Computer-Aided Design 41 (2009) 649–667

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

EvoArch: An evolutionary algorithm for architectural layout design

Samuel S.Y. Wong *, Keith C.C. Chan

Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, China

ARTICLE INFO

Article history:
Received 26 October 2008
Accepted 20 April 2009

Keywords:
Architectural space topology
Evolutionary algorithm
Crossover
Genetic algorithm
Graph algorithm
Mutation

ABSTRACT

The architectural layout design problem, which is concerned with the finding of the best adjacencies between functional spaces among many possible ones under given constraints, can be formulated as a combinatorial optimization problem and can be solved with an Evolutionary Algorithm (EA). We present functional spaces and their adjacencies in form of graphs and propose an EA called EvoArch that works with a graph-encoding scheme. EvoArch encodes topological configuration in the adjacency matrices of the graphs that they represent and its reproduction operators operate on these adjacency matrices. In order to explore the large search space of graph topologies, these reproduction operators are designed to be unbiased so that all nodes in a graph have equal chances of being selected to be swapped or mutated. To evaluate the fitness of a graph, EvoArch makes use of a fitness function that takes into consideration preferences for adjacencies between different functional spaces, budget and other design constraints. By means of different experiments, we show that EvoArch can be a very useful tool for architectural layout design tasks.

1. Introduction

The combinatorial optimization problem is concerned with the efficient exploring of a large solution space consisting of discrete solutions. Many problems in the real world can be formulated as combinatorial optimization problems. If solutions of these problems can be encoded in linear-string chromosomes, they can be tackled with the use of a special form of Evolutionary Algorithm (EA) called the Genetic Algorithms (GAs), which has a long history of development. The standard GA starts off with an initial population of individuals generated at random or heuristically. At every evolutionary step, called a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, usually referred to as the *fitness*, or *fitness function*. To form a new population (i.e., the next generation), individuals are *selected* according to a selection scheme, such as the *roulette wheel selection scheme*, where individuals are selected with a probability proportional to their relative fitness. This ensures that individuals with greater fitness have a better chance of "reproducing".

Selection alone cannot introduce any new individuals into the population, that is, it cannot find new points in a search space. In EAs, they are generated by evolutionary operators such as *crossover* and *mutation*. Crossover is performed with a probability between two selected individuals, called *parents*, by exchanging parts of their genomes (encodings) to form two new individuals, called *offspring* or *children*. In its simplest form, substrings are exchanged after a randomly selected crossover point. Crossover operators tend to enable searches towards "promising" regions of the search space. The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random, with some small probability. GAs are stochastic iterative processes that are not guaranteed to converge. The termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level.

GA made use of a binary string encoding scheme when first proposed [1]. It was only in early 1990s other types of Evolutionary Algorithms (EA) were proposed and studied. Many other encoding schemes, such as floating point presentation in chromosomes, permutation, matrices presentation of graphs by Michalewicz [2], and tree encoding by Koza, etc. [3] have been developed and used with much success when tackling different problems in different application areas. Examples of such problems that GAs can solve include vehicle routing, the traveling salesman, linear programming, the knapsack problem, multi-objective optimization, etc [1]. For some problems in some application areas, such as the problem of automating spatial configuration, their solutions may not be easily represented in linear-string chromosomes. For such problems, solutions may best be represented as graphs.

One way to represent directed or undirected graphs is to use adjacency matrices. To evolve graphs, one can therefore evolve their corresponding adjacency matrices. Encoding an adjacency

* Corresponding author. Tel.: +852 92508588; fax: +852 28828568.
E-mail addresses: cssywong@comp.polyu.edu.hk (S.S.Y. Wong), cskcchan@comp.polyu.edu.hk (K.C.C. Chan).

0010-4485/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2009.04.005

matrix into linear chromosomes is required for GA applications. Dodd [4] encoded graphs by concatenating rows of an adjacency matrix into a linear string, the length of the chromosome has to be the square number of the dimension of the adjacency matrix. This square number constraint has to be maintained in order to enable successful decoding of the chromosome back into a graph. Hence, it does not favor variable length chromosome evolution. In order to produce legal offspring, it requires special encoding and decoding procedures and repair mechanisms before and after the evolution process. Furthermore, as the size of the graph increases, the encoding of all details will result in a very long chromosome.

Different indirect encodings of graphs are also developed for graph evolution. They generate graphs by reading instructions from genotypes in the form of a chromosome [5] or a tree [6]. Evolution is then carried out at the genotype level by using GA for chromosomes or Genetic Programming (GP) for trees. Other forms of indirect encoding involve evolution of graph generating rules such as that proposed by Kitano [7] or of encoding on a grammar based construction program in the genome proposed by Lindenmayer [8]. However, the efficiency of search is hampered by the indirect evolution on the genotype or evolving graph generation rules instead of the evolving the graphs themselves or their direct representations. This paper addresses this problem by encoding graphs directly into adjacency matrices so that evolution of graphs can take place directly with them using specially designed crossover and mutation operators. This encoding scheme avoids the need for special encoding and decoding procedures to be developed.

The problem of automating spatial configuration is concerned with the finding of feasible locations for a set of interrelated objects that meet design requirements and maximize design quality according to design preferences. Spatial configuration is necessary for all physical design problems such as component packing [9], route path planning [10], VLSI [11], and architectural layout design [12]. Of all these problems, the architectural layout design problem is particularly interesting because, in addition to common engineering objectives such as costs and performance, it is concerned especially with aesthetics and usability, which are generally more difficult to describe formally. Also, the components in a building layout (such as rooms or walls, etc.) often do not have pre-defined dimensions and every such component has to be resizable. These characteristics make the design problem very difficult.

The conventional approach to such architectural layout design problem is for an architect to receive a briefing from the client, usually a layperson, on functional requirements. Once such requirements are obtained, it is an architect's job to convert the client's requirements into building plans. The process begins with conceptual sketches of architectural space topologies of the functional spaces and their adjacencies based on the client preferences. These sketches are sometimes called bubble diagrams. After these diagrams are done, the architect's next task is to add dimensions to the space topologies drawn in the bubble diagrams and this whole process is idiosyncratic and very dependent on the artistic talent of individual architects.

To facilitate the architectural layout design process so that both experienced and less experienced architects can respond quickly to their clients' requests, we propose to use an EA called EvoArch (The name combines the two words Evolutionary and Architecture). The EvoArch represents spatial configuration in the form of a labeled graph so that functions such as bedrooms and kitchens can be represented as nodes and the adjacency between them, such as separation by a wall, etc., can be represented as edges. These graphs are in fact equivalent to bubble diagrams from which floor plans can be generated by inserting into them dimensions and geometries. The architectural design process is evolutionary in nature. As the work progresses, the architect chooses between alternative potential graphs with required adjacencies between functional spaces. Those graphs that do not lend themselves to desirable mix of functional spaces or potential of developing into reasonable dimensions and geometries will be discarded at early stage while leaving the others for further development. The use of EvoArch is to evolve potential graphs with preferred adjacencies of functional spaces for the architect's input of dimensions and geometries in the next stage to generate floor plans.

In addition to the preferred adjacencies, the budget and some other design constraints are imposed in the EvoArch process. The resulting graphs that satisfy these constraints will facilitate the ease of architectural layout plan generation in the next stage as there are fewer remaining requirements to be catered for. In the experiments, we apply EvoArch to the design of a house with 9 required functional spaces. Each functional space may present more than one time in the house depending on the budget available. The other design constraints to be imposed in EvoArch include the maximum number of functional spaces that can be adjacent to a particular functional space known as the valence, and the relative ratios of functional spaces in the house which is a design norm in the architect's profession. The resulting graphs will have different topologies at different costs and functional mixes for the client and the architect's evaluation and selection for further development into layout plans.

EvoArch encodes graphs in adjacency matrices to conduct evolution. The difficulty with the evolution of graph topologies using EAs is the tremendous increase in the complexity of the search space when the number of nodes of the graphs increases [13]. Even though this problem can theoretically be mitigated by a corresponding increase in population size so that more candidates can be utilized to evolve the best target, such an increase in population demands too much computational resource for the evolutionary process. To overcome this problem, we propose a novel uniform graph-based crossover operator, *Uniform Graph Crossover* (UGC), that can effectively facilitate the exchange of useful characteristics of two graphs.

The UGC that operates on adjacency matrices is similar to the *uniform crossover* operator for linear-string chromosomes in GAs. For such GAs, the uniform crossover operator has been shown to be more efficient at a heuristic search even when given a small population [14–16]. The reason is that such an operator is a relatively more disruptive operator and novel offspring can be generated even when the size of the population being evolved is relatively small to provide a sufficient variety of building blocks for the evolution of an optimal design.

In addition to the use of the UGC, EvoArch also makes use of a set of matrix based mutation operators called *Number-of-Node* mutation, *Number-of-Edge* mutation, *Node-Label* mutation and *Swap-Node* mutation operators. Given the design constraints based on the adjacency preferences of the functional space of a house and the budget constraints, EvoArch makes use of a fitness function to evolve optimal architectural space topologies. To evaluate the effectiveness of EvoArch, experiments to find optimal architectural space topologies were conducted based on different budget ranges on a set of adjacency preferences between the functional spaces required of a house. The results are very promising.

2. Literature review

The architectural layout design problem is concerned with the allocating of a set of space elements according to certain design criteria. A solution to this problem usually involves the specifying of topological or geometrical relationships between elements. Given a topology that describes the adjacencies between

space elements such as different geometrical shapes, these space elements can be arranged to satisfy the pre-specified topology. The mapping of geometries of space elements in a topology is therefore a many-to-one mapping. Given the geometries of space elements, it may therefore not be possible to retrofit them to satisfy a particular topology.

There have been some attempts to automate architectural space planning. For example, in [17], an evolutionary approach is described to map functional activities to a floor plan by encoding the activities in a chromosome. This evolutionary approach is an extension of the traditional GA, called genetic engineering [18], Other than the traditional crossover operator of swapping of genes between two chromosomes, this genetic engineering approach attempts to also convert low fitness genes by high fitness genes in a chromosome. The evolutionary approach described in [17] requires that the architectural space topology be given and fixed. Different functional activities encoded in different chromosomes, which represent different architectural spaces, are then swapped with each other during crossover so as to obtain optimal fitness of functional activities to individual spaces.

Other than the above, there have been studies on the automatic generation of floor plans by combining basic elements (such as a straight line of unit length etc.) of a plan using a set of combination rules or shape grammar [19]. The combination rules are encoded as a genotype in a chromosome on which GA or genetic engineering is applied. The floor plan is generated by decoding the chromosomes as a phenotype after the applying of the crossover or mutation operations on the genotype. The fitness is evaluated according to the phenotype—the floor plan. An illustration of this approach can be found in [20] where a square module is used as a basic module for the generation of floor plans. Such modules are to be combined to form higher level components which constitute the functional spaces of a house. The grammar of combination of the modules and the components are encoded in a chromosome on which GA is applied. The floor plan is generated according to the evolved combination rules. Like other indirect encoding method in EAs, the efficiency of evolution is not high. Furthermore, the resulting geometries of the rooms are dictated by the geometries of the basic elements, such as an aggregation of squares. As a result, the dimensions of the evolved plans can only be multiples of the basic elements. These are some constraints on how creative one can be!

