Automated Self-Assembly Programming Paradigm: The Impact of Network Topology Q1

Lin Li,* Jonathan M Garibaldi,† Natalio Krasnogor‡
ASAP Group, School of Computer Science and Information Technology, Univeristy of Nottingham, Nottingham, NG8 1BB, UK

In our previous work Li et al., in *Proc of 3rd IEEE Int Workshop on Engineering of Automatic & Automation Systems*, Potsdam, Germany, 2006, pp 25–34; Li et al., in *Proc Workshop on Nature Inspired Cooperative Strategies for Optimization*, Granada, Spain, 2006, pp 123–134, we introduced automated self-assembly programming paradigm ($ASAP^2$) using unguided self-assembly and swarm-inspired methodologies. We investigated how external environment settings affect software self-assembly speed and diversity of the generated programs. In this paper, we extend our previous work with a diversified compartments approach based on general graphs. This diversified compartments approach is integrated into a network structure such that each compartment can be seen as a node in the network. We investigate how structures of the network impacts on software self-assembly speed, complexity, and diversity of generated programs. Results indicate that network structure can substantially affect the dynamics, diversity, and complexity of generated programs.

1. INTRODUCTION

Self-assembly is a ubiquitous process in nature in which a disordered set of components autonomously assembles into a complex and more ordered structure. Components interact with each other without the presence of central control or external intervention. The information on how to assemble the final products is implicitly encoded on the way components interact with each other, and this interactions are embodied in the structures and properties of the individual components and the environment where they live in. Hence, the design of individual components and their environment are the key for successful control of self-assembling systems. One popular example of self-assembly is that of amphiphilic molecules. These are composed of two ends with opposite properties: the Hydrophilic heads tend to be close to small water molecules, whereas hydrophobic ends tend to repel water molecules and be close to similar chains. As is shown is Figure 1, the circle-shaped ends are

*Author to whom all correspondence should be addressed: e-mail: lxl@cs.nott.ac.uk. Q2
†e-mail: jmg@cs.nott.ac.uk.
‡e-mail: nxk@cs.nott.ac.uk.

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 00, 1–25 (2009)
 Published online in Wiley InterScience (www.interscience.wiley.com). • DOI 10.1002/int.20361

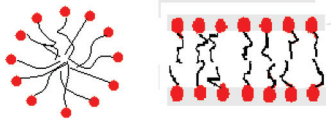


Figure 1. (a) Micelle structure and (b) bilayer structure formed by amphiphilic molecules when they are placed into water.

hydrophilic heads and the tails are hydrophobic heads. As a result, various structures can be self-assembled when amphiphilic molecules are placed into water, like for example lipid membrane, giant vesicles, micelles, bilayers, and so on.[1–4]

Q3

Self-assembly systems are regarded as being robust and versatile.[5] These two features come from the fact that self-assembly involves the utilization of a potentially large, perhaps simple, set of components where only some will be involved in constructing the final structure. Self-assembly systems are versatile because a given structure can be achieved using different configurations of components. Self-assembly systems are also considered to be robust because if a certain part of the system fails, other components can be used to replace the failed part as to ensure the functionality and integrity of the system as a whole.

Self-assembly plays an important role in nature not least in its purported role in the origin of life. There are strong arguments claiming that life originated from inanimate matter through a spontaneous and gradual increase of molecular complexity.[6] One such model that seeks to explain the origin of life is called *the compartmentalistic approach*. The main concept behind this approach is the fact that known life forms are based on cells, i.e., closed compartments that can keep inside a running metabolism and information polymers.[a] The main function of life can be seen as an interaction between the compartment and the external medium realized by the flux of information and material exchanged through the boundaries of compartments. The compartmentalistic approach is reinforced by the fact that molecules of prebiotic origin are thought to have self-assembled and formed cell-like compartments as the micelle shown in Figure 1. It has been argued[6–8] that cell-like compartments were the essential building blocks for early life forms. That is self-assembly, and more generally, self-organization was a prerequisite for evolution by natural selection.

Inspired by natural self-assembly and the compartmentalistic approach on the origin of life, we are taking a step back from the investigation of automatic program synthesis by evolutionary methods such as GP,[9,10] or grammatical evolution[11] to study the role that self-assembly could have in the automatic synthesis of program parse trees. Hence, rather than using an evolutionary approach, we intend to focus on software self-assembly instead. We focus on self-assembly for the automated construction of programs *structures* rather than evolution of programs *functionality*, which clearly is the ultimate and more difficult goal. In this paper we focus on generating diversity of structure as a precursor for novel functionality because it is widely believed in nature structure precedes function. Although genetic programming and other evolutionary methods are amongst the most popular methodologies

[a] A virus does not have metabolism.

for automated program synthesis[12] and having been applied to a wide range of problem domains (e.g., Refs. 13–16), we seek to answer whether self-assembly can provide any complementarities and insights to automated software synthesis. Thus our aim is not to replace genetic programming, but to complement it. Software self-assembly can be seen as a "bottom-up" manufacturing methodology as opposed to traditional software engineering techniques.

In our previous work, we presented automated self-assembly programming paradigm ($ASAP^2$). $ASAP^2$ is a software self-assembly system in which manually decomposed software components move and interact with each other in a confined space and eventually self-assemble into programs. The $ASAP^2$ could have an impact on our understanding of self-healing[17] and self-reconfigurable[18] software. $ASAP^2$ uses manually deconstructed manmade software rather than initializing the system with random software (partial) parse tress as usually done in GP because we are interested in understanding $ASAP^2$ based on real software, i.e. software as it is rather than as it could be. We previously introduced unguided software self-assembly in Ref. 19. We extended the model and introduced a PSO-inspired approach in Ref. 20 in an attempt to shorten the so-called "time to equilibrium." We investigated how different factors can affect the course of self-assembly and the diversity of the generated systems in both methodologies. This previous work is detailed in the following sections.

In this paper, we extend this work by introducing diversified population structures, that is, compartments and connections among them. The population structure is represented by embedding compartments in a graph in which software self-assembly takes place simultaneously in each compartment. Each vertex represents a compartment, and components can freely move to a neighboring compartment that is connected by an edge, thus enabling software components exchange between compartments. We systematically investigate a range of graph topologies covering simple reticular structures, to small world networks, to fully random ones. The idea behind this work is the compartmentalistic view of the origin of life, and we are interested in how different topologies and average interconnection distances within the network can have an influence (if any) on the software self-assembly process, along with the resulting complexity and diversity of the generated programs. In addition, a network structure allow us to abstract the "internet" and study $ASAP^2$ as a phenomena of mobile code in the Web/grid.

This paper is organized as follows. In Section 2, we introduce the $ASAP^2$ model and give brief review of our previous experimental results. In Section 3, we discuss the motivation of our current work and describe our extended system together with the graph models it is based on. We explain how the experiments are conducted and present results in Section 4. Finally, Section 5 presents conclusions and future work.

2. PREVIOUS WORK ON AUTOMATED SELF-ASSEMBLY PROGRAMMING PARADIGM

In Ref. 19, we presented $ASAP^2$, a software self-assembly system that automatically generated programs from a set of manually provided software components.

A program can be represented by a parsing tree, and a software component can be an arbitrary node on this parsing tree. Software components contained ports to play the role of binding sites. Ports can be divided into two classes with a corresponding data type associated: input ports and output ports. A software component can have only one input port but can contain an arbitrary number of output ports. Those components with no output ports are the leaf nodes in its parsing tree representation. We also assume that an input port can only connect to an output port with the same data type. This type constraint was designed to ensure valid and meaningful configurations.

In $ASAP^2$, software self-assembly starts by placing all software components retrieved from a given software repository into a pool within which they move randomly. When a component senses another component within a Euclidean distance d_δ and their types match, they self-assemble into a bigger aggregate structure. However, if no bindings can be formed, the two components will repel each other. This notion is formalized in Algorithm 1.

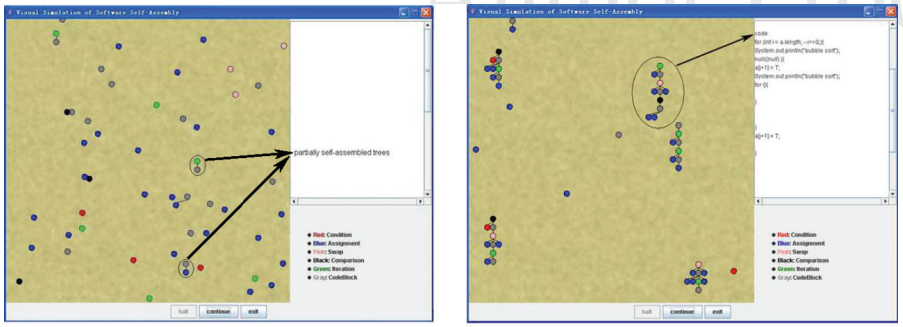
Algorithm 1 binding algorithm: d_δ is a threshold distance for components to bind, and μ_i indicates displacement of component c_i

```
1:  for (every pair of components c1 and c2 in the component set S_A) do
2:    if (distance (c1, c2) ≤ d_δ) then
3:      success = attemptBind(c1, c2)
4:      if (success == true) then
5:        add assembled component to the set S_A and delete c1, c2.
6:      else
7:        c1.move(currentLocation, μ_1)
8:        c2.move(currentLocation, μ_2)
9:      end if
10:   end if
11: end for
```

Figure 2a shows an early stage of software self-assembly in which simple partial structures have assembled. Figure 2b shows a latter stage in which more complex program trees have emerged. The simulation was deemed to have reached an *equilibrium* when no more binding actions could occur between the remaining components in the pool.

