

# **A Genetic Algorithm Approach to Space Layout Planning Optimization**

Hoda Homayouni

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Architecture

University of Washington

2007

Program Authorized to Offer Degree:

Architecture

University of Washington  
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Hoda Homayouni

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Committee Members:

---

Brian Robert Johnson

---

Mehlika Inanici

---

Carrie Sturts Dossick

Date: \_\_\_\_\_

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature\_\_\_\_\_

Date\_\_\_\_\_

University of Washington

Abstract

A Genetic Algorithm Approach to Space Layout Planning Optimization

Hoda Homayouni

Chair of the supervisory Committee:  
Associate professor Brian R. Johnson  
Department of Architecture

Every architectural design process starts with the schematic design phase, wherein architects have to satisfy a collection of adjacency constraints among spaces and dimensional constraints over each space element. Here, architects face a complicated problem. Some constraints contradict others; priorities may not be clear and the adjacency constraints grow exponentially as the number of rooms in a design problem increases. In large design problems, optimizing such a problem is a time consuming trial-and-error task that could benefit from computational assistance.

Among the different computational methods that have been used in optimization problems, artificial intelligence methods have shown a potential to produce novel optimized solutions. In this thesis, genetic algorithm, one of the powerful search methods in artificial intelligence, is used to create an intelligent prototype to be used in early phases of design. This prototype is able to generate alternative schematic designs to help the architects choose a direction for their design, while having a broad perspective about other good possibilities.

# Table of Contents

	Page
List of Figures.....	iii
List of Tables.....	v
1. Introduction.....	1
2. Background.....	3
2.1. Solution Approaches to Space Layout Planning Automation.....	3
2.2. Genetic Methods.....	7
2.2.1. Genetic Algorithm Approach.....	7
2.2.2. Genetic Engineering Approach.....	9
3. Related Work.....	11
3.1. Procedural Methods- Additive Space Allocation.....	11
3.1.1. History of the program.....	11
3.1.2. Structure of the program (Grason, 1971).....	11
3.1.3. Discussion.....	17
3.2. Procedural Methods- Permutational Space Allocation.....	18
3.2.1. History of the program.....	18
3.2.2. Structure of the program (ACTLOC, 1992).....	19
3.2.3. Discussion.....	21
3.3. Heuristic Methods.....	22
3.3.1. History of the program.....	22
3.3.2. Structure of the program (Arvin and House, 1999).....	23
3.3.3. Discussion.....	26
3.4. Evolutionary Methods.....	27
3.4.1. History of the Program.....	28

3.4.2. Structure of the program (Rosenman and Gero, 1999) .....	29
3.4.3. Discussion .....	38
3.5. Comparison and Results .....	39
4. A genetic engineering approach to Space Layout Planning .....	43
4.1. Implementation and results .....	49
4.1.1. Design objectives .....	50
4.1.2. Design layout.....	50
4.2. Experiments and results .....	55
4.2.1. Experimental objectives .....	55
4.2.2. Experimental setup .....	57
4.2.3. Results .....	64
5. Future Directions .....	68
6. Conclusions.....	70
Bibliography.....	71
Appendix: Application source code .....	75