Other studies on the automatic generation of architectural space plans start off with rectangular rooms that constitute a floor plan [21,22]. For example, in [22], a computer program called ARCHiPLAN can be used as a tool to manipulate the absolute locations, orientations, and dimensions of the rectangular rooms under a set of physical constraints. During the floor plan generation process, it checks the topological consistencies of the rooms by avoiding overlaps among them. In another example in [21], an approach for representing rectangular rooms as nodes of a tree is described. The topology of the tree is a description of room-adjacencies. The tree topology and the absolute locations of the rooms are encoded in different chromosomes and each of them undergoes crossovers and mutations independently. The separate results decoded from the chromosomes are put together in a floor plan and this is equivalent to a swapping of rectangular rooms attached to a tree topology where dimensions and orientation of the rooms as well as the tree topology are changing at the same time. The efficiency of search of such an approach is hindered by the combinatorial complexity. Furthermore, tree-representation is not a good representation of architectural space topology because it only represents the adjacencies between the nodes or rooms from one layer of node in the tree to the layer immediately above and below it. It cannot take into consideration adjacencies between rooms in the same layer or nodes separated by more than one layer of nodes in the tree. The result is that the search tends to be fixed around the first feasible design. Also, both [21] and [22] are designed for rectangular room plan manipulation but not other geometric shapes. Again, these represent constraints on the degree of creativity that can be exercised.

Current studies on automated architectural space planning concentrate on packing rooms with fixed geometries and evolving floor plans from basic elements with fixed dimensions. Architectural space topology plays a secondary role as a condition to be satisfied. The creative freedom of the architect is therefore restructured to a large extent by pre-determined geometric shapes and dimensions of the basic elements adopted in the design automation algorithms. The resulting floor plans tend to be predictable and mundane. In order not to restrain the creativity of an architect, we propose to start off the design process by evolving an optimal architectural space topology while leaving the insertion of geometry of rooms to the creative hands of the architect. Since the most natural form of presentation of architectural space topology is in form of a graph and it has been used in other researches for encoding floor plans [23,24], we propose to use an EA called EvoArch that can evolve such graphs. The details of EvoArch are given below.

3. The details of EvoArch

Many EAs are developed to handle special kinds of graphs. For example, GP algorithms [3,25] have been used to evolve trees and algorithms, such as [26], have been used to evolve Artificial Neural Network (ANN) topologies which can be considered as special bipartite graphs. These algorithms cannot be used to evolve architectural space topologies as these topologies cannot be represented easily as trees and bipartite graphs, What is needed is an EA that can evolve graphs without many of the topological constraints like those that other EAs have to work under. In this paper, we propose such an EA called EvoArch.

Using EvoArch, an architectural space topology can be represented as a graph with nodes representing space and edges their adjacencies. EvoArch converts these graphs into their adjacency matrices and conducts crossovers and mutations on these matrices. These reproduction operators of crossover and mutation can work with adjacency matrices to produce effects similar to the uniform crossover and mutation operators in the case of linear-string GA. Like other EAs, EvoArch finds optimal spatial topologies in several steps as follows:

1. Initialize a population of graphs at random and encode these graphs as their adjacency matrices so that evolution can be carried out on them.
2. Evaluate the fitness of each graph.
3. Select two graphs for reproduction using the roulette wheel selection scheme.
4. Apply crossover and mutation operators on the selected graphs.
5. Replace the least-fit graphs in the existing population by the newly generated offspring using the steady-state reproduction scheme.
6. Repeat Steps 2 to 5 until the termination criteria are met.

3.1. Encoding graphs in graph adjacency matrices

Encoding is usually the first thing one needs to decide when solving a problem with EA. Many encoding schemes, such as binary, permutation, value encoding, etc. in GA [1] have been developed and used with much success when tackling different problems in different application areas. In addition to these popular encoding schemes, there have been studies on cellular [27] or edge encoding [28], developed to encode tree-like structures. Similarly, work on GP [3,25,29] and the evolution of ANN [26,30,31] architecture has also been proposed.

With a few exceptions in some domain-specific studies [32,33], not much work on the use of a direct representation scheme to conduct EA for graphs in general has been done. Here, we propose a direct graph representation scheme. This scheme, which is used by EvoArch, is to encode graphs by encoding their adjacency matrices. An adjacency matrix represents graph nodes and their connectivity at the same time. Though these matrices can also be encoded in linear-string chromosomes which simple GAs can operate on by concatenating the rows of a graph adjacency matrix to form a linear array [34], it should be noted that this can be computationally clumsy. If linear-string chromosomes are used, the connectivity between nodes cannot be read directly. Special decoding is required to convert the linear-string chromosome back into a graph. Furthermore, the length of the chromosome also needs to be a square number of the dimension of the adjacency matrix $|V|^2$ or $|V|C_2$ if an upper diagonal adjacency matrix is used. Hence, it does not favor crossover of chromosomes of different lengths. Illustrative examples are given in Section 3.4.

An additional advantage of the adjacency matrix encoding scheme is that an effective crossover operator can be relatively easily implemented with it. The "repairing" of a matrix after crossover to ensure connectivity can also be more easily implemented with adjacency matrices (see the next section). In the following, the details of the proposed encoding scheme and evolution operators are described.

A graph adjacency matrix can be used to represent any graph including both directed and undirected graphs. Architectural space topologies can be represented in undirected graphs and the value, c_{ij}, of the ith row and jth column of the adjacency matrix of such a graph can be set to 1 if a connection exists between the ith to the jth node and c_{ij} can be set to 0 if there is no connection between them.

3.2. Reproduction operators

The EvoArch makes use of both crossover and mutation operators in its reproduction process. There is one type crossover operator that EvoArch uses and it is called the *Uniform Graph Crossover* (UGC) operator, allows any node to have equal chance of being selected to be swapped. For mutation, EvoArch makes use of two types of operators. They are called the *Number-of-Node*, *Number-of-Edge*, *Node-Label* and *Swap-Node* mutation operators.

When performing crossover and mutation, there is a need to ensure connectivity of the graphs being evolved. To do so, EvoArch embeds a spanning tree at random to a graph which may become disconnected after crossover or mutation. Starting with a spanning tree, addition of connections between nodes in a spanning tree and connection to new nodes will form a connected graph.

A spanning tree can be considered as a directed graph with the directed edge starting at the root and descending toward the terminal nodes. A spanning tree has $|V| - 1$ number of edges where $|V|$ is the number of nodes in the graph. Except for the root, each node receives a directed edge from the node above it. When represented in an adjacency matrix with the first column as the root, this property implies the second column to the end column will have only one $c_{ij} = 1$, where c_{ij} represents an edge directed from node i to node j. An example of such an embedded spanning tree in a graph is shown in Fig. 1. For the case of an undirected graph, embedding of a spanning tree into such a graph is presented by using the upper diagonal adjacency matrix shown in Fig. 2.

During crossover of graphs as described in the next section, each of the parent graphs will be broken up into two subgraphs. The subgraphs are then exchanged forming two intermediate degenerate graphs prior to re-connection to form the children graphs. The re-connection mechanism is done by first generating a spanning tree at random for each set of nodes of the intermediate degenerate graphs followed by embedment into them. This process does not impose any order on the re-connection. Hence EvoArch is a general EA with no bias.

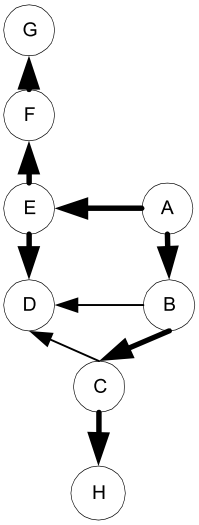
3.2.1. Uniform Graph Crossover (UGC) operator

The UGC operator allows two parent graphs, $G^{P1}(V^{P1}, E^{P1})$ and $G^{P2}(V^{P2}, E^{P2})$, that may contain different number of nodes to be crossed. Each labeled node in a graph represents a functional space. Each edge between two nodes represents the adjacency of the functional spaces the two nodes represent. Given the two graphs, $G^{P1}(V^{P1}, E^{P1})$ and $G^{P2}(V^{P2}, E^{P2})$, their corresponding adjacency matrices $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$ can be constructed.

The first step that UGC takes is to randomly permute the ordering of the nodes in the adjacency matrices of the two graphs, G^{P1} and G^{P2}. This is to ensure that there is no bias when nodes on one side of the cut line of the adjacency matrix of a parent graph are selected by the UGC operator for swapping with the adjacency matrix of another parent graph (see the formalism below). An adjacency matrix has the advantage that it is invariant to the row and column permutation. This permutation-invariant property of adjacency matrices does not exist in linear chromosomes in GA. In a linear chromosome, permutation of alleles will result in different properties and hence different fitness but this is not the case with adjacency matrices. The fitness of a graph is the same regardless of the permutation of the rows and columns of its adjacency matrix.

After row and column permutations, for each of $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$, a crossover point between two neighboring rows and columns in an adjacency matrix is selected randomly. One submatrix from each of the split parent matrix is then swapped to form two new child matrices. Connections within the submatrices are retained throughout the process while connections outside them are deleted. New edges are generated with repairing in each of the resulting matrices to form two children, $\mathbf{G}^{C1}$ and $\mathbf{G}^{C2}$. The UGC operator can be described step-by-step as follows.

1. Given two graphs, $G^{P1}(V^{P1}, E^{P1})$ and $G^{P2}(V^{P2}, E^{P2})$, with node sets $\{v_1^{P1}, v_2^{P1}, \ldots, v_i^{P1}, v_{i+1}^{P1}, \ldots, v_n^{P1}\}$ and $\{v_1^{P2}, v_2^{P2}, \ldots, v_j^{P2}, v_{j+1}^{P2}, \ldots, v_m^{P2}\}$ respectively. For the two graphs, corresponding adjacency matrices $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$ are constructed. The order of nodes is randomly permuted.
2. A crossover point in each of $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$ is randomly selected.
3. Assume that the crossover point for $\mathbf{G}^{P1}$ is between v_i^{P1} and v_{i+1}^{P1} and for $\mathbf{G}^{P2}$ is between v_j^{P2} and v_{j+1}^{P2}, the lower right portions of these two adjacency matrices are then swapped so that the rows and columns corresponding to $\{v_{i+1}^{P1}, \ldots, v_n^{P1}\}$ are swapped with the rows and columns corresponding to $\{v_{j+1}^{P2}, \ldots, v_m^{P2}\}$ to form two matrices $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$. The valid node labels for $\mathbf{G}^{PC12}$ are therefore given by $\{v_1^{P1}, v_2^{P1}, \ldots, v_i^{P1}, v_{j+1}^{P2}, \ldots, v_m^{P2}\}$ and for $\mathbf{G}^{PC21}$ by $\{v_1^{P2}, v_2^{P2}, \ldots, v_j^{P2}, v_{i+1}^{P1}, \ldots, v_n^{P1}\}$.
4. All cell entries in each of $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$ are scanned to remove invalid edges.
5. The number of edges to be added to each of $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$ respectively are then decided with a random number generator so that, for $\mathbf{G}^{PC12}$,
   $|V^{PC12}| - 1 \le |E^{PC12}| \le |V^{PC12}|C_2$ and for $\mathbf{G}^{PC21}$,
   $|V^{PC21}| - 1 \le |E^{PC21}| \le |V^{PC21}|C_2$.
6. A spanning tree is generated at random in each matrix and the remaining edges to be added are generated randomly in such a way that they connect nodes from $\{v_1^{P1}, v_2^{P1}, \ldots, v_i^{P1}\}$ with those from $\{v_{j+1}^{P2}, \ldots, v_m^{P2}\}$ and from $\{v_1^{P2}, v_2^{P2}, \ldots, v_j^{P2}\}$ with those from $\{v_{i+1}^{P1}, \ldots, v_n^{P1}\}$ and satisfy the number of edge requirements in 5.
7. Once edge-generation is complete, two children graphs, $\mathbf{G}^{C1}$ and $\mathbf{G}^{C1}$, are produced.