Particle swarm intelligence (PSO) is a population-based search algorithm in which each individual represents a solution.[21] PSO is similar to software self-assembly in a way that each individual in the population is an autonomous agent that has a velocity and follows simple rules. However, high-level behaviors emerge by those agents as a group as in real particle swarm. In Refs. 22–24, components follow leaders that exhibit higher fitness value, and particles determine and change their neighbors dynamically according to the distances in each generation.

In Ref. 20, we extended our work[19] by introducing dynamic leader and neighborhood during a software self-assembly process. When components were first



(a) (b)

Figure 2. (a) Early stage of software self-assembly. (b) Latter stage of software self-assembly.

placed into the pool, leaders are randomly selected from the component set based on a fixed proportion. If a nonleading component was within a distance threshold (D_α) to a leader, it followed the leader component. Leader components or those components that do not have a leader perform Brownian motion as they do in Ref. 19 until a leader appears in its neighborhood. When a certain component binds with a leader component, the self-assembled structure also becomes a leader.

If more than one leader is within the threshold range (d_α) of a nonleading component, there needs to be a way to decide which leader to follow. To do that, we introduced attractive force of a leader component. Nonleader components followed a leader component containing the greatest attractive force. In this way, a nonleader component may change its leader dynamically in a self-assembly process. The objective was to shorten the time to equilibrium for the system and observe how diversity of the generated populations was affected. Attractive force of a leader component was determined by the number of available ports it contained and the distance to its following component. Hence if a component found two identical leader components within D_α, it followed the closer one.

2.1. Summary of Previous Results

We were interested in seeing whether we could control the result of software self-assembly in terms of time to equilibrium and diversity of generated programs at equilibrium. Time to equilibrium (t_ε) records how long it takes the system to reach equilibrium. D_ε is used to assess the diversity of the emergent parse trees. For measuring diversity we recorded the total number of assembled tree classes, we considered that two trees belong to the same parse tree class if both their structures and content are identical.[25] We introduced the following environment parameters to manipulate the behavior of individual components and the environment they are living in temperature (T), area (A), and number of copies (N) placed into the pool. These environment parameters affect the software self-assembly process in the following ways: (1) Software components move faster with a higher temperature.

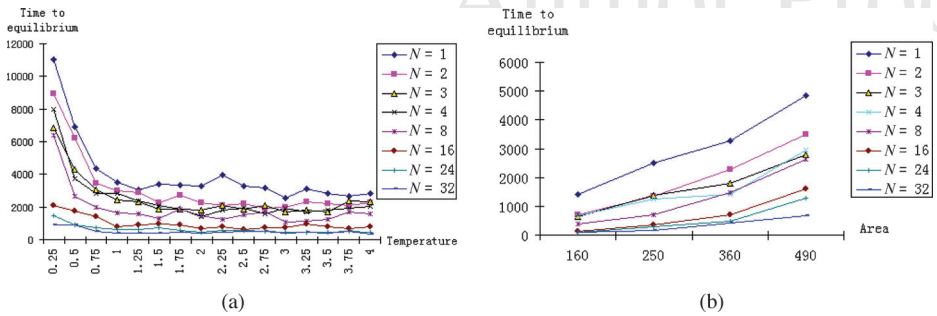


Figure 3. Relationship between (a) time to equilibrium and temperature ($A = 3.6 \times 10^5$) and (b) time to equilibrium and area ($T = 2.0$) with different number of copies of components.

(2) A pool with a smaller size exhibits a greater density. (3) With more copies of components placed into the search pool, greater diversity can be achieved. However, it is important to note that the diversity of the generated programs is not only a function of the total number of components but it is also dependent on their relative concentrations. That is, D_ε will be lower from a system starting with a large number of similar components than one with same number of greatly diversed components.

Experimental results shown in Figures 3a and 3b illustrate how t_ε was affected by area (A), temperature (T), respectively, with unguided dynamics. It can be seen that time to equilibrium decreases as temperature increases. This is because as components move faster to explore more areas in the pool, it is more likely for them to form valid bindings. Second, it takes longer for the system to reach equilibrium with a larger pool because the level of concentration in the pool decreases. Moreover, an interesting point to note here is that with more components in the pool, it takes less time for the system to reach equilibrium. This is because when there are more components in a fixed confined space, it is easier for components to find another component to bind with.

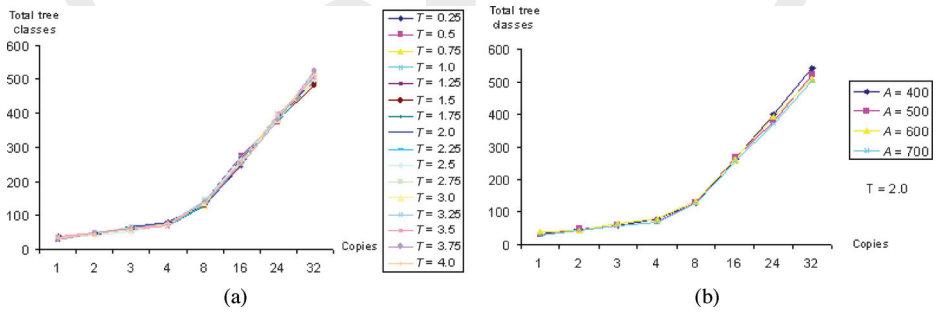


Figure 4. Using unguided self-assembly: (a) relationship between total different tree classes and number of copies ($A = 3.6 \cdot 10^5$). (b) relationship between total number of different tree classes and number of copies ($T = 2.0$).

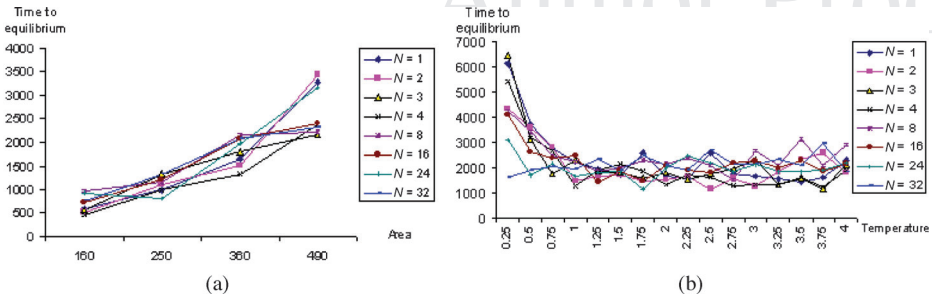


Figure 5. Relationship between (a) time to equilibrium and temperature ($A = 3.6\dot{1}0^5$) (b) time to equilibrium and area ($T = 2.0$) with different number of copies of components.

The experimental results from Ref. 19 in Figure 4 illustrate that the number of copies of components placed into the pool is the primary factor that influence the diversity of generated programs. As more components are placed into the pool, self-assembly system yields more diversity of the generated programs.

In Ref. 20, PSO-based methodology was used to guide the process of software self-assembly, rather than the unguided dynamics used in Ref. 19. The same set of experiments in Ref. 20 was performed as in Ref. 19. Figure 5 shows a faster time to equilibrium is achieved. However, temperature (T) and area (A) no longer played an important role in t_ε. The presence of leaders resulted in a less diversified population as shown in Fig. 6. It can be seen, however, that D_ε was influenced by the same factors in similar ways. Q4

2.2. Prediction Models

We presented prediction models in Ref. 20 to interpolate t_ε and D_ε with (A, T, N) for software self-assembly with unguided dynamics. The following formulae were produced to predict t_ε having one of the three free environment parameters fixed to a prespecified value ($T = 2.0$, $N = 8$, $A = 360$, respectively):

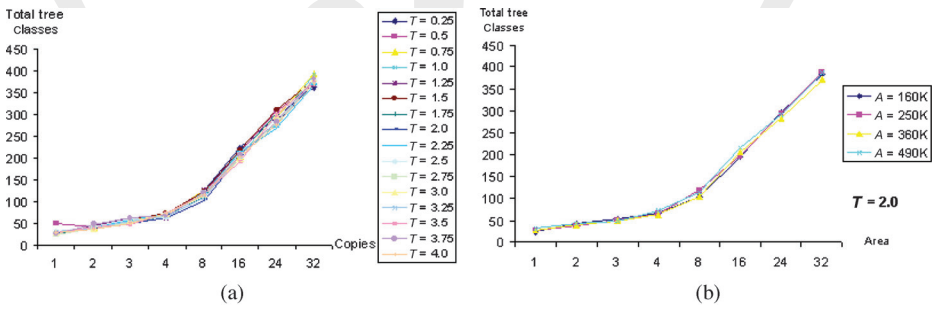
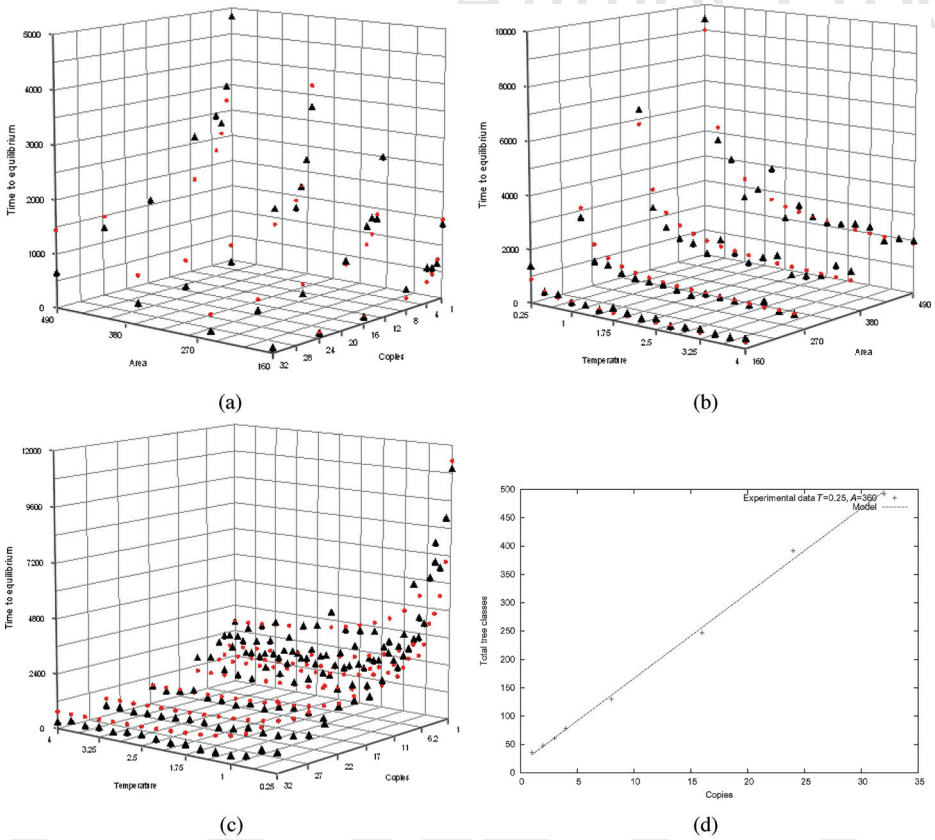