## List of Figures

Figure Number	Page
<i>Figure 1- Flow chart of a genetic algorithm (Gero and Schnier, 1995)</i> .....	9
<i>Figure 2- Flowchart of a genetic engineering (Gero and Schnier, 1995)</i> .....	10
<i>Figure 3- A typical floor plan (Grason, 1971)</i> .....	12
<i>Figure 4- Floor plan graph with dual graph (Grason, 1971)</i> .....	13
<i>Figure 5- Four planar realizations of a graph (Grason, 1971)</i> .....	15
<i>Figure 6- Output of GRAMPA (Grason, 1971)</i> .....	17
<i>Figure 7- ACTLOC sample output (ACTLOC, 1992)</i> .....	20
<i>Figure 8- Applying topological objectives (Arvin and House, 1999)</i> .....	24
<i>Figure 9- Applying geometrical objectives (Arvin and House, 1999)</i> .....	25
<i>Figure 10- Sample results (Arvin and House, 1999)</i> .....	26
<i>Figure 11-Example of an evolving representation (Gero and Schnier, 1995)</i> . ....	30
<i>Figure 12-Room plans used as design cases (Gero and Schnier, 1995)</i> .....	31
<i>Figure 13- Evolved representation (Gero and Schnier, 1995)</i> .....	32
<i>Figure 14- Using design knowledge from the design cases (Gero and Schnier, 1995)</i> ....	33
<i>Figure 15- Results of Living room Generation (Rosenman and Gero, 1999)</i> .....	36
<i>Figure 16- Results of Living Zone Generation (Rosenman and Gero, 1999)</i> .....	37
<i>Figure 17- Results of House Generation (Rosenman and Gero, 1999)</i> .....	37
<i>Figure 18- Results of the crossover function at the room level</i> .....	44
<i>Figure 19-Crossover function at the building level</i> .....	46
<i>Figure 20- Creating an evolved gene</i> .....	47
<i>Figure 21- An Example of a Mutation process</i> .....	49
<i>Figure 22- An example of a GUI with population of rooms</i> .....	51
<i>Figure 23- An example of a GUI in the room level</i> . ....	52
<i>Figure 24- Adjacency requirements GUI</i> .....	53
<i>Figure 25- An example of a GUI with populations of building layouts</i> .....	54

<i>Figure 26- A building-level GUI that shows the results of creating building layouts.....</i>	<i>55</i>
<i>Figure 27- Best fitness values for different population sizes.....</i>	<i>58</i>
<i>Figure 28- Variations of the best fitness values by increasing the size of selected populations for comparison and making evolved genes .....</i>	<i>59</i>
<i>Figure 29- Variations of best fitness values by increasing the population size.....</i>	<i>60</i>
<i>Figure 30- The graph shows the program runtime verses the population size .....</i>	<i>61</i>
<i>Figure 31- Improving the best fitness value by increasing the number of generations....</i>	<i>62</i>
<i>Figure 32- Variations of the best fitness value by increasing the population size .....</i>	<i>63</i>
<i>Figure 33- A result of running the program with 10 rooms (first example).....</i>	<i>65</i>
<i>Figure 34- A result of running the program with 10 rooms (second example) .....</i>	<i>65</i>
<i>Figure 35- A result of running the program with 10 rooms (third example) .....</i>	<i>66</i>
<i>Figure 36- A result of running the program with 10 rooms (4<sup>th</sup> example) .....</i>	<i>66</i>
<i>Figure 37- A result of running the program with 10 rooms (5<sup>th</sup> example) .....</i>	<i>67</i>

## List of Tables

Table Number	Page
Table 1- Comparison chart of strengths and weaknesses of the studied approaches.....	40
Table 2- "add2right" function .....	75
Table 3- "add2up" function.....	75
Table 4- "addLabel" function.....	76
Table 5- "Adjacancies_GUI" function .....	77
Table 6- "arrangedBlock3" function .....	84
Table 7- "buildBlock" function.....	85
Table 8- "buildPerimeter" function.....	87
Table 9- "checkBlockNames" function.....	89
Table 10- "checkBlocks" function .....	90
Table 11- "CloseFigure" function.....	91
Table 12- "combinationFitness" function .....	91
Table 13- "combiningRooms" function .....	92
Table 14- "CombsAreaFitness" function .....	95
Table 15- "CombsProFitness" function .....	96
Table 16- "CombsSmoothFitness" function .....	96
Table 17- "convexityFitness" function .....	97
Table 18- "CreatBuilding_GUI" function.....	97
Table 19- "CreatePopulation" function.....	100
Table 20- "creatingRooms" function .....	101
Table 21- "CreatingRooms_GUI" function .....	102
Table 22- "CrossOverCombinations" function.....	105
Table 23- "CrossOverFen" function .....	111
Table 24- "DeleteRoom" function .....	112
Table 25- "FirstPopulation" function.....	114
Table 26- "isContinuous" function .....	114