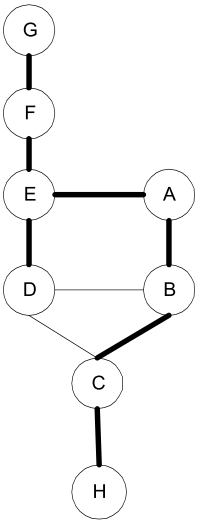


(a) Graph with spanning tree embedment and node A as root. The spanning is shown as thick directed edges.

	A	B	C	D	E	F	G	H
A	0	**1**	0	0	**1**	0	0	0
B	0	0	**1**	1	0	0	0	0
C	0	0	0	1	0	0	0	**1**
D	0	0	0	0	0	0	0	0
E	0	0	0	**1**	0	**1**	0	0
F	0	0	0	0	0	0	**1**	0
G	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0

(b) Adjacency Matrix for directed graph with a '**1**' in each column of non-root node of the embedded tree.

Fig. 1. Directed graph presentation illustrated.



(a) Undirected Graph. Spanning tree shown in thick edges.

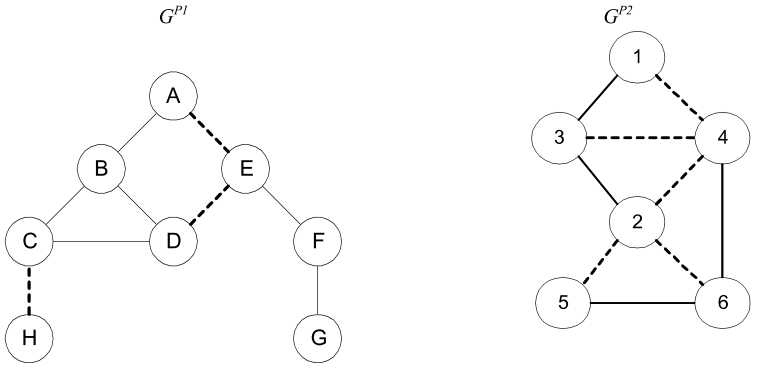
	A	B	C	D	E	F	G	H
A	-	**1**	0	0	**1**	0	0	0
B		-	**1**	1	0	0	0	0
C			-	1	0	0	0	**1**
D				-	**1**	0	0	0
E					-	**1**	0	0
F						-	**1**	0
G							-	0
H								-

(b) Adjacency Matrix for Graph. Spanning tree edges shown as '**1**'s.

Fig. 2. Undirected graph presentation illustrated.

For illustration, let us consider the following examples in Figs. 3–6. In order to indicate how nodes from one parent graph are swapped with the other, we use alphabetical node labels ('A' to 'H') for one parent graph and numerical node labels ('1' to '6') for the other. These labels may represent different functional spaces in a building. Using the functional spaces in Fig. 12 as an example, label '1' is the 'study area' of G^{P2} and label 'A' the 'study area of G^{P1}. Label '2' is the master ensuite of G^{P2} and label 'B' the master ensuite of G^{P1} and so on.

1. Nodes of 2 graphs in adjacency matrices $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$ are randomly permuted prior to crossover.
2. A crossover point in each of $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$ is randomly selected.
3. The lower right portions of these two adjacency matrices are then swapped to form $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$.
4. All cell entries in each of $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$ are scanned to remove invalid edges. Both of them are degenerate graphs.
5. The number of edges to be added to each of $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$ respectively are then decided with a random number generator so that,
   for $\mathbf{G}^{PC12}$, $|V^{PC12}| - 1 = 6 \leq |E^{PC12}| \leq |V^{PC12}|C_2 = 21$ and
   for $\mathbf{G}^{PC21}$, $|V^{PC21}| - 1 = 6 \leq |E^{PC21}| \leq |V^{PC21}|C_2 = 21$.
   In this example, say, $|E^{PC12}| = 10$, $|E^{PC21}|6$.
6. A spanning tree is generated at random for the nodes in each matrix and superimpose on the adjacency matrix of each of the degenerate graphs to re-connect them. A spanning tree is now embedded in each of the graphs. $\mathbf{G}^{PC12}$ has 7 edges. The requirement of $|E^{PC12}| = 10$ in step 5 is not satisfied in $\mathbf{G}^{PC12}$. New edges should be added to $\mathbf{G}^{PC12}$ to fulfill the requirement.



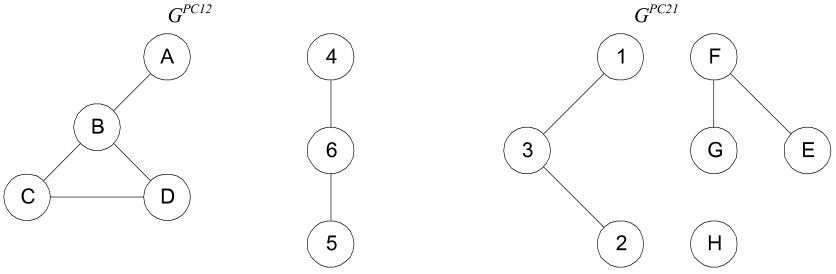
$\mathbf{G}^{P1}$

	A	B	C	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
C			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

$\mathbf{G}^{P2}$

	1	2	3	4	5	6
1	-	0	1	1	0	0
2		-	1	1	1	1
3			-	1	0	0
4				-	0	1
5					-	1
6						-

Fig. 3. Nodes are randomly permuted in the adjacency matrices. A crossover point is randomly chosen for $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$. Corresponding graphs are shown above the adjacency matrices. Invalid edges to be removed are shown in dotted lines.



$\mathbf{G}^{PC12}$

	A	B	C	D	4	5	6
A	-	1	0	0	0	0	0
B		-	1	1	0	0	0
C			-	1	0	0	0
D				-	0	0	0
4					-	0	1
5						-	1
6							-

$\mathbf{G}^{PC21}$

	1	2	3	E	F	G	H
1	-	0	1	0	0	0	0
2		-	1	0	0	0	0
3			-	0	0	0	0
E				-	1	0	0
F					-	1	0
G						-	0
H							-

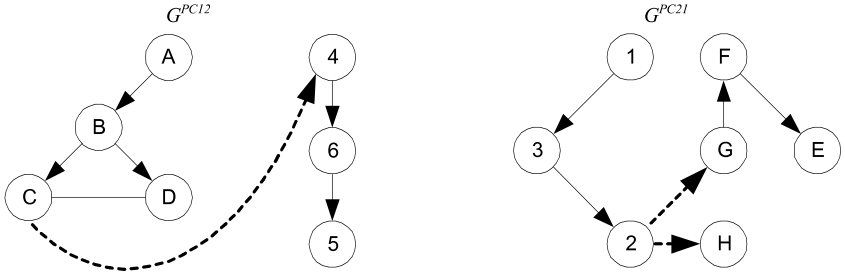
Fig. 4. Invalid edges are removed after swapping the lower right portions of $\mathbf{G}^{P1}$ and $\mathbf{G}^{P2}$. Degenerate graphs $\mathbf{G}^{PC12}$ and $\mathbf{G}^{PC21}$ are formed.

Note that if the edges in the resulting graphs are more than specified, edges that are not in the embedded spanning tree in each graph can be deleted at random to achieve the target.

7. Three new edges are randomly added to $\mathbf{G}^{PC12}$, say, $c_{D4} = 1$, $c_{D5} = 1$, $c_{A6} = 1$ to form $\mathbf{G}^{C1}$. $\mathbf{G}^{C2}$ is formed without addition of edges.

3.3. Mutation of graphs

The basic mutations of graphs that cause changes in graph topologies and node labels are introduced. They are *Number-of-Node Mutation*, *Number-of-Edge* mutation, *Node-Label* mutation and *Swap-Node* mutation. Similar to GA, the intention of mutation



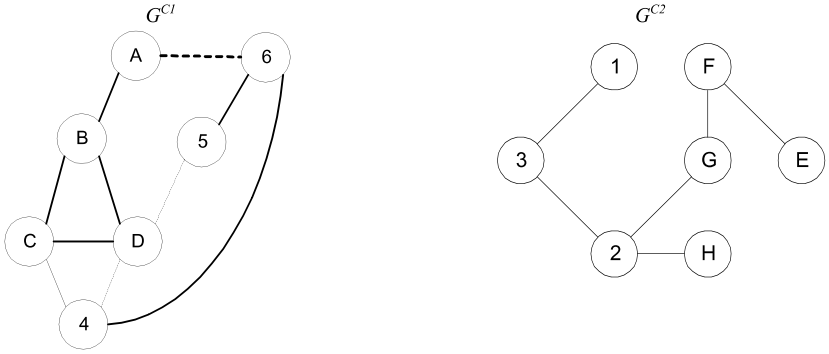
$\mathbf{G}^{\mathbf{PC12}}$

	A	B	C	D	4	5	6
A	-	**1**	0	0	0	0	0
B		-	**1**	**1**	0	0	0
C			-	1	**1**	0	0
D				-	0	0	0
4					-	0	**1**
5						-	**1**
6							-

$\mathbf{G}^{\mathbf{PC21}}$

	1	2	3	E	F	G	H
1	-	0	**1**	0	0	0	0
2		-	**1**	0	0	**1**	**1**
3			-	0	0	0	0
E				-	**1**	0	0
F					-	**1**	0
G						-	0
H							-

Fig. 5. Re-connect degenerate graphs by generating spanning tree at random. Spanning tree edges are shown as '**1**'s in the adjacency matrices and directed edges in the graph. New edges added to re-connect the degenerate graphs are shown as dotted edges in the graph.



$\mathbf{G}^{\mathbf{C1}}$

	A	B	C	D	4	5	6
A	-	1	0	0	0	0	**1**
B		-	1	1	0	0	0
C			-	1	1	0	0
D				-	**1**	**1**	0
4					-	0	1
5						-	1
6							-

$\mathbf{G}^{\mathbf{C2}}$

	1	2	3	E	F	G	H
1	-	0	1	0	0	0	0
2		-	1	0	0	1	1
3			-	0	0	0	0
E				-	1	0	0
F					-	1	0
G						-	0
H							-

Fig. 6. New edges $c_{D4} = 1$, $c_{D5} = 1$, $c_{A6} = 1$ in '**1**' are added randomly to $\mathbf{G}^{PC12}$. Children graphs $\mathbf{G}^{C1}$ and $\mathbf{G}^{C2}$ are formed.

is to bring about a breakthrough when the population converges to a suboptimal fitness.