Figure 6. Software self-assembly with leaders: (a) relationship between total tree classes and number of copies in different temperature settings; (b) relationship between total tree classes and number of copies in different area settings.



(a) (b) (c) (d)

Figure 7. Prediction model assessment on (a) time to equilibrium with $T = 2.0$ (b) time to equilibrium with N = 8 (c) time to equilibrium with $A = 3.6 \cdot 10^5$ (d) diversity of the generated programs with $A = 3.6 \cdot 10^5$, $T = 0.25$. Triangles (▲) represent experimental data, and circles (●) represent the corresponding data obtained from our predictive model.

As has been shown in Figure 4, diversity of self-assembled programs was not mainly affected by temperature neither by area of the pool. Hence, we concluded Equation 4 to predict program diversity in relation to number of copies of components only.

$$t_\varepsilon(A, N) = \frac{(7.05 \times A) + 321.2}{N} + (3.91 \times A) - 593.71 \quad (1)$$

$$t_\varepsilon(A, T) = \frac{(5.21672 \times A) - 659.015}{T} + (3.7274 \times A) - 416.114 \quad (2)$$

$$t_\varepsilon(T, N) = \left(\frac{1}{T} + 1\right) \times \left(\frac{1726}{N} + 637.325\right) \quad (3)$$

$$D_\varepsilon(N) = 15 \times N + 16.6 \quad (4)$$

Table I. Error statistics for predictive model on *ASAP*^{2}.

Equation	Category	Average error rate	Standard deviation
1	A = 160	0.208	0.148
	A = 250	0.425	0.525
	A = 360	0.389	0.473
	A = 490	0.261	0.374
2	A = 160	0.242	0.145
	A = 250	0.152	0.146
	A = 360	0.192	0.098
	A = 490	0.097	0.090
3	N = 1	0.106	0.106
	N = 2	0.129	0.089
	N = 3	0.181	0.108
	N = 4	0.263	0.095
	N = 8	0.224	0.144
	N = 16	0.219	0.189
	N = 24	0.733	0.207
	N = 32	1.190	0.521
4	N/A	0.039	0.010

Figure 7 shows the comparison between the predicted results and the obtained experimental data, in which Figures 7a, 7b, 7c, and 7d correspond to Equations 1, 2, 3 and 4, respectively. The comparison suggests that the predictive model was fairly accurate as experimental data follow the same trend as predicted by the formulae. To the best of our knowledge, similar predictive capacity has not yet been achieved in GP.

With a lower area setting in Equations 1 and 2, the equations predicted more accurately with a smaller average error rate. Table I also shows a smaller average error rate for the predicted D_ε using Equation 4, which means the prediction agreed quite well with the experimental results.

3. THE IMPACT OF TOPOLOGICAL NETWORKS

3.1. Motivation

The compartmentalistic theory on the origin of life argues that compartment structures form spontaneously through self-assembly processes, and perhaps provided the original membrane-bounded environment required for cellular life to begin.[7,8] The importance of compartment structure lies in the fact that information and materials can be exchanged through its border. Although the previous work on *ASAP*^{2} resembles this compartmentalistic approach such that self-assembly takes place in a confined area, no information exchange occurs from the inside to the outside of compartments. In this paper, we include this important aspect of compartmentalistic approach by introducing a general graph structure of compartments within which software self-assembly takes place. In this graph structure, each vertex represents a compartment and software components at the border of a compartment have the choice to move to another compartment through a connecting edge.

The graph model we use for our system is called β-graph, which is originated in one of the graph models that Watts[26] proposed to analyze small world phenomena. The question Watts tries to answer can be briefly explained as: What are the most general conditions under which the elements of a large, sparsely connect network will be "close" to each other. The closeness of vertices is determined by the length property of the graph, which has been an active research area and been studied on different problem classes, for example, the performance of computer networks,[27] telecommunication network,[28] etc.[29] *Characteristic path length* (abbreviated as CPL in the remaining of this paper) is one of the most important statistics used to measure the *shortest* distance between each vertex (i, j) in a graph. The formal definition of CPL is given in Ref. 26 as "The characteristic path length (CPL) of a graph (G) is the median of the means of the shortest path lengths connecting each vertex $v \in V(G)$ to all other vertices. That is, calculate $d(v, j) \forall V(G)$ and find $\bar{d}_v$ for each v. Then define L as the median of $\{\bar{d}_v\}$."

Based on β-graphs, we seek to answer what impact, if any, closeness between compartments and the neighborhood structure of the graph have on the process and results of software self-assembly. The length properties of the graph is measured by CPL, and the neighborhood structure is measured by the *clustering coefficient* (abbreviated as CC in the remaining of this paper) of the graph, the definition of which is given in Ref. 26 as "The CC γ_v of Γ_v characterises the extent to which vertices adjacent to any vertex v are adjacent to each other. More specifically,

$$\gamma_v = \frac{|E(\Gamma_v)|}{\binom{k_v}{2}}$$

where $|E(\Gamma_v)|$ is the number of edges in the neighbourhood of v and $\binom{k_v}{2}$ is the total number of *possible* edges. The CC of G is $\gamma = \gamma_v$ averaged over all $v \in V(G)$."

Solé[30] has investigated self-organized network traffic flow in a simple lattice architecture. Our work is also inspired by this approach in which the graph topology is used as a model of the network. The general graph model that we deploy for *ASAP*[2] can also be regarded as representatives of various network topologies on a LAN, the Internet or grid infrastructure, where each vertex in the graph is an individual computer and the edges represent direct telecommunication links. Hence software self-assembly can happen anywhere in the network.

Farley[31] presented a similar approach with the goal of investigating how fitness and diversity of genes within an evolutionary algorithm can be affected by different population structure. Farley uses specific graphs (i.e., string, ring, and various tree topology) as population structures. Moreover, each individual resides at a vertex of the graph and could only choose a mating partner from among its neighbors in the graph.

3.2. Our Implementation

We use β-graphs to study software self-assembly based on the following restrictions on the generated graphs. First, to study meaningful configurations, the graph is assumed to be *simple*, meaning that multiple edges between the same pair

of vertices or edges connecting a vertex to itself are forbidden. Second, the graph has to be *connected* such that a unique global equilibrium will eventually be reached for software self-assembly. Furthermore, software components can flow in and out of any vertex in a graph. Hence, the graph is assumed to be *undirected*. Finally, the graph is *unweighted* because we are merely taking into account the relation between vertices on the graphs and the connections themselves, while ignoring constraints such as distances between vertices or link capacity.

β-graphs capture a variety of network topologies from a highly ordered to a completely random graph. Three parameters are used to define the properties of graphs generated under the β-graph model.

- v: the number of vertex in the graph. Each vertex represents a compartment.
- k: determines how many initial nearest neighbors each vertex has.
- β: a probability value determining the "rewiring rate."

With the parameters mentioned above defining the properties of graphs, β-model starts with a perfect ring structure, in which each vertex has precisely k-neighbors ($k/2$ on either side, and hence we restrict k to be an even number). Given a prespecified probability value β, the algorithm then *randomly rewires* the edges of the ring using the following algorithm: (1) Each vertex i is chosen in turn, along with the edge that connects it to its nearest neighbor in a clockwise fashion. (2) A random deviator γ is generated. If $\gamma < \beta$, then $(i, i + 1)$ is deleted and *rewired* such that i is connected to another vertex j, which is chosen *randomly* from the entire graph (excluding self-connection and repeated connections). Otherwise, the connection is unaltered. (3) After all vertices have been considered once, the procedure is repeated for edges that connects each vertex to its *next nearest neighbor*, i.e., $(i, i + 2)$, until all edges have been considered for rewiring once.