Table 27- “isOutSide” function .....	116
Table 28- "leftSideRight" function .....	116
Table 29- “minimizingArea” function .....	117
Table 30- “moveRoom” function.....	117
Table 31- "mutate_permutation3”function .....	118
Table 32-“plotBlock” function .....	119
Table 33- “plotCombinations” function.....	119
Table 34- “plotRooms” function.....	119
Table 35- “proportionFitness” function .....	120
Table 36- “putToGether” function.....	120
Table 37- “SelectingTheBest” function .....	122
Table 38-“shiftRoom” function .....	123
Table 39-“smoothnessFitness” function .....	124
Table 40-“SmoothPerimeter” function .....	124
Table 41- “turnBlock” function .....	124
Table 42- “upSideDown” function .....	125

## **Acknowledgements**

In this special moment of my life, I stop for a while and look back to the past; to what I have done and how I have accomplished that. I find myself too small for relating these achievements to myself. It is the almighty God who has given me the passion and power, and he is the one that I praise and thank the most. He is the one who surrounded me with caring and thoughtful people. People, that I would have not achieved many of my accomplishments without their wisdom and endless love. It is not possible for me to state all of their deeds that I am thankful for and show how thankful I am. I therefore limit my great acknowledgements into few lines, and ask the almighty God to bless them with peace and love.

First of all, it is a pleasure to state my gratitude to my supervisor, Prof. Brian Johnson, for his continuous help and the invaluable experiences he shared with me throughout the project. I am also grateful for the inspirations of Dr. Mehlika Inanici in the beginning of my research and her enlightening comments throughout the documentation. My deep appreciations belong to Dr. Carrie Dossick for her great support in my carrier and her insightful contributions to the thesis.

I would also like to thank my friends and colleagues in the Design Machine Group for providing such a warm, cooperative and stimulating environment for studying. Thanks guys for all the good moments we shared together.

Words fail to express my sincere appreciations to my parents who are always my sense of safety and calmness. Thank you for giving me everything in life and for being so loving, caring and supportive.

My last but not least gratitude belongs to my beloved husband. Dear Sauleh, I would like to especially say thanks to you for stepping into my life and making a big difference. I am deeply thankful for your continuous love and invaluable support and help. Thank you....

# Dedications

To my parents

and

dear Sauleh

## 1. Introduction

Space layout planning is one of the most challenging phases of architectural design, especially in complex projects. Putting together all the rooms and spaces of the building according to hundreds of relationships between spaces, and many functional and aesthetic factors is not a trivial job. Architects are usually dealing with a problem that is not fully formulated and thus is ill-defined (Yoon, 1992). Resolving ill-defined problems is a cumbersome process of searching for a set of design constraints, satisfying and then refining them. These problems do not have a single best solution (Balchandran and Gero, 1987) and thus architects always are looking for better and more feasible solutions.

While there are many CAD systems that are very good at producing precise, accurate renderings of well defined designs and many include detailed analysis packages for obtaining simulation feedback on the performance of a design, very few address the need of a designer during the initial conceptual exploratory phase of design (Michalek, 2001). This is mostly due to the ill-defined nature of the design process and dealing with non-quantifiable qualities such as aesthetic values. However, if we can find a way to take advantage of the excellent rational and search abilities provided by computers, while using a human's creativity and intuition needed to solve problems, we will have a system that designs more efficiently and more accurately.

Yet, this is not simple. Automated space planning systems need a method of producing a good solution from a large set of possible solutions, and a method of allowing the designer to modify the set of design constraints to continually refine the problem definition (Arvin and House, 1999). Tsang et al. have shown that there does not appear to be a universally best algorithm and that certain algorithms may be preferred under certain circumstances (Tsang et al., 1995).

In this thesis, among the many approaches investigated, a genetic algorithm approach (introduced in section 2.2) has been selected and implemented.

## 2. Background

The following paragraphs provide an introduction to space layout planning and a survey of computational approaches to the problem. For each solution capabilities and drawbacks of the approach and the methods of contribution toward the design process are demonstrated. The method selected as a platform for further investigations is introduced at the end of the section.