3.3.1. Number-of-Node mutation

The *Number-of-Node* mutation operator allows us to increase or decrease the number of nodes in a graph by one. It works by selecting a node at random for deletion or addition. Its details are given below.

1. For a graph $G^P(V^P, E^P)$ with node set $\{v_1^P, v_2^P, \ldots, v_i^P, v_{i+1}^P, \ldots, v_n^P\}$ we construct its corresponding adjacency matrices, $\mathbf{G}^P$.
2. Generate at random either '0' or'1'.
3. If '0' is generated, a node, v_i^P, in $\mathbf{G}^P$ is selected at random.
4. v_i^P and all edges connected to v_i^P are deleted from G^P to form G^C with corresponding adjacent matrix $\mathbf{G}^C$.

$\mathbf{G}^P$

	A	*B*	C	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	***1***	0	0	0	0
C			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(a) Select the node B.

$\mathbf{G}^C$

	A	C	D	E	F	G	H
A	-	0	***1***	1	0	0	0
C		-	1	0	0	0	1
D			-	1	0	0	0
E				-	1	0	0
F					-	1	0
G						-	0
H							-

(b) For '0' case, delete the selected node B and connect node A and D which are previously connected to B.

$\mathbf{G}^C$

	A	B	C	D	E	F	G	H	*I*
A	-	1	0	0	1	0	0	0	0
B		-	1	1	0	0	0	0	0
C			-	1	0	0	0	1	0
D				-	1	0	0	0	0
E					-	1	0	0	***1***
F						-	1	0	0
G							-	0	0
H								-	***1***
I									-

(c) For '1' case, add node I to $\mathbf{G}^P$ and edges connecting to nodes E and H.

Fig. 7. The *Number-of-Nodes* mutation operator illustrated.

5. Connect nodes previously connected to v_i^P at random and the resulting node set of G^C is represented as $\{v_1^P, v_2^P, \ldots, v_{i-1}^P, v_{i+1}^P, \ldots, v_n^P\}$.
6. If '1' is generated, add node v_{n+1}^P to G^P and $\mathbf{G}^P$ to form G^C and $\mathbf{G}^C$ respectively with a new node set represented as $\{v_1^P, v_2^P, \ldots, v_i^P, v_{i+1}^P, \ldots, v_n^P, v_{n+1}^P\}$. Edges are generated at random to connect v_{n+1}^P to other nodes in $\mathbf{G}^C$.

An example to illustrate the *Number-of-Node* mutation operator is given in Fig. 7.

3.3.2. Number-of-Edge mutation

The *Number-of-Edge* mutation operator allows us to increase or decrease the number of edges in a graph. It works by selecting an edge at random for deletion or addition. Its details are given below.

1. For a graph $G^P(V^P, E^P)$ with edge set, E^P, we construct its corresponding adjacency matrices as $\mathbf{G}^P$.
2. Generate at random either '0' or'1'.
3. If '0' is generated, an edge in the graph G^P is selected and this is done by choosing a 'non-zero' cell in $\mathbf{G}^P$ randomly. '1' is then deducted from the value of the selected cell to form a child graph G^C and its corresponding adjacency matrix $\mathbf{G}^C$. Check if $\mathbf{G}^C$ is connected. If yes, stop. If not, the graph is broken down into two connected subgraphs and each has their own embedded spanning tree. Superimpose a spanning tree across the disjoint subgraphs to re-connect them.
4. If '1' is generated, an edge is added to the graph by adding a '1' to the value of a cell at random in $\mathbf{G}^P$ to form G^C and $\mathbf{G}^C$.

An example of the *Number-of-Edge*mutation operator is given in Fig. 8.

3.3.3. Node-Label mutation

The *Node-Label* mutation operator allows us to replace one node by another in the same graph. It works by selecting a node to be replaced and a node to replace it at random as follows.

1. For a graph $G^P(V^P, E^P)$ with node set $\{v_1^P, v_2^P, \ldots, v_i^P, \ldots, v_j^P, \ldots, v_n^P\}$ we construct its corresponding adjacency matrices as $\mathbf{G}^P$.
2. Select a node in the graph G^P and corresponding adjacency matrix $\mathbf{G}^P$ randomly. Assume that the node chosen in $\mathbf{G}^P$ are v_i^P.
3. Replace node label v_i^P by another node label chosen at random in the same graph, say, node label v_j^P, to form a child graph G^C and adjacency matrix $\mathbf{G}^C$. The order of node set in $\mathbf{G}^C$ is changed to $v_1^P, v_2^P, \ldots, v_j^P, \ldots, v_j^P, \ldots, v_n^P$.

We give an example of the *Node-Label* mutation operator in Fig. 9 below.

3.3.4. Swap-Node mutation

The *Swap-Node* mutation operator allows us to swap two nodes in the same graph. It selects a pair of nodes at random and then swaps them as follows.

$\mathbf{G}^P$

	A	B	C	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	***1***	0	0	0	0
C			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(a) For '0' case, select at random edge to be deleted.

$\mathbf{G}^C$

	A	B	C	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	***0***	0	0	0	0
C			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(b) Delete selected edge. Check if PC is connected. If yes, stop. If no, superimpose a spanning tree at random.

$\mathbf{G}^P$

	A	B	C	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
C			-	1	0	0	***0***	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(c) For '1' case, select at random cell to add an edge.

$\mathbf{G}^C$

	A	B	C	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
C			-	1	0	0	***1***	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(d) add an edge to the selected cell.

Fig. 8. The *Number-of-Edge* mutation operator illustrated.

$\mathbf{G}^P$

	A	B	***C***	D	E	***F***	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
C			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(a) Step 1. Select C to be replaced by F.

$\mathbf{G}^C$

	A	B	***F***	D	E	F	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
F			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H								-

(b) Step 2. Replace C with F.

Fig. 9. The Node-Label Mutation operator illustrated.

4. For a graph $G^P(V^P, E^P)$ with node set $\{v_1^P, v_2^P, \ldots, v_i^P, \ldots, v_j^P, \ldots, v_n^P\}$ we construct its corresponding adjacency matrices as $\mathbf{G}^P$.
5. Two nodes in the graph G^P are selected and this is done by choosing 2 nodes in $\mathbf{G}^P$ randomly. Assume that the nodes chosen in $\mathbf{G}^P$ are v_i^P and v_j^P.
6. Swap v_i^P and v_j^P to form a child graph G^C and adjacency matrix $\mathbf{G}^C$. The order of node set in $\mathbf{G}^C$ is changed to $v_1^P, v_2^P, \ldots, v_j^P, \ldots, v_i^P, \ldots, v_n^P$.

We give an example of the *Swap-Node* mutation operator in Fig. 10 below.

3.4. Comparison of EvoArch operators and conventional genetic algorithm operators

To compare conventional GA operators with that used by EvoArch, we use G^{P1} and G^{P2} in Fig. 3. The most direct form of encoding of graphs into a linear-string chromosome in GA is by concatenating the rows of the upper diagonal adjacency matrix as

G^P

	A	B	*C*	D	E	*F*	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
C			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
F						-	1	0
G							-	0
H							-	-

(a) Select two nodes F and C to be swapped.

G^C

	A	B	*F*	D	E	*C*	G	H
A	-	1	0	0	1	0	0	0
B		-	1	1	0	0	0	0
F			-	1	0	0	0	1
D				-	1	0	0	0
E					-	1	0	0
C						-	1	0
G							-	0
H								-

(b) Swap the nodes F and C.

Fig. 10. The *Swap-Node* mutation operator illustrated.

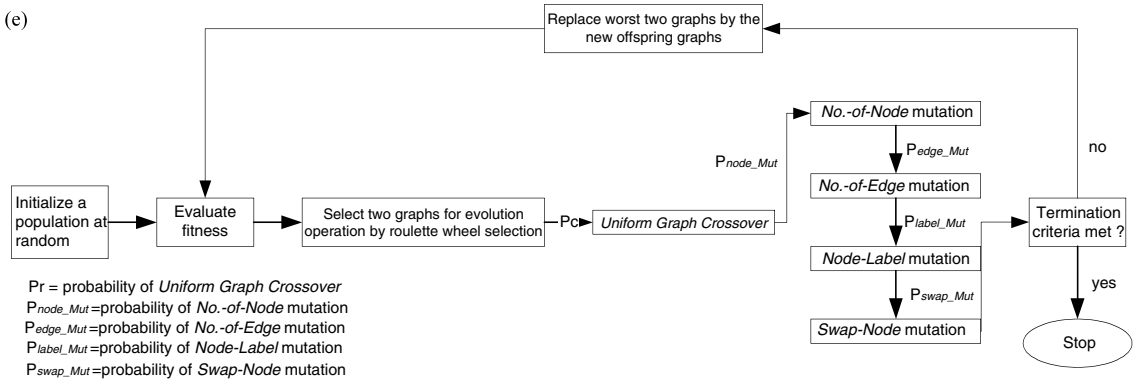
G^{P1}: ABCDEFGH1001000110000100011000100100 (|V|=8, length of binary string = $8C_2$ =28)
G^{P2}: 123456011001111100011 (|V|=6, length of binary string = $6C_2$ =15)
(a) Parent graphs encoded into linear-string chromosomes

G^{P1}: ABCDEFGH10010 / 0011000010001100010100
G^{P2}: 12345601100111/ 1100011
(b) Crossover point chosen at random for both parents

G^{C1}: ABCDEFGH100101100011 (|V|=8, length of binary string = 12 < $8C_2$)
G^{C2}: 12345601100111001100001000110001001000 (|V|=6, length of binary string = 31 > $6C_2$)
(c) Illegal children produced after crossover

G^{C1}: ABCDEFGH**I**1001000110000100011000100100 (|V|=9, length of binary string = 28 < $9C_2$)
G^{C2}: ACDEFGH1001000110000100011000100100 (|V|=7, length of binary string = 28 > $7C_2$)
(d) Illegal children produced by Number-of Node mutation of G^{P1} in (a)

(e)



Flow chart of EvoGraph process

Fig. 11. Comparison between conventional genetic algorithm and EvoArch on encoding graphs and the EvoArch flow chart.

illustrated in Fig. 11(a). The length of the binary strings following the node label strings in the chromosome should be $|V|C_2$. A crossover point is introduced at random for both G^{P1} and G^{P2} in Fig. 11(b) (symbol '/'). A conventional one point crossover is carried out to produce children G^{C1} and G^{C2} in Fig. 11(c). G^{C1} and G^{C2} are illegal children because they do not satisfy the length requirement for converting back into an adjacency matrix. This situation is very common for crossover of chromosomes with different lengths. Furthermore, linear-string chromosomes cannot directly represent the connectivity between nodes as do adjacency matrices. As such, the connectivity of nodes has to be reconstructed by some special repair procedures instead of simply incorporating a spanning tree in the adjacency matrix adopted in EvoArch. Hence, the use of adjacency matrices for crossover in EvoArch is more appropriate than linear-string crossover in conventional GA.