Figure 8 shows some β-graphs constructed in this fashion with $v = 20, k = 4$, and $\beta \in \{0.0, 0.2, 0.6, 1.0\}$. On one extreme, β is set to 0 and the original structure is unaltered. Hence a highly ordered structure is constructed as shown in Figure 8a. On the other extreme, with $\beta = 1.0$, a stochastic graph with all edges rewired is produced as shown in Figure 8d. Hence, β-graphs can be gradually transformed from ordered graphs to random graphs with increasing β value. As Figure 8 suggests, the properties of randomness and order is controlled by this single parameter β.

We use unguided dynamics in this work so that components move randomly in each pool and interact with each other as in Ref. 19. That is, software components perform Brownian motion in the compartment where they "live" in. Our previous work has suggested that the environment has more obvious impacts on software self-assembly with unguided dynamics than with swarm-based methodology. Although software self-assembly with leaders improves the speed to reach equilibrium to some extent, we are more interested in how the graph structure and properties can have impacts on complexity and diversity of software self-assembly generated programs, which are more favored by unguided rather than swarm realizations.

With the generated β-graphs, software components are distributed evenly in each compartment. If two compartments are directly connected via an edge, this pair of compartments is identified as *neighboring* compartments. When a component hits

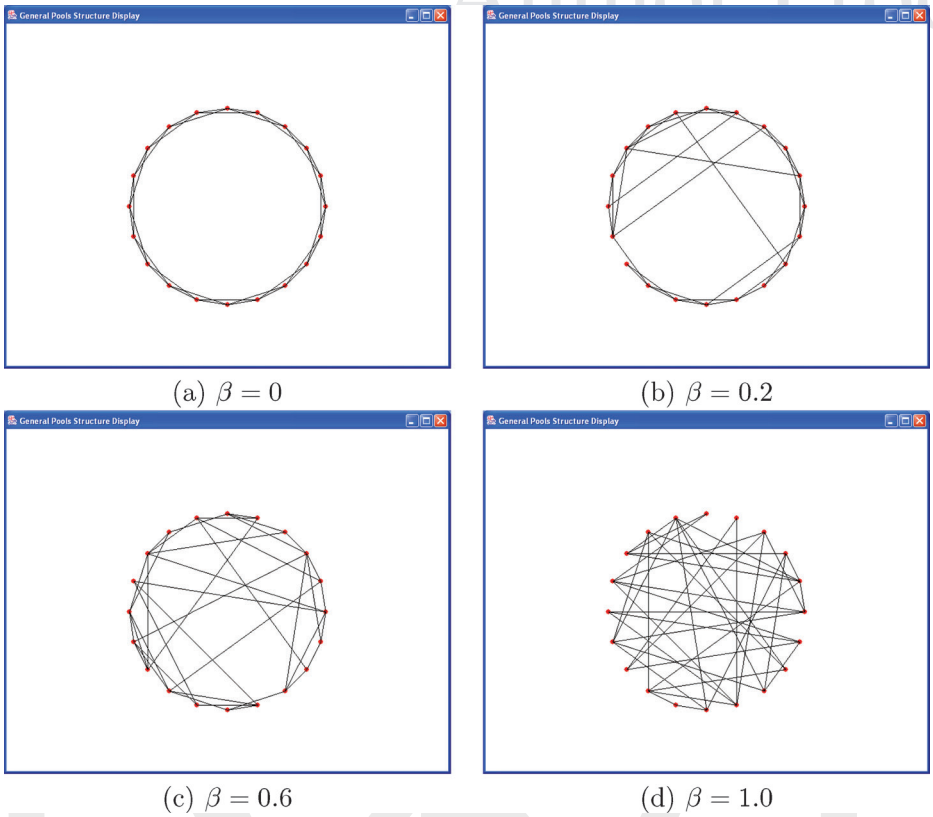


Figure 8. Exemplar β-graphs with $v = 20$, $k = 4$, $\beta \in \{0.0, 0.2, 0.6, 1.0\}$.

a border, it will be randomly transferred to one of its neighboring compartment with equal probability. The simulation finishes when there are no more possible binding actions between the remaining self-assembled trees *in the whole graph network*. That is, global equilibrium on the network rather than local equilibrium of each compartment is the terminating condition for our simulations.[b]

The compartmentalistic approach introduces local gradients of software components concentrations. As studied in previous work,[19,20] a smaller compartment might exhibit a greater concentrations of key components than a larger compartment. We introduce equal probability for components to migrate into a neighbor compartment and all compartments in the network have equal size.

3.3. Graph Properties

Watt investigated in Ref. 26 how length and clustering properties, i.e., CPL and CC, are affected by β under fixed k and v value. Figure 9 illustrates the relationship

[b]Please note that global equilibrium of the complete graph implies equilibrium at local level.

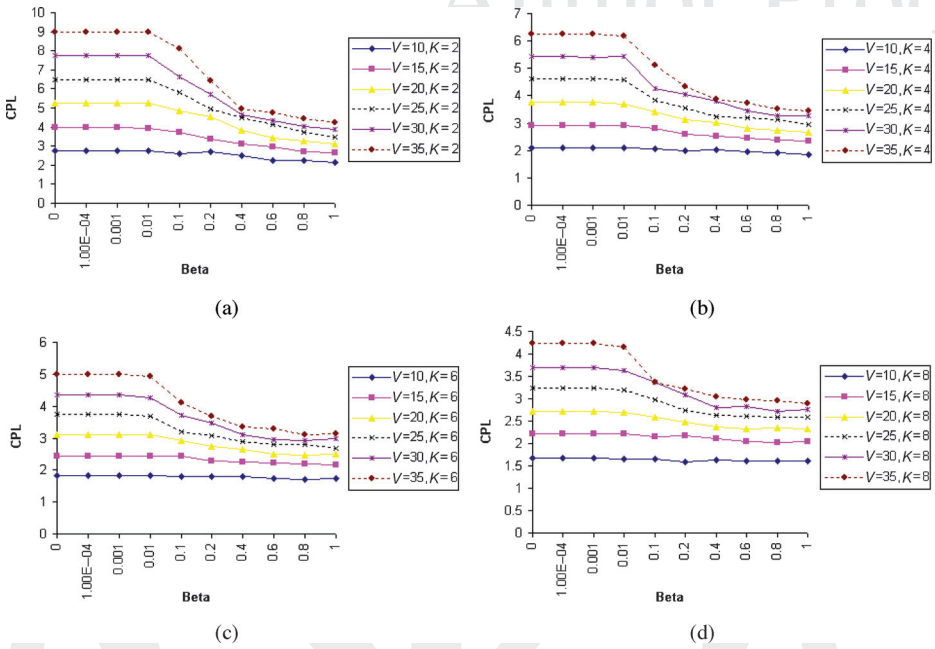


Figure 9. Relationship between CPL and β, k with (a) $k = 2$, (b) $k = 4$, (c) $k = 6$, and (d) $k = 8$.

that exists between CPL of a graph and the parameters (β, k, v) used to construct it. It can be seen from the figures that CPL of a graph decreases with a higher β value. Hence, a graph with more ordered structure results in a greater average length than a random graph does. The transition of CPL value occurs when β is greater than 0.01 regardless of k. In addition, the decrease in CPL value against β becomes more obvious with a larger graph, i.e. a greater v value. Moreover, CPL increases with a larger graph as Figure 9 suggests. Finally, CPL decreases when k value increases. The reason is rather straightforward, a graph with more vertices connected before β-model rewires connections exhibit shorter average length property of the graph.

Hence, it can be concluded that in general, a β-graph starting with higher k, β, and lower v value will result in a graph with lower average graph length. On the contrary, a graph with high CPL value has a greatest probability of being the result of a low k, β, or high v value.

Figure 10 illustrates the relationship between CC and the three graph parameters (β, k, v). It can be seen that CC behave dramatically different under different k. When $k \in 4, 6, 8$, it can be seen that the CC decreases as β increases. The decrease in CC becomes more significant and obvious with a higher v. In addition, CC increases as k increases. This means when $k \in 4, 6, 8$, a β-graph starting with a higher k, v and β value is more clustered than those constructed with a lower k, v and β value. On the contrary, when k equals to 2, CC increases as β increases, and number of vertices in the graph does not have an obvious impact on CC. The relationship between β

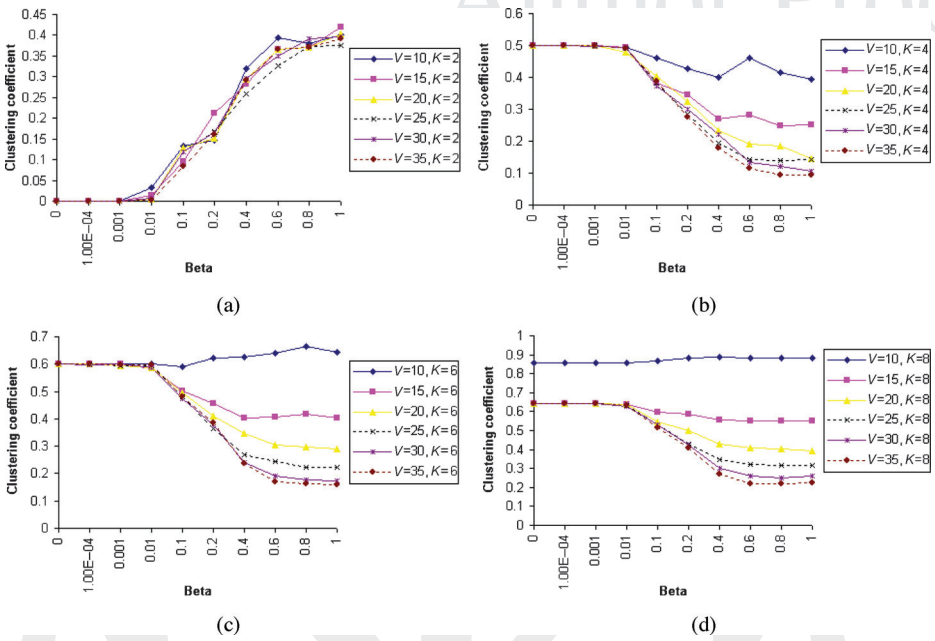


Figure 10. Relationship between CC and β, k with (a) $k = 2$, (b) $k = 4$, (c) $k = 6$, (d) $k = 8$.

and CC is shown as a parabola under $k = 2$. Hence, when k is set to 2, a β-graph constructed under high β value is more clustered than those constructed under a low β value.