### *2.1. Solution Approaches to Space Layout Planning Automation*

Architectural space layout planning is the process of allocating a set of space elements according to certain design criteria. Before categorizing the solution approaches to space layout planning and consequently different programs that are presented and studied so far, it is worth mentioning that all of these different approaches begin by propounding numerical definitions for all the constraints existing in the design situation. There are basically two types of constraints that express the design objective requirements:

- Dimensional constraints (geometrical constraints): constraints that are considered over one place, i.e. constraints on surface, length or width, or space orientation.
- Topological constraints: constraints that are considered over a couple of spaces, i.e. adjacency between the rooms, adjacency to perimeter of the building, non-adjacencies or proximities. The numerical criteria that topologically define arrangements usually come from site analysis questions and user need questions.

Therefore, attempt to solve space layout planning problem usually results in geometrical and topological relationships between elements (Jo, 1996).

Kalay (2004) categorizes computational design synthesis methods as:

- Procedural Methods
- Heuristic Methods
- Evolutionary Methods

Examples of each type of floor plan layout approach, along with discussions about their advantages and drawbacks, are presented in chapter 3. However, these three methods are described here.

First, Procedural Methods leverage our ability as human designers to specify local conditions and the ability of the computer to apply or test for these relationships over much larger sets of variables. The basic procedural approach is to completely enumerate all the possible arrangements of floor plans from a given set of rooms. Then, architects can choose the most appropriate one from those alternatives for a given design project. However, the numbers of possible combinations, or solutions, expand dramatically as the number of design parameters is increased. Therefore, it is an inefficient approach for computers to try to calculate all possible solutions. Even if a computer can generate a large number of possible solutions, no architect has sufficient time and energy to review all those solutions (Kalay, 2004).

Another procedural approach to arrange rooms in a floor plan is to enlist the services of the computer in the layout of spaces in a building according to some rational principles (mostly minimization of distances between spaces that ought to be close to each other). This approach is known as “Space Allocation”. The uses of space allocation approaches however are limited to building types where the main factor in their design is

distances between the related spaces (e.g., schools, hospitals and warehouses) (Kalay, 2004).

Attempts to improve space allocation with the help of procedural methods continued by including additional design criteria (e.g., lighting, privacy and orientation) in the decision-making process of the placement algorithm. Different “Constraint Satisfaction” methods can be introduced to include multiple objectives in space allocation. As Kalay examined, with some exceptions, the results of space allocations with constraint satisfaction methods were poor compared to the results obtained by competent architects. In fact, resolving more constraints with satisfying results needs the more heuristic methods of simulation (Kalay, 2004).

Second, Heuristic Methods are the computational design methods that are inspired by analogy, similar to the habitual design synthesis methods that are typically inspired by analogies and guided by the architect’s own or another designer’s previous experiences. These methods rely on personal and professional expertise accumulated over a lifetime of confronting a variety of design issues. Unlike procedural methods, heuristic methods cannot guarantee that they arrive at a solution to the problem, nor that a solution they arrive at will be optimum necessarily. Their reasoning is not exact, and the knowledge on which they rely may contain logical gaps and inconsistencies. Yet, despite these shortcomings, or perhaps because of them, heuristic methods are able to solve problems that procedural methods can not (Kalay, 2004).

One of the most common heuristic methods for synthesizing design solutions is to borrow rules to approach the problem from other knowledge areas that appear to hold some relevance to the problem at hand. These methods are known as Analogical Methods (an example is discussed in section 3.3). Another approach was to start with an old solution to a similar design problem and adapt it to the needs and circumstances of the new problem where it differs from the other one. These methods are known as Case-

Based Methods. Expert systems, the third subcategory of the Heuristic Methods, capture the knowledge and expertise of past designers directly in the form of design rules (Kalay, 2004).

In the late 1970's, researchers recognized the familial nature of the case-based methods and the kit-of-parts nature of the expert systems as kinds of "languages" and formalized them into systems of rule-based geometrical constructions for designing shapes based on strict compositional rules. This approach came to be known as the shape grammar methods. All of these heuristic methods were able to solve design problems that procedural methods cannot. However none of their resultant design solutions would be considered as a novel solution (Kalay, 2004).

