz, M. Reformat, Genetic learning of fuzzy cognitive maps, Fuzzy Sets and Systems 153 (3) (2005) 371–401.

[35] W. Stach, L. Kurgan, W. Pedrycz, M. Reformat, Learning fuzzy cognitive maps with required precision using genetic algorithm approach, Electronics Letters 40 (24) (2004) 1519–1520.

[36] W. Stach, L. Kurgan, W. Pedrycz, M. Reformat, Parallel fuzzy cognitive maps as a tool for modeling software development project, in: Proc. of the North American Fuzzy Information Processing Society Conf., 2004, pp. 28–33.

[37] W. Stach, L.A. Kurgan, W. Pedrycz, Data-driven nonlinear Hebbian learning method for fuzzy cognitive maps, in: Proc. World Congr. on Computational Intelligence, 2008.

[38] W. Stach, L.A. Kurgan, W. Pedrycz, Numerical and linguistic prediction of time series with the use of fuzzy cognitive maps, IEEE Transactions on Fuzzy Systems 16 (1) (2008) 61–72.

[39] M.A. Styblinski, B.D. Meyer, Signal flow graphs vs. fuzzy cognitive maps in application to qualitative circuit analysis, International Journal of Man-Machine Studies 35 (1991) 175–186.

[40] C.D. Stylios, P.P. Groumpos, Fuzzy cognitive maps: a model for intelligent supervisory control systems, Computers in Industry 39 (3) (1999) 229–238.

[41] C.D. Stylios, P.P. Groumpos, Modeling complex systems using fuzzy cognitive maps, IEEE Transactions on Systems, Man and Cybernetics Part A 34 (1) (2004) 155–162.

[43] Z. Wei, L. Lu, Z. Yanchun, Using fuzzy cognitive time maps for modeling and evaluating trust dynamics in the virtual enterprises, Expert Systems with Applications 35 (4) (2008) 1583–1592.

[44] G. Xirogiannis, M. Glykas, Fuzzy cognitive maps in business analysis and performance-driven change, IEEE Transactions on Engineering Management 51 (3) (2004) 334–351.