4. EXPERIMENT

4.1. Methods

Based on the knowledge of how (β, k, v) influence the characteristic path length and clustering properties of a graph, we aim to find out the relationship between CPL, CC, number of components (N), time to equilibrium (t_ϵ), diversity (D_ϵ), and complexity of the generated programs at equilibrium.

Our previous work has shown how external environment parameters can influence the software self-assembly process. Because we are working on the graph topology and its influence on software self-assembly, we fix the previously introduced parameter as temperature (T) to be 2.25, as this is the median temperature value we used in our previous work.[19,20] The total area (A) is 5000 $\times$ 5000, which is 10 times the size we used before. The total area (A) in our simulation is constant regardless of the number of compartments. The size of each compartment (A_i) is inversely proportional to the total number of compartments involved in the graph, i.e., $A_i = A/v$. Hence, the size of each compartment decreases and the software components concentration in the compartment could increase with more compartments

involved added to the network. However, note that for the V values used in our experiments the size of A_i is never smaller than that used in Refs. 19 and 20.

Since a particular β-graph is constructed by rewiring edges in a stochastic fashion, each (β, k, v) triplet represents a *family* of graphs for which an average CPL and CC must be computed. We construct five graphs in each (β, k, v) triplet, with different number of copies (N) placed into each graph and run 20 replicas. The ranges for k, v, N are $k \in \{2, 4, 6, 8\}$, $v \in \{10, 15, 20, 25, 30, 35\}$, and $N \in \{10, 25, 40, 55, 70, 85, 100\}$. As Watts[26] suggests, a significant transition in CPL and CC occurs when β ranges from 0.0001 to 0.1. Hence, we set β in a detailed full scale, i.e., $\beta \in \{0, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0\}$. That is, we run a total of 151,200 experiments. The average of time to equilibrium, diversity, and complexity is calculated for each run. The complexity of a program is measured by the height and number of nodes contained in its tree representation.

4.2. Results

4.2.1. Time to Equilibrium Analysis

Figures 11 and 12 show for different number of copies of components how time to equilibrium (t_ε) is influenced by CPL and CC, respectively. Most β-graph constructed have CPL value less than 4 and CC ranging from 0.2 to 0.6 as indicated in Figures 11 and 12. We wish to see how software self-assembly with multiple compartment structures differs from software self-assembly with single compartment. To do that, we compared the experimental results with predicted t_ε using Equations 1 and 2 for a single compartment structure. Each prediction data uses the same parameter settings $\{\beta, k, v\}$ as the corresponding experimental data. In one case, we set the area (A) in Equations 1 and 2 to be the total area of the compartments in the network. The predicted results are shown as empty square and empty diamonds for Equations 1 and 2, respectively. Figure 11 illustrates the predictive outcome differ slightly between Equations 1 and 2. Moreover, as CPL increases, t_ε gradually becomes lower than the corresponding predicted results.

In another case, we set the area (A) deployed in Equations 1 and 2 to be the size of each subcompartments in the network, that is $A_i = A/V$. The predicted results from Equations 1 and 2 are indicated in Figure 11 as black square and black diamonds, respectively. It can be seen that the predicted results of Equations 1 and 2 are similar. The corresponding scattered points are situated at the bottom of the figures, showing a faster time to equilibrium is expected using a single compartment structure with $(A_i = A/V)$.

Comparing across panels in Figure 11, we can see that t_ε decreases significantly from 1.2×10^6 to 3.5×10^5 as more components are placed into the network. This behavior is similar to the single compartment approach.[19]

Figures 11a and 11b also show that for $N = 10$, $N = 25$, t_ϵ increases to maximum when CPL approaches 4, and then decreases for values greater than 4. On the other hand, Figures 11c–11g indicate that t_ϵ on average decreases with larger CPL. The decrease on t_ε is counterintuitive as it suggests that a network with longer average path length results in faster software self-assembly. This phenomenon is

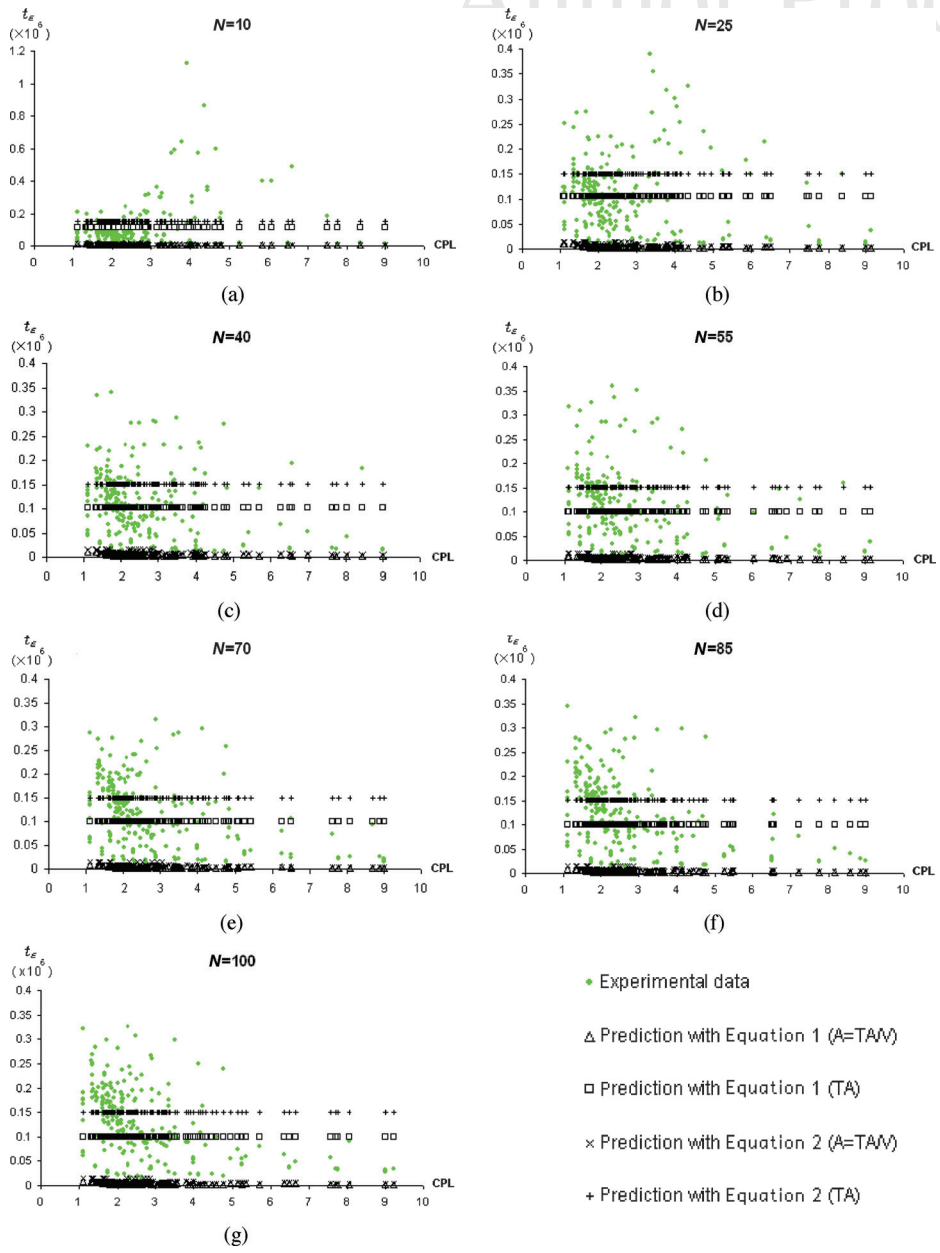


Figure 11. Relation between time to equilibrium and CPL for runs with (a) $N = 10$, (b) $N = 25$, (c) $N = 40$, (d) $N = 55$, (e) $N = 70$, (f) $N = 85$, and (g) $N = 100$. Comparisons are made between the experimental data and predicted time to equilibrium using Equation 1 (squares) and Equation 2 (circles) under the same environment parameters of the experimental data, i.e., using the same temperature ($T = 2.25$) and area ($A = 5000 \times 5000$, $Ai = 5000 \times 5000/V$).