Third, the development of artificial intelligence (AI) strategies provided architectural researchers with new methods and tools in their quest to rationalize the design synthesis process. AI's heuristic approaches to solving problems, known as Evolutionary Methods, seemed to have the potential to produce novel design solutions. Evolutionary methods are often superior to other search algorithms for problems consisting of large unstructured search spaces, where no heuristic method is able to guide the search. However, the search time for evolutionary systems depends greatly on efficient coding (Rosenman and Gero, 1999).

Genetic Algorithms (GAs) are one of the evolutionary methods that are able to generate unexpected, novel forms. This technique has potential to evolve solutions to real world problems by imitating the Darwinian idea of survival of the fittest. (Mitchell, 1996). This method is described in depth in section 2.2.

Another approach in Evolutionary Methods, named Artificial Neural Networks (ANNs), are derived from the cognitive theory of connectionism, which argues for processing information as patterns, by means of networking many independent small signal processors, rather than by executing explicit instructions one at a time. The pattern

recognition ability of ANNs makes them suitable for the task of recognizing innovative design solutions (Kalay, 2004).

## ***2.2. Genetic Methods***

In this section computational methods that were used in implementing the prototype program developed during this research are examined more closely.

### **2.2.1. Genetic Algorithm Approach**

Genetic Algorithms are adaptive methods which are used to solve search and optimization problems. They are based on the genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and “survival of the fittest”. By mimicking this process, genetic algorithms are able to “evolve” solutions to real world problems, if they have been suitably encoded (Mitchell, 1996).

The basic principles of GAs were first laid down rigorously by John Holland (Holland, 1975). GAs simulate those processes in natural populations which are essential to evolution. Exactly which biological processes are essential for evolution, and which processes have little or no role to play, is still a matter for research but the foundations are clear (Mitchell, 1996). In nature, individuals in a population compete with each other for resources such as food, water and shelter. Also, members of the same species often compete to attract a mate. Those individuals which are most successful in surviving and attracting mates will have relatively larger numbers of “offspring”. Poorly performing individuals will produce few or even no offspring at all (Mitchell, 1996).

The combination of good characteristics from different ancestors, along with random changes, called mutation, can sometimes produce offspring whose fitness is

greater than that of either parent. In this way, species evolve to become more and better and better suited to their environment.

Genetic algorithms require that appropriate design parameters be identified and encoded as the unique characterization, or genome, of an individual solution. Each possible solution is then a permutation on this parameter space. However, this space is not searched randomly, nor is it searched exhaustively. It is searched according to the notion of fitness or survivability in the problem requirements environment.

GAs use a direct analogy of natural behavior. They work with a population of individuals, each representing a possible solution to a given problem. Each individual is assigned a “fitness score” according to how good a solution to the problem it is. (In nature this is equivalent to assessing how effective an organism is at competing for resources.) The highly fit individuals are given opportunities to “reproduce”, by “cross breeding” with other individuals in the population. This produces new individuals as offspring, which share some features taken from each parent. The least fit members of the population are less likely to get selected for reproduction, and so they get eliminated from the population (Mitchell, 1996).

A whole new population of possible solutions is thus produced by selecting the best individuals from the current generation, and mating them to produce a new set of individuals. This new generation contains a higher proportion of the characteristics possessed by the good members of the previous generation. In this way, over many generations, good characteristics are spread throughout the population, being mixed and exchanged with other good characteristics as they go. By favoring the mating of the more fit individuals, the most promising areas of the search space are explored. If the GA has been designed well, the population will converge to an optimal solution to the problem (Mitchell, 1996). Although GAs are not guaranteed to produce the global optimum

solution to a problem, but they are generally good at finding acceptably good solutions to problems in a reasonably time frame (Mitchell, 1996).