Mutation in conventional GA is by flipping a bit at random in the linear-string chromosome of binary presentation. In case of real valued representation in chromosomes, mutation is carried out by a random change of values of a gene within it. When *Number-of-Node* mutation is carried out using conventional GA mutation by addition (node I) or deletion (node B) of one node label at random in the chromosome G^{P1} in Fig. 11(a), illegal children are again produced in Fig. 11(d). An additional procedure is required to convert the children chromosomes to lengths conforming to their corresponding adjacency matrices with the new number of nodes. For *Number-of-Edge* mutation, addition of an edge can be carried out by randomly selecting a '0' in the linear-string chromosome and change it to '1'. The deletion of an edge is by converting at random a '1' to '0'. As the linear-string chromosome does not directly encode connectivity between nodes, it needs to be

	1. SA	**2**. MA	**3**. BED	**4**. LR	**5**. D&K	**6**. B	**7**. CP	**8**. P	**9**. CIR	**10**. EXT
1. Study area (SA)	-2	2	1	-2	-2	-2	-2	-1	3	3
2. Master ensuite (ME)	2	-2	2	-2	-2	-2	-2	3	3	3
3. Bedroom (BED)	1	2	2	-2	-2	3	-2	1	3	3
4. Living room (LR)	-2	-2	-2	-2	1	2	1	3	3	3
5. Dining & kitchen (D&K)	-2	-2	-2	1	-2	2	-2	1	3	2
6. Bathroom (B)	-2	-2	3	2	2	3	-2	-2	3	1
7. Car park (CP)	-2	-2	-2	1	-2	-2	-2	-2	3	3
8. Patio (P)	-1	3	1	3	1	-2	-2	-2	-2	3
9. Hall/stair/circulation area (CIR)	3	3	3	3	3	3	3	-2	3	-2
10. Exterior (EXT)	3	3	3	3	2	1	3	3	-2	3
Average	-0.2	0.3	0.9	0.5	-0.1	0.6	-0.7	0.2	2	2.2

Fig. 12. Adjacency preference matrix.

reverted back into adjacency matrix for connectivity checking after edge-deletion. The *Node-Label* mutation and *Swap-Node* mutation can be conducted by flipping node labels in the linear-string chromosome to achieve their purposes. There are no complications for these two types of mutation as there are no required-length and connectivity issues.

In summary, adjacency matrix is a more appropriate data structure to represent graphs as it encodes the node labels and their connectivity at the same time. EvoArch operators evolve adjacency matrices and produce legal offspring without the need of complicated repair or re-conversion procedures. These operators are applied with their respective probabilities in the evolution process stated in the beginning of this section. The process is summarized in the flow chart in Fig. 11(e).

4. Encoding architectural space topology and fitness function

At the design inception stage in a typical architectural project, the architect has to interpret requirements from a client who is, in most cases, a layperson. The client usually describes his or her preferred spatial groupings verbally. Such preferences can be translated into an *adjacency preference matrix*, which describes the preferred adjacency between functional areas. In Fig. 12, we give an example of an adjacency preference matrix of a house, drawn up based on the requirements of a client. The numbers, which make up an *Adjacency Preference Scale (APS)*, are defined in such a way that -2 means that the adjacency arrangement is *very much not preferred*, -1 means that it is *not preferred*, 1 means that it is *preferred*, 2 means that it is *very much preferred*, and 3 means that it is *extremely preferable*. Each functional space has an APS in relation to each of the other functional spaces as well as itself.

An adjacency preference matrix can be drawn up by interpreting the client's adjacency preferences between each pair of functional space into APS as shown in Fig. 12. Several APS can be obtained by interpreting the client's descriptive statements. Some examples are listed below. This matrix will be used in the fitness function and the experiments in Sections 4 and 5. The fitness function will maximize the APS subject to constraints. This will be discussed in Section 4.

A. Bedrooms should be grouped. $(\mathrm{APS}(2, 3) = 2)$.

B. Bathroom is to be shared by the bedroom, living room, and dining room.

$(APS(6, 3) = 3, APS(6, 4) = 2, APS(6, 5) = 2)$.

C. Patio is to be shared by master ensuite and living room and if possible, the dining room and bedroom. It is preferable to be exposed.

$(APS(8, 2) = 3,\ APS(8, 3) = 1,\ APS(8, 4) = 3,$
$APS(8, 5) = 1,\ APS(8, 10) = 3)$.

D. Circulation areas to link all rooms and carpark.

$(APS(9, i) = 3 \quad \text{where } 1 \leq i \leq 10 \quad \text{and} \quad i \neq 8, 10)$.

E. Study area to be attached to master ensuite. $(\mathrm{APS}(1, 2) = 2)$.

F. All rooms, carpark, and patio are preferred to be in contact with the exterior.

$(APS(10, i) = 3 \quad \text{where } 1 \leq i \leq 10 \quad \text{and} \quad i \neq 9)$.

The architectural layout design process is divided into two parts: topology and geometry. Topology refers to the logical relation between layout components. Geometry refers to the absolute location and size of each component layout. Topological decisions define constraints for the geometric design space. For example, a topological decision that "room i is adjacent to room j" restricts the geometric coordinates of room i relative to room j. Such decisions are important and have to be made before finalizing on the geometry. What a designer needs to do is therefore to try, as much as possible, to enumerate all topologies that can produce feasible geometries [22] and then review them to select those to explore geometrically. This process is slow and time-consuming. As it is very difficult for designers, especially the inexperienced ones, to run through the process of optimizing topologies manually, subsequent geometric designs produced may not best match the client's need.

To overcome this problem, EvoArch can be used to find possible optimal topologies in the design process so that, instead of having to deal with too many feasible but suboptimal topologies, designers need only deal with the optimized ones when starting geometry design.

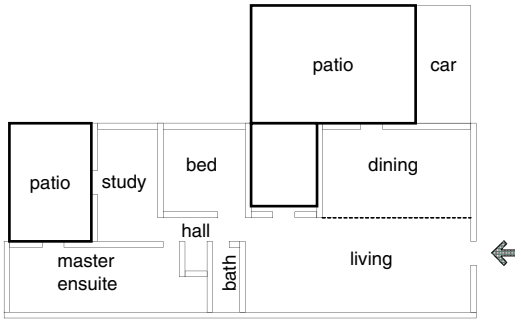


Fig. 13. The functional area floor plan of a house.

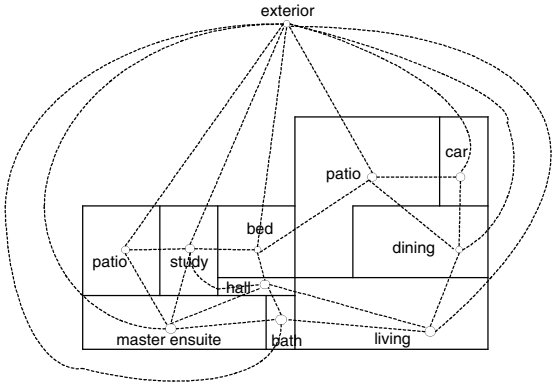


Fig. 14. Graph (dotted line and circular node) representing architectural space topology of the floor plan in Fig. 13.

4.1. Conversion between floor plan and graph representing architectural space topology

To use the graph representing the optimal architectural space topology evolved by EvoArch, we demonstrate how the graph can be converted into a floor plan and vice versa. If two architectural spaces, represented by two nodes, share a common boundary, say a wall, we represent this adjacency by an edge connecting the two nodes. Figs. 13 and 14 give an example of the floor plan of a house with respective functional rooms and its corresponding graph representation on its architectural space topology. The graph in Fig. 14 should be a planar graph which can be encoded in a graph adjacency matrix. EvoArch conducts evolution on the graph adjacency matrices to obtain the optimal graph topology. Floor plans can then be drawn up by decoding the resultant optimal graph adjacency matrix.

For decoding, given an optimized graph adjacency matrix, the corresponding graph is drawn. This graph may represent a one storey or a multi-storey building. It can be decomposed into one or more planar subgraphs. Each of the planar subgraphs represents a floor plan of a storey. There can be more than one way of decomposing a graph that represents a multi-storey building into planar subgraphs and it is up to the architect to decide which way to choose. There have been algorithms designed to extract planar graphs from a graph that are useful to the architects [35,36]. This process is equivalent to the architect's juggling with an optimized bubble diagram in the conceptual design stage. An example on decomposition of a graph into planar subgraphs is demonstrated in Section 5.4 by using one of the optimal architectural space topology generated by the experiments. In the following, we show how a planar graph can be converted into a space enclosure representing a floor plan.

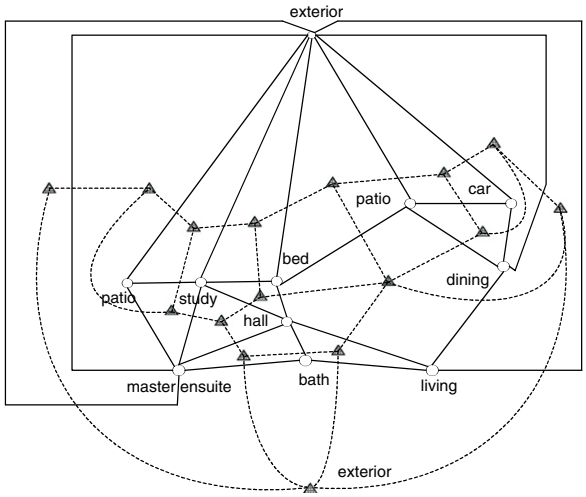


Fig. 15. Dual graph (dotted lines and triangular nodes) construction from the solid line graph.

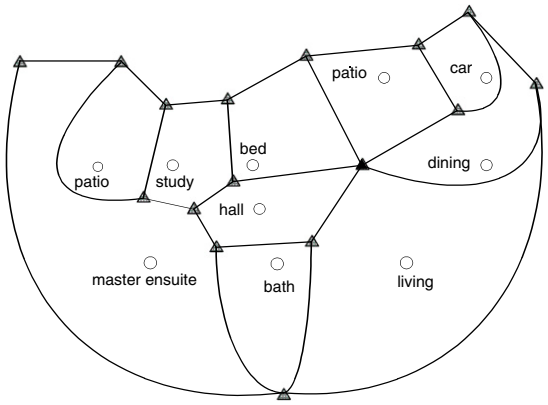


Fig. 16. Dual graph of the solid line graph in Fig. 15. It can be converted to floor plans by inserting geometries to the space bounded by the edges.

Given a planar subgraph that represents the architectural space topology of a floor plan, its dual graph (which is also planar) can be drawn up in two steps illustrated by an example in Figs. 15 and 16 as follows.

1. Lay out the planar graph on a plane without crossing edges (shown as solid lines and circular nodes in Fig. 15).
2. Every edge in the planar graph represents the boundary separating the two spaces corresponding to the nodes it connects. In order to construct the dual graph, a pair of nodes is created on both sides of an edge in the planar graph (shown as triangular-shape nodes in Fig. 15). A new edge is then created to connect the newly created nodes (shown as a dotted line in Fig. 15). These new edges are the spatial boundaries of the architectural space topology in the planar graph. The dual graph in Fig. 16 can then be derived from Fig. 15.

In this example the adjacencies between functional spaces in Fig. 15 are the same as the floor plan in Fig. 13. The conversion of the dual graph in Fig. 16 into the floor plan in Fig. 13 is done by inserting geometries into the space enclosed by the edges in the dual graph. Note that more than one floor plan with different geometries can be generated from the same dual graph as many geometries may fit a single space bounded by the edges in the dual graph. This provides creative freedom for the architects.