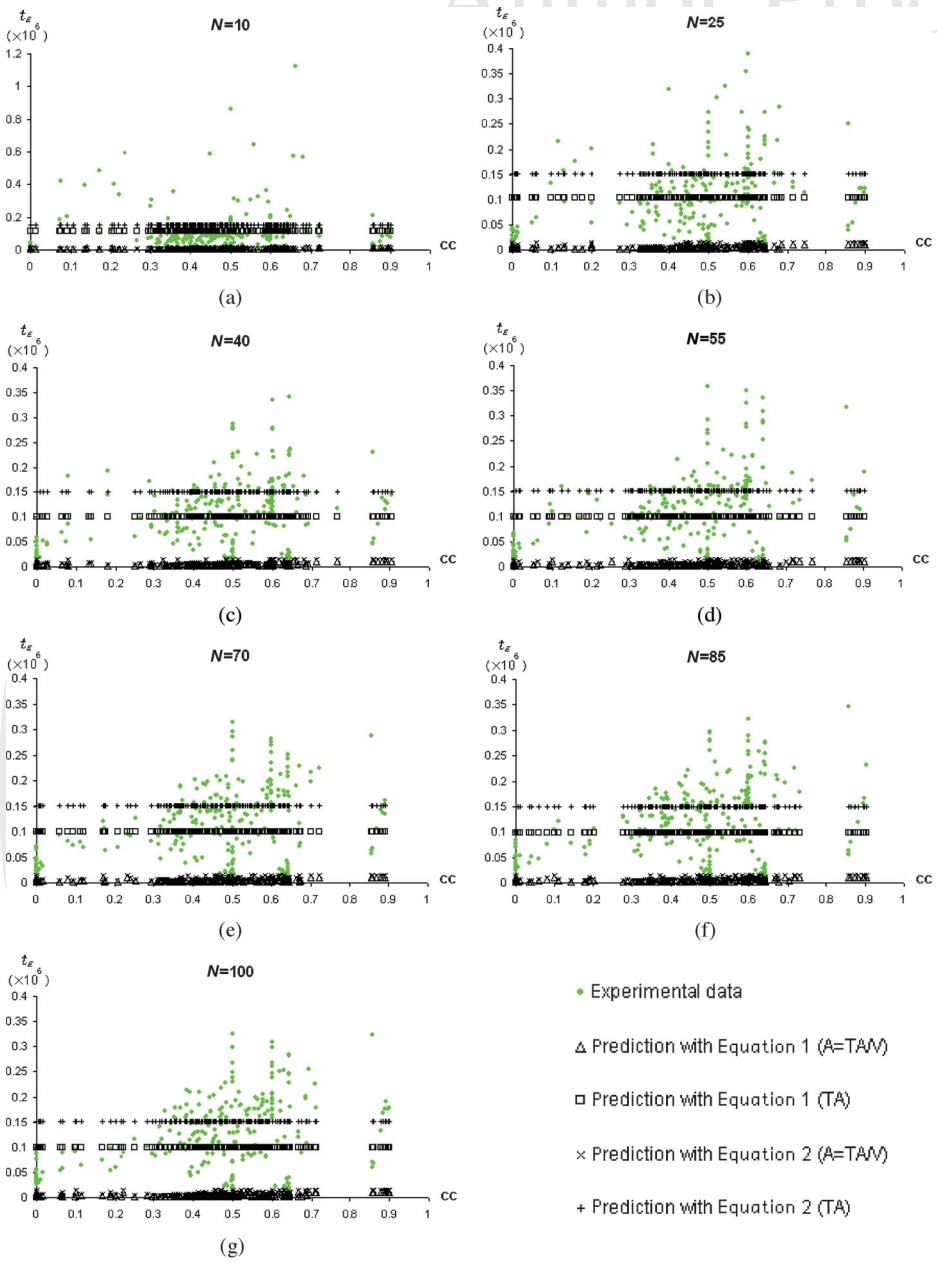


Figure 12. Relation between time to equilibrium and CC for runs with (a) $N = 10$, (b) $N = 25$, (c) $N = 40$, (d) $N = 55$, (e) $N = 70$, (f) $N = 85$, and (g) $N = 100$. Comparisons are made between the experimental data and predicted time to equilibrium using Equation 1 (squares) and Equation 2 (circles) under the same environment parameters of the experimental data as before.

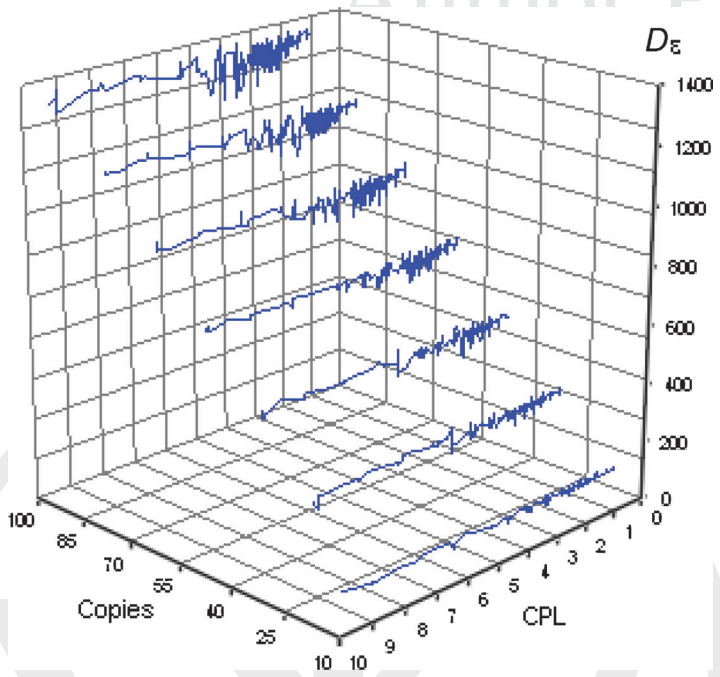


Figure 13. Relationship between D_ε and number of copies and CPL.

possibly due to the decrease in the size of the subcompartments in the network. As the total area is fixed to 5000×5000 in our experiments, a larger network with greater V will have smaller individual compartment size and hence have a higher pressure and concentration. As the results in Ref. 18 suggests, higher pressure leads to a smaller t_ε. Hence, software components find possible binding components faster in local compartments.

Figure 12 shows how t_ε is influenced by the clustering features of the graphs. It can be seen that t_ε increases to maximum while CC approaches 0.5 regardless of N. In addition, Figure 12 suggests that t_ε is higher than the predicted results with area equals to the size of subcompartments, i.e., $A_i = A/V$. However, when area deployed in Equations 1 and 2 is the size of the total area in the network, t_ε gradually increases and eventually exceeds the predicted data as CC increases from 0 to 0.6.

4.2.2. Diversity Analysis

Figure 13 illustrates how CPL and number of copies of components affect the diversity of the generated programs. As can be seen, the total number of assembled tree classes rises with more copies of components placed into the system. In addition, fluctuations occur in the number of total distinct assembled tree classes when CPL value ranges from 2 to 4, and the fluctuation is more obvious with a greater N.

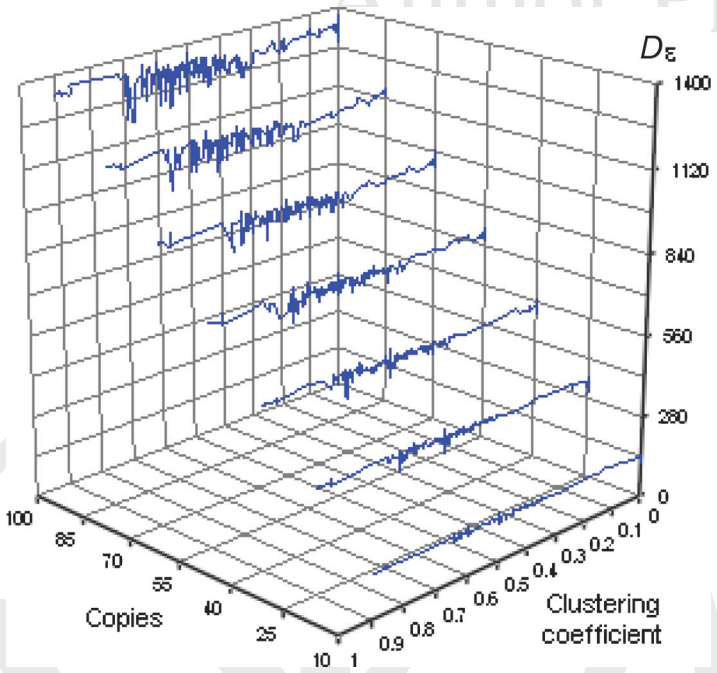


Figure 14. Relationship between D_ε and number of copies and CC.

Figure 14 shows how CC and number of copies of components affect the diversity of the generated programs. The fluctuations in the number of total assembled trees occur under all number of copies of components. However, under a greater number of copies, the fluctuations become more obvious (when CC ranges from 0.5 to 0.8) than under a small number of copies.

4.2.3. Analysis of Emerging Complexity

As has been mentioned, complexity of a program tree is measured by its height, h_ε, and number of nodes it contains, n_ε, at equilibrium. Figure 15 shows the histograms of number of assembled trees versus complexity bins for different number of copies. For each copy number, we differentiate graphs based on their CPL and CC. In Figure 15, we can see that there is an exponentially larger number of simple, small trees than complex and large ones. Figure 15 shows that in general a smaller CPL value results in a greater n_ε. Therefore, a sparsely connected network structure yields less complex programs than a densely connected network. Moreover, with a larger number of copies placed into the system, the rate of growth on n_ε increases; n_ε grows linearly with the increase in the number of copies. This result matches the diversity analysis, where the total number of distinct assembled trees increases with number of copies in a linear way. Figure 16 illustrates that h_ε behaves similarly yet not identical to n_ε. The only difference is that the rate of growth in number of assembled trees is greater with h_ε than with n_ε.