A flowchart of a general genetic algorithm method is shown in Figure 1:

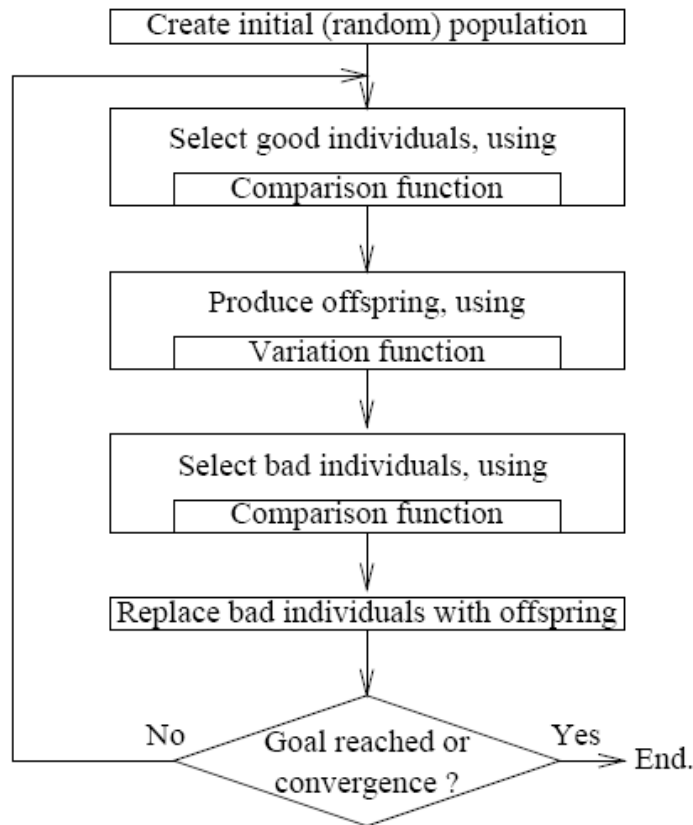


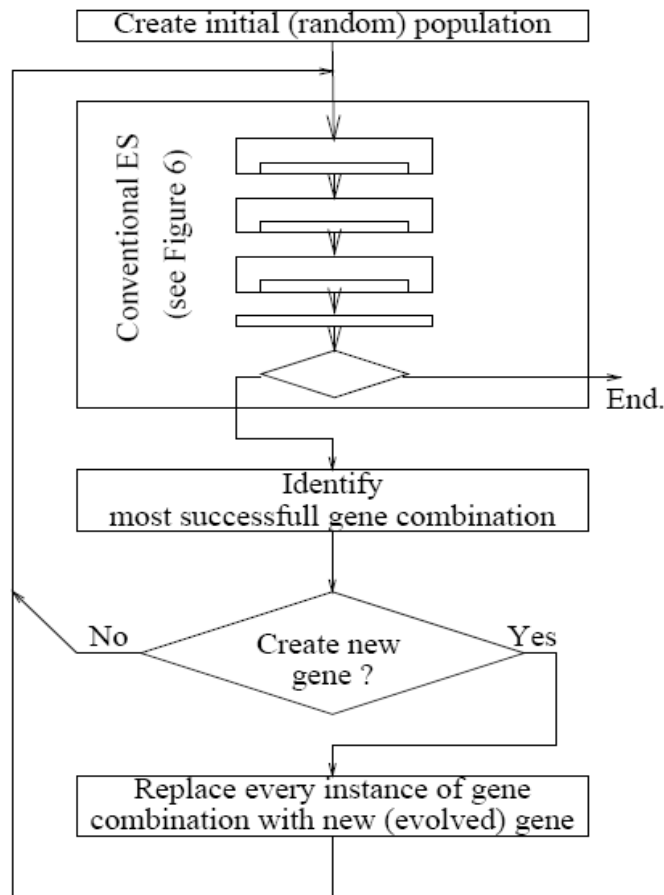
Figure 1- Flow chart of a general genetic algorithm (Gero and Schnier, 1995)

### 2.2.2. Genetic Engineering Approach

The idea of genetic engineering in the context of GA optimization has been presented in several papers (Gero and Schnier, 1995; Cervone et al., 2000). The basic idea is to use machine learning to identify some sort of hypothesis to characterize good

and bad individuals. The hypothesis can then be instantiated to propose better individuals. A genetic engineering routine can also be invoked to propose new individuals instead of a genetic operator (Rashid, 2000).