4.2. Design objectives and the fitness function

The final configuration of the resulting graph can be controlled by the evolution operators or the fitness function in the evolution process. The introduction of specific order in the evolution operators may destroy theirs randomness and render their tackling of large search space of graph topologies inefficient. Hence, we use fitness function to control the final configuration of the graph evolved. The fitness function is designed to always produce a positive value and to ensure that this value increases with the client's satisfaction with the design as well as the degree of fulfillment of the design constraints. As such, its value is to be maximized in the evolutionary process and it can be ranked for the purpose of selection during evolution. As common to all EAs, non-fit graphs may be produced in the evolution process but further breaking them into subgraphs by UGC or mutation operators may provide potential building blocks for highly fit graphs. Given that the objective is to find optimal architectural space topologies, we propose to use a fitness function in the evolutionary process that takes into consideration: (i) clients' preferences as given in an Adjacency Preference Matrix and (ii) physical constraints as given by an *Adjacency Limitation* defined to be the maximum number of adjacent rooms that one room can be in contact with, (iii) budget constraint, (iv) the range of relative ratios between rooms and (v) the minimum functions that are required to constitute an acceptable design. All these constraints can be quantified in our everyday life but their numerical value may differ a lot. For example, the budget is in the order of million dollars while the room ratios are within the range of unit figures. The large figures may dominate the small ones if they are used indiscriminately in the fitness function. In order to avoid this issue and to keep the fitness function simple, we adopt normalized values instead of actual values in the fitness function. The set of actual values used in the fitness function can be transformed to comparable normalized values by assigning their lowest non-zero actual value as one in the normalized scale, so that based on it, other normalized values are assigned in proportion to the lowest non-zero actual value. We use hypothetical normalized values in integers which are comparable to each other so that no one of them is dominating the other. The adjacency preference is quantified in Fig. 12 as APS that ranges from −1 to 3. This scale can be obtained by interpreting the client's preference in an interview. The cost is to be minimized in the fitness function. We intend to explore different optimal solutions for different budgets in the experiments. There is a possibility of evolving solutions well below a given budget in the expense of functional spaces due to the cost minimization effect built in the fitness function. This may lead to the result that only very few optimal graphs can be obtained at the lowest costs. Hence, a lower bound is set for each given budget resulting in a budget range for each experiment. Any cost deviation from the given budget range will be punished. As such, higher cost solutions can be explored. The maximum valence of a node is decided by the normal geometry of the functional space it represents. It is common to have a room rectangular in shape. Its four walls, ceiling and floor can be adjacent to other rooms. Though the number of rooms adjacent to it can be more than 6 or a room can be of other shapes, we adopt 4 as the maximum valence to avoid over compact configuration. The circulation space can link to more rooms and hence we adopt 6 as the maximum valence. The cost of providing each node is a hypothetical normalized value. The exterior is free of charge (cost = 0), the car park is the cheapest to build (cost = 1) and the living room is the most expensive (cost = 6). The maximum valence and costs are given in Table 1a. The acceptable range of relative room ratios is design norms adopted by an architect. They are given in Table 1b. The 'circulation areas' is a supportive space an there is no relative room ratio constraint imposed. They are generated throughout the process as necessary subject to other constraints. There is also no room ratio constraint to the 'exterior' as it is not a functional space. The 9 different functions in Fig. 12 should be included in each design as the minimum requirement. The fitness function is defined as follow.

Table 1a
The function space description, maximum valence, and cost corresponding to the room labels.

Label	Functional space	Max. valence	Cost
1	Study area	4	3
2	Master ensuite	4	4
3	Bedroom	4	3
4	Living room	4	6
5	Dining and kitchen	4	5
6	Bathroom	4	2
7	Carpark	No limit	1
8	Patio	No limit	4
9	Hall/stair/circulation area	6	4
10	Exterior	No limit	0

Table 1b
The relative room ratios.

Function 1/Function 2	Min. ratio	Max. ratio
Study area/Master ensuite	0	1
Study area/Bedroom	0	1
Study area/Patio	0	4
Master ensuite/Patio	1	2
Bedroom/Living room	1	3
Bedroom/Bathroom	1	2
Bedroom/Patio	1	4
Living room/Bathroom	1	2
Living room/Patio	1	2

$$fitness = \frac{x}{2^{a+b+c+d}} \tag{1}$$

where x = sum of the Adjacency Preference (APS).

a = absolute deviation to the budget range.

b = total valence exceeding the maximum allowed for each node.

c = total absolute deviation to the allowed range of room ratios.

d = number of functions deficient in the graph.

The APS is the value to be maximized and the deviations to the other assigned constraints a, b, c, and d are the values to be minimized. The denominator is made an exponential function to avoid it from being equal to zero in the process of evaluation during the evolution. When all the imposed constraints are satisfied, the values of a, b, c and d will be 0. In such circumstance, all the design constraints are satisfied by the resulting graph and its fitness value equals the APS of the graph topology.

Once terminated, the optimal graph can be decoded and the floor plans can be drawn as described in Section 4.1. Using the optimal architectural space topology given in a form like Fig. 15, an architect can insert his or her favorite architectural motifs or geometries to complete the design. Different optimal graphs may have different proportions of functional spaces (functional mix) and different costs. This is the freedom allowed in the fitness function to provide options to the client and the architect to exercise their judgmental call to select them for further development.

5. Experimental setup

The objective of the experiments we performed is to determine if EvoArch can be used to effectively find an optimal architectural topological design. For this purpose, we assume that we need to design a house with 9 functional spaces with an adjacency

Table 2
The probabilities of each operator being selected for reproduction.

Operator	Probabilities of being selected
Uniform Graph Crossover	0.2
Number-of-Edge mutation	0.2
Number-of-Node mutation	0.2
*Node-Label*mutation	0.2
Swap-Node mutation	0.2

Table 3
Experiment and budget range.

Experiment	Budget range
1	30–34
2	35–39
3	40–44
4	45–49
5	50–54
6	55–59

preference matrix as shown in Fig. 12, and each functional space has other constraints specified in Tables 1a and 1b. Using 10 nodes that include 9 functional areas and one node representing the exterior, EvoArch generates optimal design topologies that satisfy budget requirements and other design constraints. These alternatives compete with each other by compromising design objectives in give-and-take situations and eventually arrive at the one that best fits the design intent expressed in the fitness function. For our experiments, the fitness function given by Eq. (1) is used.

5.1. Initialization and EvoArch parameters selection

In our experiments, an initial population of 100 graph adjacency matrices is generated at random. The nodes encoded in each of them are given the same labels as that in Fig. 12. The crossover and mutations operators selected include all of UGC, *Number-of-Nodes mutation*, *Number-of- Edge mutation*, *Node-Label mutation* and *Swap-Node mutation*. They have equal probabilities to be selected and applied (Table 2).

5.2. Running EvoArch

Using the parameters as described above, we run EvoArch on the population of 100 graph adjacency matrices generated at random. The maximum number of generations to be run for each experiment is 5000. To compare performance on the same basis, all experiments start with the same randomly generated initial populations. Experiments are carried out each with a budget interval of 5 starting from the lowest range of 30 to 34, up to the highest range of 55 to 59 (see Table 3). Each experiment is run 10 times and the one with the highest APS is adopted. The reason for adopting a budget interval of 5 is to allow a variation of one to two rooms (cost per room range between 2 to 5 in Table 1a) between each interval. All the other constraints on budget, valence of nodes, minimum functions required, and range of room ratios as described in Section 4.2 should be satisfied in order to be acceptable as a valid solution. In such circumstances, the fitness value according to (1) should be equal to the total APS of the resulting graph.

5.3. Discussion of results and limitations

Optimal architectural space topologies are obtained in the form of adjacency matrices for each experiment. It should be emphasized that this optimal is based on the adjacency preferences and constraints input to EvoArch and the optimal graph evolved is the one that is closest to these input targets. The limitation is that EvoArch can only achieve quantitative targets. The overall selection criteria based on the preference of functional mix and the give-and-take on APS versus cost on the resulting graphs has to base on the client's personal liking and the architect's experience. Hence, given the limited preliminary information in the initial stage of design, EvoArch plays a role of providing quick alternative graphs for an architect to select for further development.

Table 4
Summary of experimental results.

Experiment	Budget range	No. of nodes	Converging generation	APS	Cost	APS/Cost
1	30–34	10	1312	55	32	1.719
2	35–39	11	611	64	36	1.778
3	40–44	12	1536	74	40	1.85
4	45–49	13	1036	74	45	1.644
5	50–54	15	1294	108	52	2.077
6	55–59	16	2101	113	57	1.982

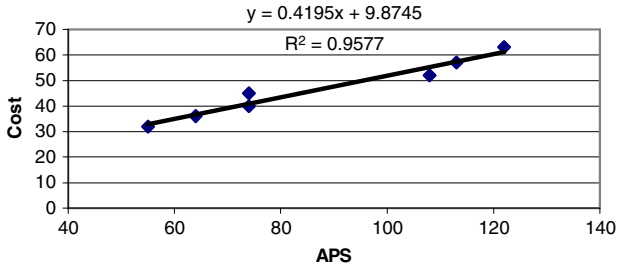


Fig. 17(a). Relation between cost and adjacency preference scale (APS).

All adjacency matrices with corresponding APS that represent the optimal architectural space topologies obtained in the experiments are presented in the Appendices A and B. When more than one node has the same function, a small letter will be added to the node number label to differentiate it from the others. For example, two different 'circulation areas' will be represented by '9a' and '9b' etc. The fitness plots throughout the evolution for different experiments are also included.

The results will be first analyzed in the APS cost efficiency aspect. That is, the adjacency preference of the client achieved with respect to cost expressed in APS per unit cost. High cost efficiency does not necessarily mean a quality design with a good balance of all factors because the solution may bias toward high average APS elements (Fig. 12) that render further development of a quality architectural plan difficult. An overall appraisal that takes into consideration the client's adjacency preference, the functional mix and cost is required for the selection of solution for further development. The experimental results on the properties of the graphs generated with respect to the number of nodes, number of generations to convergence, APS and costs are summarized in Table 4. All results satisfy the constraints on valence and room ratios required in Tables 1a and 1b.

The relation between costs and APS is given in Fig. 17(a) if the points are plotted on a graph. The relationship between them is close to linear with least square regression value close to 1 at 0.9577. This reflects the fact that the more the client pays, the more satisfaction on the adjacency preference he can derive from the design. The marginal increase in APS per unit increase in cost is approximately 0.42.

The relation between the number of generations it takes to convergence and the number of nodes is plotted on a graph as

Table 5
Number of functional spaces in the resulting graph in each experiment.