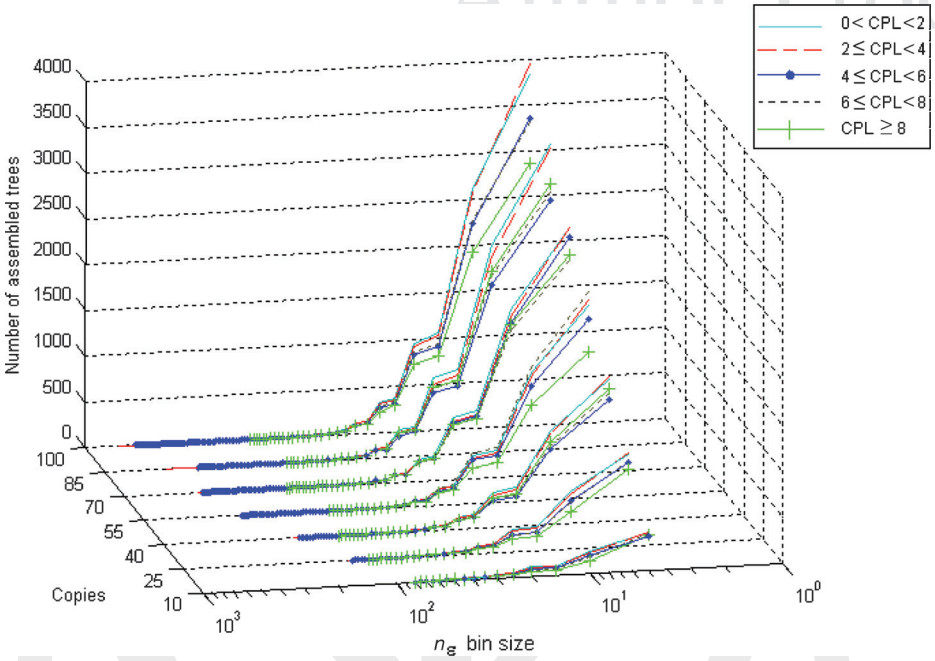


Figure 15. Histogram of average number of trees assembled in different nodes bin size and in different CPL categories.

Figures 17 and 18 show the histograms for generated programs as a function of CC of the network along with number of copies of components. The figures suggest that the number of simple and small trees is exponentially larger than the number of complex and large ones. In addition, it can be seen that the smallest and greatest CC value results in the greatest complexity (n_ε and h_ε).

4.2.4. ANOVA Analysis

We perform an ANOVA analysis to assess whether the number of assembled trees varies significantly for different CC and CPL. This was done for each copy number. The computed p-value from the ANOVA test is the probability that the variation between groups may have occurred by chance, hence a smaller p-value indicates a more significant difference. By convention, p-values below 0.05 are considered to be statistically significant. Table II illustrates the p-values grouped by CPL and CC under different number of copies of components (N). p-values are close to 1 for a small number of copies, and as N increases, p-value decreases. This means that the difference between n_ε (h_ε) in different CPL or CC becomes more obvious with a larger N. Moreover, Table II shows that CC p-values become significant earlier, i.e., with smaller N, than CPL p-values. This indicates that CC is perhaps a more important factor for software self-assembly in terms of the complexity of the self-assembled programs.

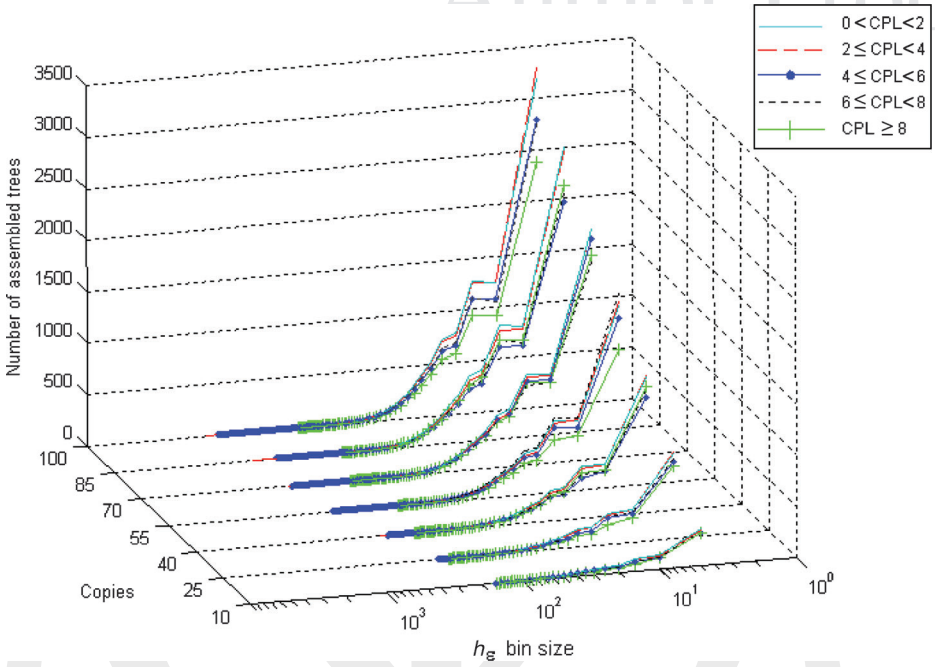


Figure 16. Histogram of average number of trees assembled in different tree height bin size and in different CPL categories.

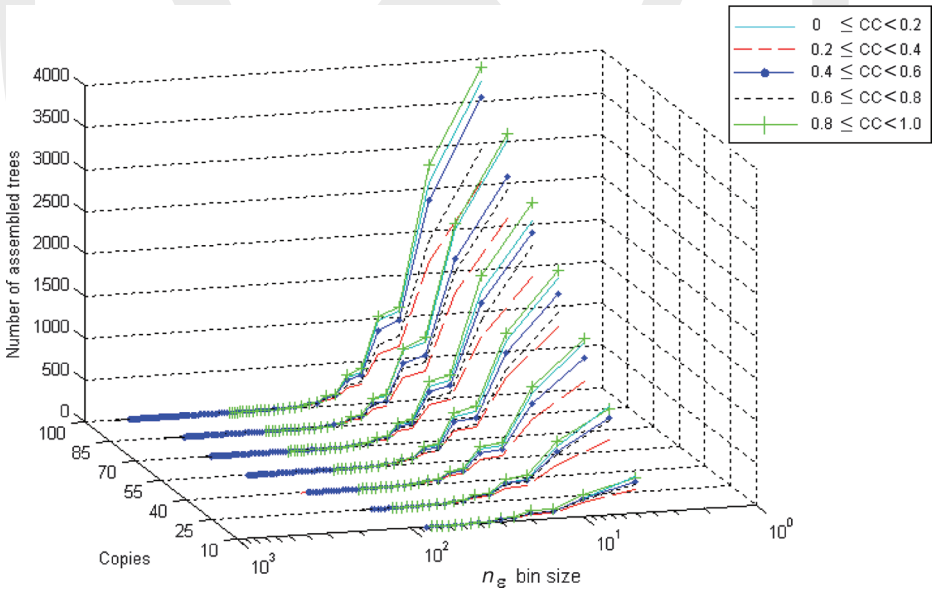


Figure 17. Histogram of average number of trees assembled in different nodes bin size and in different CC range.

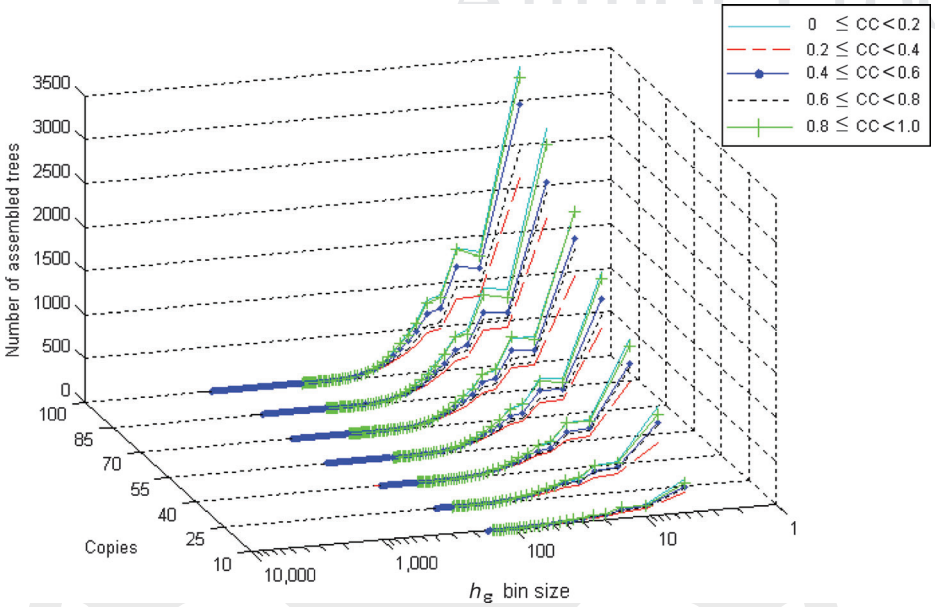


Figure 18. Histogram of average number of trees assembled in different tree height bin size and in different CC range.

Table II. ANOVA analysis for n_ε and h_ε under different number of copies. The figures in bold are those p-values close to or smaller than the threshold.

	$N = 10$	$N = 25$	$N = 40$	$N = 55$	$N = 70$	$N = 85$	$N = 100$
n_ε grouped by CPL	0.99287	0.98592	0.80157	0.3715	0.32458	0.24854	0.05604
h_ε grouped by CPL	0.98344	0.9581	0.59761	0.14386	0.13001	0.05601	**0.00697**
n_ε grouped by CC	0.97513	0.82167	0.35649	0.0576	**0.04857**	**0.02995**	**0.0139**
h_ε grouped by CC	0.93468	0.651	0.12889	**0.0451**	**0.03301**	**0.00116**	$\mathbf{3.2 \cdot 10^{-5}}$

5. CONCLUSION AND FUTURE WORK

In our previous work in $ASAP^2$, unguided dynamics and PSO-driven approach was presented for software self-assembly simulations within unstructured compartments. In this paper, we extend our previous work and introduced software self-assembly using diversified compartment structures based on β-graphs, a graph theoretic model that uses one parameter to control the complexity of the graphs ranging from highly ordered to totally random. The graph constructed by β-model represents a software self-assembly network system, in which each vertex in the network can be seen as a compartment where software components self-assemble locally. Components can migrate into another compartment by a connecting edge in the graph.

5.1. Conclusions

We overviewed how length and clustering property of a graph, measured by the characteristic path length (CPL) and clustering coefficient (CC) of the graph, are affected by the β-graph parameters (β, k, v). Based on the knowledge of the relationship between (β, k, v) and CPL, and CC, we investigated how length and clustering properties of a graph and the amount of software components involved can affect the process of software self-assembly within a network, we measured: time to equilibrium, diversity, and complexity of the generated programs. We report the experiments conducted on the extended system, which have shown that complexity of self-assembled programs rise with more copies of components.

Experimental results also show a counterintuitive behavior: On average, a higher CPL of the graph leads to a lower time to equilibrium. Furthermore, time to equilibrium decreases with more number of components, which matches our previous results based on $ASAP^2$ with single compartment structure. Moreover, regardless of number of components placed into the system, a fluctuation in diversity of generated programs is observed when CPL ranges from 2 to 4. The fluctuation in total number of distinct assembled trees is also observed when CC ranges from 0.5 to 0.8.

A possible explanation for the unexpected experimental results in the analysis of time to equilibrium and diversity of generated programs is that the concentration in each compartment is changed because of the change in CPL and CC.

Experimental results have also suggested that a sparsely connected network structure (greater CPL values) yields less complex programs than a densely connected network (smaller CPL values). In addition, a network structure with CC ranging from either 0 to 0.2 or from 0.8 to 1.0 manifests greatest complexity of the generated programs.

5.2. Future Work

We will investigate how gradient of concentrations changes the behavior of software self-assembly in greater details in our future work. To do this, we will set different migration probability of each compartment based on its size. This will essentially introduce nested compartment structures in which information between compartments can flow to the inner/outer world through its border. We will also investigate what effects (if any) do capacity, i.e., bandwidth, constraints have on $ASAP^2$.

The $ASAP^2$ system we introduced in this paper focuses on static self-assembly in which an equilibrium state will eventually be reached over time. We will present source and sink compartments in which software components are flown in or taken away from the system, hence introducing an $ASAP^2$ system far from equilibrium. In addition to the random and ordered graphs that we studied in this paper, we will further propose preferential attachment model[32] to investigate software self-assembly on scale-free networks in dynamic software self-assembly.

Acknowledgments

We thank the reviewers for their detailed and constructive comments. We acknowledge the EPSRC funding projects EP/D021847/1 and EP/E017215/1.

References

1. Hargreaves WR, Deamer DW. Liposomes from ionic, single-chain amphiphiles. Biochemistry 1978;17:3759–3768.
2. Hargreaves WR, Mulvihill S, Deamer DW. Synthesis of phospholipids and membranes in prebiotic conditions. Nature 1977;266:78–80.
3. Imae T, Mori O, Takagi K, Itoh M, Sawaki Y. Self-assembly formation of amphiphilic molecules mixed with photoreactive, aromatic unsaturated-acids: examination by light scattering. In: Colloid & polymer science. Berlin: Springer; 2004.
4. Vauthey S, Santoso S, Gong H, Watson N, Zhang S. Molecular self-assembly of surfactant-like peptides to form nanotubes and nanovesicles. Proc Natl Acad Sci 2002;99:5355–5360.
5. Hogg T. Robust self-assembly using highly designable structures. In: Papers from the Sixth Foresight Conference on Nanotechnology, 1999.
6. Luisi PL. The emergence of life. Cambridge, UK: Cambridge University Press; 2006.
7. Morowitz HJ. Beginning of cellular life. New Haven, CT: Yale University Press; 1992.
8. Dyson FJ. Origins of life. Cambridge, UK: Cambridge University Press; 1985.
9. Koza JR. Genetic programming: On the programming of computers by means of natural selection. Cambridge, MA: MIT Press; 1992.
10. Ryan C. Genetic programming 3: Darwinian invention and problem solving. Genetic Program Evol Mach 2000; 1(4).
11. O'Neill M, Brabazon A, Adley C. The automatic generation of programs for classification problems with grammatical swarm. In: Proc 2004 IEEE Congress on Evolutionary Computation, Portland, OR, June 20–23, 2004. Piscataway, NJ: IEEE Press. pp 104–110.
12. Abbott RJ. Object-oriented genetic programming, an initial implementation. In: Proc of the Sixth Int Conf on Computational Intelligence and Natural Computing, Embassy Suites Hotel and Conference Center, Cary, NC, Sept. 26–30, 2003.
13. Koza JR, Jones LW, Keane MA, Streeter MJ. Towards industrial strength automated design of analog electrical circuits by means of genetic programming. In: O'Reilly U-M et al., editors. Genetic programming: Theory and practice II; May 13–15, 2004.
14. Lohn J, Hornby G, Linden D. An evolved antenna for deployment on NASA's space technology 5 mission. In: O-Reilly U-M et al., editors. Genetic programming theory and practice II; May 13–15, 2004.
15. Abramson M, Hunter L. Classification using cultural co-evolution and genetic programming. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL, editors. Genetic Programming 1996: Proc First Annual Conf, Stanford University, CA; July 28–31 1996. Cambridge, MA: MIT Press. pp. 249–254.
16. Adorni G, Cagnoni S, Mordonini M. Genetic programming of a goalkeeper control strategy for the robocup middle size competition. In: Poli R, Nordin P, Langdon WB, Fogarty TC, editors. Genetic Programming: Proceedings of EuroGP'99, Goteborg, Sweden; May 26-27 1999. Volume 1598 of LNCS. Berlin: Springer-Verlag. pp. 109–119.
17. Davidson L, George S, Evans D. A biologically inspired programming model for self-healing systems. In: Workshop on Self-Healing Systems, Proc First Workshop on Self-Healing Systems. New York: ACM Press; 2002. pp 102–104.
18. Shen W-M, Will P, Khoshnevis B. Self-assembly in space via self-reconfigurable robots. ICRA 2003;2516–2521.
19. Li L, Krasnogor N, Garibaldi JM. Automated self-assembly programming paradigm: initial investigations. In: Proc Third IEEE Int Workshop on Engineering of Autonomic & Autonomous Systems, Potsdam, Germany; 2006. Washington, DC: IEEE Computer Society. pp 25–34.

20. Li L, Krasnogor N, Garibaldi JM. Automated self-assembly programming paradigm: A particle swarm realization. In: Proc Workshop on Nature Inspired Cooperative Strategies for Optimization, Granada, Spain; 2006. Granada, Spain: University of Granada. pp 123–134.
21. Eberhart RC, Kennedy J. A new optimizer using particle swarm theory. In: Proc Sixth Int Symp on Micro Machine and Human Science, Nagoya, Japan. Piscataway, NJ: IEEE Service Center, 1995. pp 39–43.
22. Hu X, Eberhart RC. Multi-objective optimization using dynamic neighbourhood particle swarm optimization. In: Proc 2002 Congress on Evolutionary Computation, Honolulu, HI; 2002.
23. Ray T, Liew KM. A swarm metaphor for multi-objective design optimization. Eng Optim 2002;34(2):141–153.
24. Coello CA, Lechuga MS. A proposal for multiple objective particle swarm optimization. In: Proc Congress on Evolutionary Computation (CEC'2002), Piscataway, NJ; 2002. pp 1051–1056.
25. Burke E, Gustafson S, Kendall G, Krasnogor N. Advance population diversity measures in genetic programming. In: Proc Parallel Problem Solving from Nature; 2002.
26. Watts DJ. Small Worlds: The dynamics of networks between order and randomness (Princeton Studies in Complexity). Princeton, NJ: Princeton University Press; 2003.
27. Frank H, Chou W. Topological optimization of computer networks. Proc IEEE 1972; 60(11):1385–1397.
28. Lin S. Effective use of heuristic algorithms in network design. In: The mathematics of networks, Proc Symposia in Applied Mathematics, number 26, Providence, RI: American Mathematical Society. pp 63–84.
29. Rouvray DH. Predicting chemistry from topology. Sci Am 1986;255(3):40–47.
30. Valverde S, Solé R. Self-organized critical traffic in parallel computer networks. Physica A 2002;312:636–648.
31. Farley AM. Population structure and artificial evolution. In: Artificial evolution. 2005. pp 213–225. **Q5**
32. Barabasi A-L, Albert R. Emergence of scaling in random networks. Science 1999;286:509.

Queries

Q1: AU: Figures 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 are in color in the source files. Figures appear in color online at no cost to you. If you would like any figure to appear in color in the print issue, please advise and we will send you a formal quote for the cost. Otherwise they will appear in black and white in the print issue and in color online.
Q2: AU: Confirm whether name and e-mail of the corresponding author are OK as set.
Q3: AU: All references are renumbered to appear in a sequence. Please confirm.
Q4: AU: Confirm whether citation of Figure 6 is OK here.
Q5: AU: Provide name and location of the publisher.