Exp.	Cost	Number of functional spaces in a resulting graph										Remarks on functional mix
		1. SA	**2**. MA	**3**. BED	**4**. LR	**5**. D&K	**6**. B	**7**. CP	**8**. P	**9**. CIR	**10**. EXT	
1	32	1	1	1	1	1	1	1	1	1	1	Balanced
2	36	1	1	1	1	1	1	1	1	2	1	Balanced
3	40	1	2	1	1	1	1	1	1	2	1	Balanced
4	45	1	2	2	2	1	1	1	1	1	1	Bias toward bedrooms
5	52	1	1	1	1	1	1	1	1	6	1	Biased toward circulation areas
6	57	1	1	2	2	1	1	1	1	5	1	Biased toward bedrooms and circulation areas

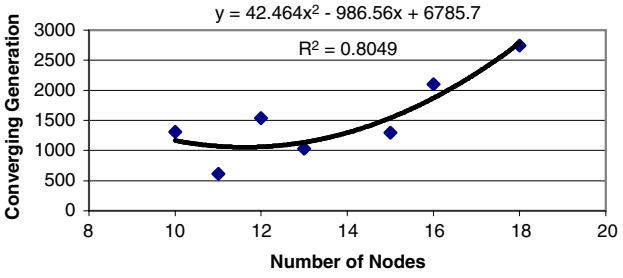


Fig. 17(b). Relation between number of nodes and converging generation.

shown in Fig. 17(b). It approximates a parabolic relation with squared regression value 0.8049. The number of generations required for convergence is proportional to $|V|^2$. This reflects the efficiency of generating optimal graphs in relation to the number of functional spaces required.

Second, we analyze the functional mix in the optimal graphs of different costs evolved in the experiments. The number of individual functional spaces generated in the resulting graphs at different is summarized in Table 5. The preference on functional mix is a judgmental call that depends on the client's preference as well as the architect's design experience. In general, the further development of an optimal graph with a balance of all the functions and reasonable cost will end up with a design that conforms to the norm. Experiments 1 to 3 have a balanced functional mix in the optimal graph as shown in Table 5. We select the optimal graph generated in experiment 3 with median cost as an example to develop the floor plan layout in Section 5.4.

5.4. Example on architectural layout plan development from an optimal graph

Using the approach given in Section 4.1, each of these optimal graphs can be converted to topological space and then to a floor plan. To do so, it should be noted that each architectural space topology may have more than one corresponding geometrical floor plan. As an illustration of this conversion process, we consider the optimal graph generated in experiment 3. A geometrical plan that can be developed based on the optimal architectural space topology shown in Fig. 18 is demonstrated in Fig. 19. When there are overlapping edges in a resulting graph, it is separated into layers without overlap to represent different floors (differentiated by solid and broken edges and nodes in Fig. 19).

6. Conclusion

In this paper, we propose an EA called EvoArch that can effectively evolve space topologies for architectural layout design. We demonstrate how adjacency matrices can be used as a more effective representation scheme for graph topologies than linear string chromosomes. We also describe how EvoArch uses a set of crossover and mutation operators and a relevant fitness function

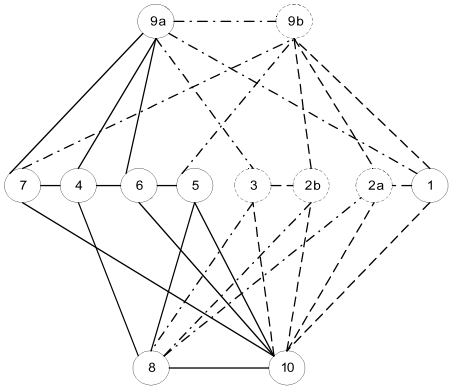
	8	**2a**	**2b**	**7**	**5**	**4**	**6**	**3**	**1**	**9a**	**10**	**9b**
8	-	3	3	0	1	3	0	1	0	0	3	0
2a		-	0	0	0	0	0	0	2	0	3	3
2b			-	0	0	0	0	2	0	0	3	3
7				-	0	1	0	0	0	3	3	3
5					-	0	2	0	0	0	2	3
4						-	2	0	0	3	0	0
6							-	0	0	3	1	0
3								-	0	3	3	0
1									-	3	3	3
9a										-	0	3
10											-	0
9b												-

Fig. 18. Adjacency matrix with APS generated by experiment 3.

in the evolutionary process to find optimal designs. Based on the experiments performed, we showed that EvoArch could be an invaluable tool for architects to convert some of the descriptive requirements of their clients to several possible architectural space topologies that satisfy the client's needs. This saves the architect from doing spatial configuration manually by the conventional approach through trial-and-error.

We show that EvoArch can be an effective technique for finding an "optimal" architecture layout design by experiment using 10 different nodes each representing a functional space of a house. EvoArch can maximize a client's adjacency preferences on functional spaces with a given budget and a set of design constraints. The time of convergence of EvoArch is in the order of the square of the number nodes. The maximum number of nodes of graph generated by the experiments is sixteen. As the time of convergence increases with the number of nodes, the development of parallel computing can be explored for evolving larger graphs to improve computational speed. The adjacency preferences gained from an optimal architectural space topology is directly proportional to its costs. A number of optimal architectural space topologies is generated for selection for further development into a geometric plan. The selection criteria depend on personal preferences such as cost efficiency (adjacency preference gained per unit cost) and balance of functional mixes. Further development of the selected optimal architectural topology into a geometric plan requires the extraction of planar graphs from the graph representing the selected optimal architectural space topology. Though this is not part of the EvoArch algorithm, improvement on existing algorithms on planar graph extraction from a graph is worth exploring because it helps to handle large graphs and it is a step towards full automation of architectural design.

EvoArch offers freedom for geometric design in a 2-dimensional plane by providing an optimal graph with preferred adjacencies



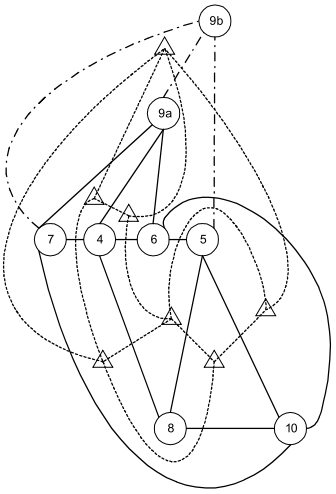
Graph generated by experiment 3

APS = 74

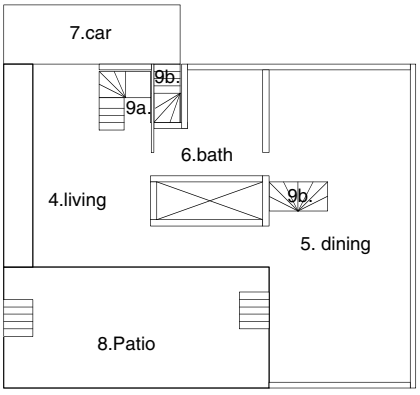
Cost = 40

Graph separation layers of planar subgraphs for floor plan generation:

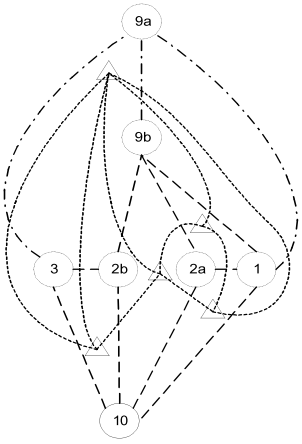
1. Solid line – Ground Floor (nodes 4, 5, 6, 7, 8, 9a, 9b,10)
2. Dot and solid line – stairs and the circulation from first floor rooms to ground floor patio .
3. Dotted Line – First Floor (1, 2a, 2b, 3, 9a, 9b, 10)
4. Node 10 is the exterior that can be used by all floors
5. Nodes 9a and 9b are circulation spaces that are commutable.



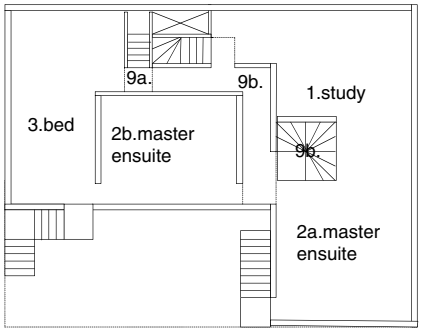
Ground Floor Graph



Ground Floor Plan (nodes 4, 5, 6, 7, 8, 9a, 9b, 10)



First Floor Graph



First Floor Plan (nodes 1, 2a, 2b, 3, 9a, 9b, 10)

Fig. 19. Optimal architectural space topology generated by experiment 3 and corresponding floor plans.

between functional spaces. The same principle can be applied to evolve 3-dimensional space that current Building Information Modeling (BIM) is attempting to do with limited availability of predefined geometries. In combination with a more holistic approach to design incorporating other factors such as materials selection and life cycle costing, more variables are required to be handled and application of parallel computing to EvoArch to achieve such a purpose is another new topic to be explored.

Appendix A. Adjacency matrices representing optimal architectural space topologies generated by experiments

	10	1	3	5	4	6	7	2	9	8
10	-	3	3	2	3	1	3	3	0	3
1		-	1	0	0	0	0	2	3	0
3			-	0	0	3	0	0	3	0
5				-	1	0	0	0	3	1
4					-	2	0	0	0	3
6						-	0	0	3	0
7							-	0	3	0
2								-	3	3
9									-	0
8										-

Experiment 1 (Budget 30 to 34)
No. of nodes = 10
APS = 55
Cost = 32

	9a	2	1	6	10	5	3	7	8	9b	4
9a	-	3	3	3	0	0	3	3	0	3	0
2		-	0	0	0	0	2	0	3	3	0
1			-	0	3	0	0	0	0	3	0
6				-	1	2	3	0	0	0	0
10					-	2	0	3	3	0	3
5						-	0	0	1	3	0
3							-	0	1	0	0
7								-	0	3	1
8									-	0	3
9b										-	3
4											-

Experiment 2 (Budget 35 to 39)
No. of nodes = 11
APS = 64
Cost = 36

	8	2a	2b	7	5	4	6	3	1	9a	10	9b
8	-	3	3	0	1	3	0	1	0	0	3	0
2a		-	0	0	0	0	0	0	2	0	3	3
2b			-	0	0	0	0	2	0	0	3	3
7				-	0	1	0	0	0	3	3	3
5					-	0	2	0	0	0	2	3
4						-	2	0	0	3	0	0
6							-	0	0	3	1	0
3								-	0	3	3	0
1									-	3	3	3
9a										-	0	3
10											-	0
9b												-

Experiment 3 (Budget 40 to 44)
No. of nodes = 74
APS = 74
Cost = 40

	9	3a	4a	7	2a	3b	2b	5	6	4b	8	1	10
9	-	3	3	0	3	3	0	3	3	0	0	0	0
3a		-	0	0	0	0	0	0	0	0	1	1	3
4a			-	1	0	0	0	0	0	0	3	0	3
7				-	0	0	0	0	0	0	0	0	3
2a					-	0	0	0	0	0	3	2	3
3b						-	2	0	3	0	1	0	0
2b							-	0	0	0	3	2	3
5								-	0	1	1	0	2
6									-	2	0	0	1
4b										-	3	0	3
8											-	0	3
1												-	3
10													-

Experiment 4 (Budget 45 to 49)
No. of nodes = 13
APS = 74
Cost = 45

	9a	9b	6	3	9c	4	2	9d	7	9e	5	9f	8	1	10
9a	-	3	3	3	3	0	0	0	3	0	0	0	0	3	0
9b		-	3	3	0	3	3	0	0	0	0	3	0	0	0
6			-	3	0	0	0	3	0	0	0	0	0	0	0
3				-	0	0	0	0	0	3	0	0	0	0	0
9c					-	3	0	3	3	0	3	3	0	0	0
4						-	0	0	0	3	0	0	0	0	3
2							-	3	0	3	0	0	3	0	0
9d								-	3	0	3	0	0	3	0
7									-	3	0	3	0	0	3
9e										-	3	3	0	0	0
5											-	3	0	0	0
9f												-	0	3	0
8													-	0	3
1														-	3
10															-

Experiment 5 (Budget 50 to 54)
No. of nodes = 15
APS = 108
Cost = 52

	10	2	9a	3a	3b	6	4a	9b	1	9c	9d	8	4b	5	9e	7
10	-	3	0	3	3	1	3	0	3	0	0	3	3	2	0	3
2		-	0	0	2	0	0	0	0	0	3	0	0	0	3	0
9a			-	0	0	3	0	0	3	3	0	0	0	3	3	3
3a				-	0	3	0	0	0	0	3	1	0	0	0	0
3b					-	0	0	0	0	3	0	0	0	0	3	0
6						-	0	0	0	0	3	0	0	0	0	0
4a							-	0	0	0	3	3	0	0	0	1
9b								-	3	3	3	0	3	3	0	3
1									-	0	0	0	0	0	3	0
9c										-	0	0	0	3	0	3
9d											-	0	0	0	0	3
8												-	0	0	0	0
4b													-	0	3	1
5														-	0	0
9e															-	3
7																-

Experiment 6 (Budget 55 to 59)
No. of nodes = 16
APS = 113
Cost = 57

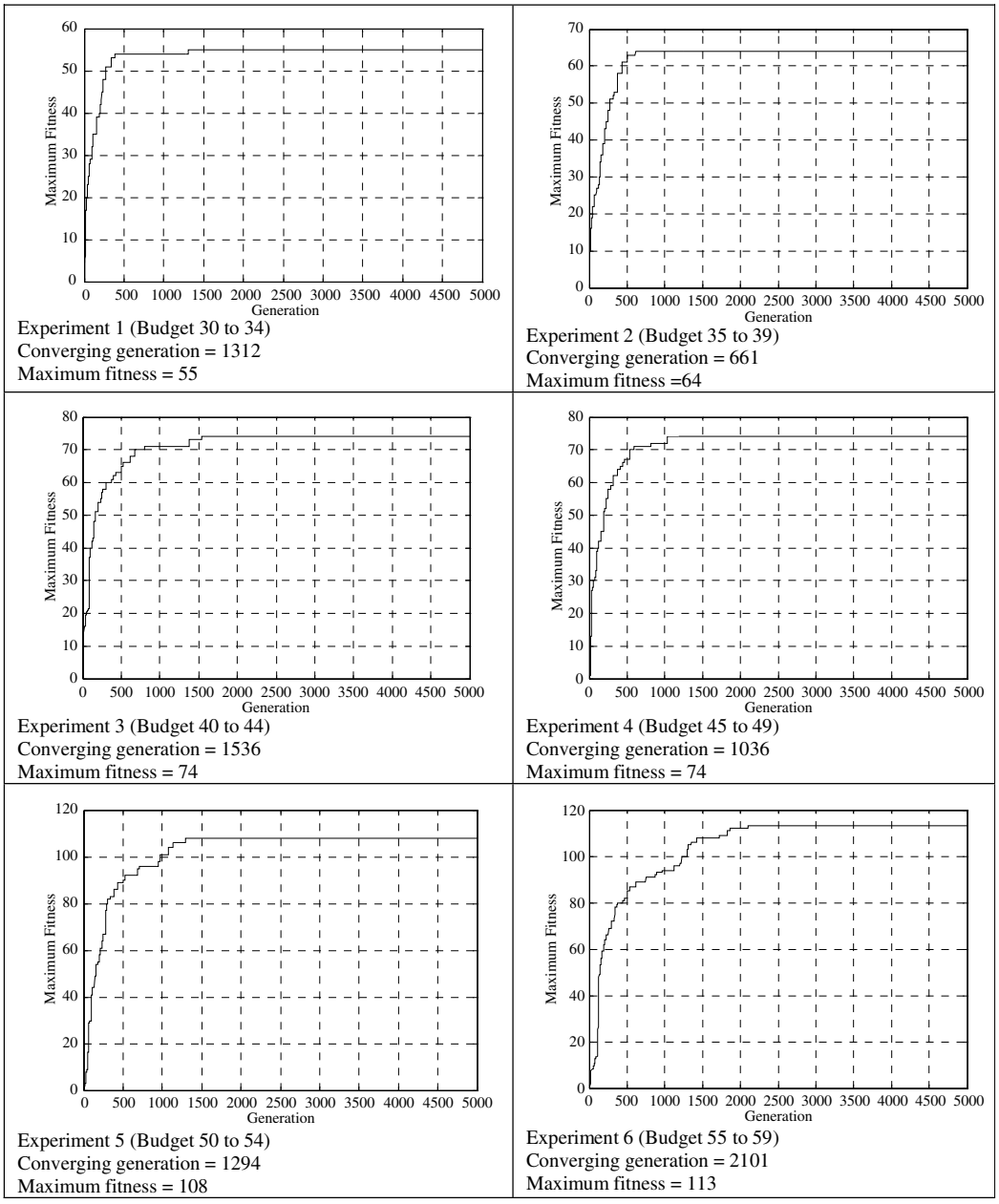


Fig. B.1. Fitness graphs on optimal architectural topologies in different experiments.

Appendix B. Fitness graphs on architectural topology evolution

See Fig. B.1.

References

[1] Goldberg DE. Genetic algorithms in search, optimization, and machine learning. Boston (MA): Addison-Wesley; 1989.
[2] Michalewicz Z. Genetic algorithms + data structures = evolutionary programs. Berlin: Springer; 1996.
[3] Koza JR. Genetic programming II: Automatic discovery of reusable programs. Cambridge (MA): The MIT Press; 1994.
[4] Dodd N. Optimization of neural network structures using genetic techniques Internal Report RIPREP/1000/63/89, Royal Signals and Radar Establishment, Malvern, UK, 1989.
[5] Nolfi S, Parisi D. Genotypes for neural networks. In: Arbib MA, editor. The handbook of brain theory and neural networks. Cambridge (MA): MIT Press/Bradford Books; 1995.
[6] Gruau F. Cellular encoding as a graph grammar. 1993. In: IEE colloquium on grammatical inference: Theory, applications and alternatives. pp. 17/1–1710.
[7] Kitano H. Designing neural network using genetic algorithm with graph generation system. Complex Systems 1990;4:461–76.
[8] Lindenmayer A. Mathematical models for cellular interaction in development I, II. Journal of Theoretical Biology 1968;18:280–315.
[9] Yin S, Cagan J. An extended pattern search algorithm for three-dimensional component layout. Transactions of the ASME 2000;122:102–8.
[10] Shaikh A, Shin Kang. Destination-driven routing for low cost multicast. IEEE Journal on Selected Areas in Communications 2007;15(3):373–81.
[11] Tang M, Yao X. A memetic algorithm for VLSI floorplanning. IEEE Transactions on Systems, Man, and Cybernetics, Part B 2007;37(1):62–9.
[12] Levin PH. Use of graphs to decide the optimum layout of buildings. Architect 1964;14:809–15.
[13] Rücker C, Rücker G, Meringer M. Exploring the limits of graph invariant- and spectrum-based discrimination (sub)structures. Journal of Chemical Information and Computer Sciences 2002;42:640–50.
[14] De Jong KA, Spears W. An analysis of the interacting roles of population size and crossover in genetic algorithms. In: Proceedings of the first int'l conf. on parallel problem solving from nature. 1990.
[15] De Jong KA, Spears W. On the virtues of parameterized uniform crossover. In: Proceedings of the 4th international conference on genetic algorithms. San Mateo: Morgan Kaufmann Publishers; 1991. p. 230–6.
[16] Xu HZ, Wei XY, Xu MS. Schema analysis of multi-point crossover in genetic algorithm. In: Proceedings of the 3rd world congress on intelligent control and automation. 2000.
[17] Gero JS, Kazakov VA. Evolving design genes in space layout planning problems. Artificial Intelligence in Engineering 1998;12(3):163–76.

[18] Gero JS, Kazakov V, Schnier T. Genetic engineering and design problems. In: Dasgupta D, Michalewicz Z, editors. Evolutionary algorithms in engineering applications. Berlin: Springer Verlag; 1997. p. 47–68.
[19] Schnier T, Gero JS. Learning genetic representations as alternative to hand-coded shape grammars. In: Gero JS, Sudweeks F, editors. Artificial intelligence in design. Dordrecht: Kluwer; 1996. p. 39–57.
[20] Rosenman M. The generation of form using an evolutionary approach. In: Dasgupta D, editor. Evolutionary algorithms in engineering applications. Springer-Verlag; 1997. p. 69–86.
[21] Michalek JJ, Choudhary R, Papalambros PY. Architectural layout design optimization. Engineering Optimization 2002;34(5):461–84.
[22] Medjdoub B, Yannou B. Separating topology and geometry in space planning. Computer-Aided Design 2000;32:39–61.
[23] Ruch J. Interactive space layout: A graph theoretical approach. In: Proceedings of the 15th conference on design automation, Issue 19-21, pp. 152–157, June 1978.
[24] Liao CC, Lu HI, Yen HC. Compact floor-planning via spanning trees. Journal of Algorithms 2003;48:441–51.
[25] Koza JR. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems, Stanford University Computer Science Department technical report STAN-CS-90-1314, June 1990.
[26] Yao X. Evolving artificial neural networks. Proceedings of the IEEE 1999;87(9): 1423–47.
[27] Gruau F.. Cellular encoding as a graph grammar. IEE colloquium on grammatical inference: Theory, applications and alternatives, 1993. pp. 17/1–1710.
[28] Luke S, Spector L. Evolving graphs and networks with edge encoding: Preliminary report. In: Late-breaking papers of the genetic programming (GP96) conference, 1996.
[29] Luke S. Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation 2000;4(3):274–83.
[30] Koza JR, Rice JP. Genetic generation of both the weights and architecture for a neural network. IJCNN-91-Seattle International Joint Conference on Neural Networks 1991;2:397–404.
[31] Nikolaev NY, Iba H. Learning polynomial feedforward neural networks by genetic programming and backpropagation. IEEE Transactions on Neural Networks 2003;14(2):337–50.
[32] Xiao L, Liu Y, Ni LM. Improving unstructured peer-to-peer systems by adaptive connection establishment. IEEE Transactions on Computers 2005; 54(9).
[33] Carrano EG, Fonseca CM, Takahashi RHC, Pimenta LCA, Neto OM. A preliminary comparison of tree encoding schemes for evolutionary algorithms. IEEE International Conference on Systems, Man and Cybernetics 2007;7(10): 1969–74.
[34] Miller G, Todd P, Hegde S. Designing neural networks using genetic algorithms. In: Proceedings of the third conference on genetic algorithms and their applications. San Mateo (CA): Morgan Kaufmann; 1989.
[35] Fisher G, Wing O. Computer recognition and extraction of planar graph from the incidence matrix. IEEE Transactions on Circuit Theory 1966;13(2):154–63.
[36] Nagamochi H, Suzuki T, Ishii T. A simple recognition of maximal planar graphs. Information Processing Letters 2004;89(5):223–6.