A flowchart of a general genetic engineering approach is depicted in Figure 2.



**Figure 2-** Flowchart of a general genetic engineering approach (Gero and Schnier, 1995)

### **3. Related Work**

Reported attempts to automate the process of space layout planning started over 35 years ago (Levin, 1964). As indicated previously, researchers have used several problem representations and solution search techniques to describe and address the problem. Some of the main examples of this work, including descriptions of applications, are presented here categorized according to Kalay's classification (2004).

#### ***3.1. Procedural Methods- Additive Space Allocation***

An example of a program that has approached the space layout problem by means of procedural methods is GRAMPA (for GRAPh Manipulating PAcKage). The type of procedural method implemented in this case is additive space allocation, described in section 2.1.

##### **3.1.1. History of the program**

The use of linear graphs for representing floor plans is a technique in space planning that received a lot of attention in 70's and early 80' decades (Levin, 1964; Casalaina and Rittel, 1967; Krejcirik, 1969; Grason, 1970a, 1970b, 1970c; Teague, 1970). The current GRAMPA program is the product which was presented by John Grason, at Carnegie-Mellon University, in 1971.

##### **3.1.2. Structure of the program (Grason, 1971)**

Grason's approach to computerized space planning is based on the methods of solution for the formal class of floor plan design problems. The methods of solution

depend on a special linear graph representation for floor plans called the ‘dual graph’<sup>1</sup> representation.

As shown in Figure 3 a “space” is defined to be either a room or one of the four outside spaces of north, south, east and west. A problem statement consists of a set of adjacency and physical dimension requirements that have to be satisfied, and a problem solution is a floor plan that satisfies all of the design requirements.

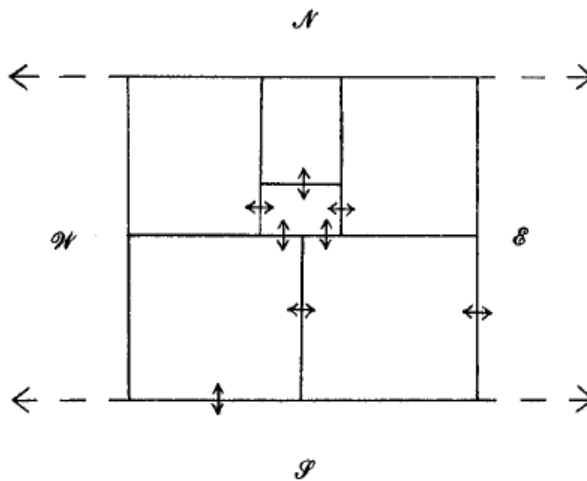


Figure 3- A typical floor plan (Grason, 1971)

In applying graph theory to floor plan layout, rooms are pictured as labeled nodes possessing certain attributes, such as intended use, area, and shape. Adjacencies between rooms are indicated by drawing lines (edges) connecting the nodes to the corresponding rooms. These notions can be implemented by dealing with the dual graph

---

<sup>1</sup> In mathematics, a *dual graph* of a given planar graph  $G$  has a vertex for each plane region of  $G$ , and an edge for each edge joining two neighboring regions. The term "dual" is used because this property is symmetric, meaning that if  $G$  is a dual of  $H$ , then  $H$  is a dual of  $G$ ; in effect, these graphs come in pairs.

of a floor plan which is itself treated as a linear graph. An example of such a floor plan graph is shown in Figure 4, with black nodes. In the floor plan graph, “edges” and “nodes” will be called “wall segments” and “corners” respectively.

A special dual of the floor plan graph can be obtained by placing a node inside each space and constructing edges to join the nodes of adjacent spaces. This special type of dual graph of the floor plan is the design representation to be used for the class of problems described in this paper. The general idea of its application is to first set down the four nodes and four edges of the dual graph that represent the four outside walls of a building. Then nodes and edges are added one by one to the dual graph in response to design requirements and other considerations until a completed dual graph is obtained.

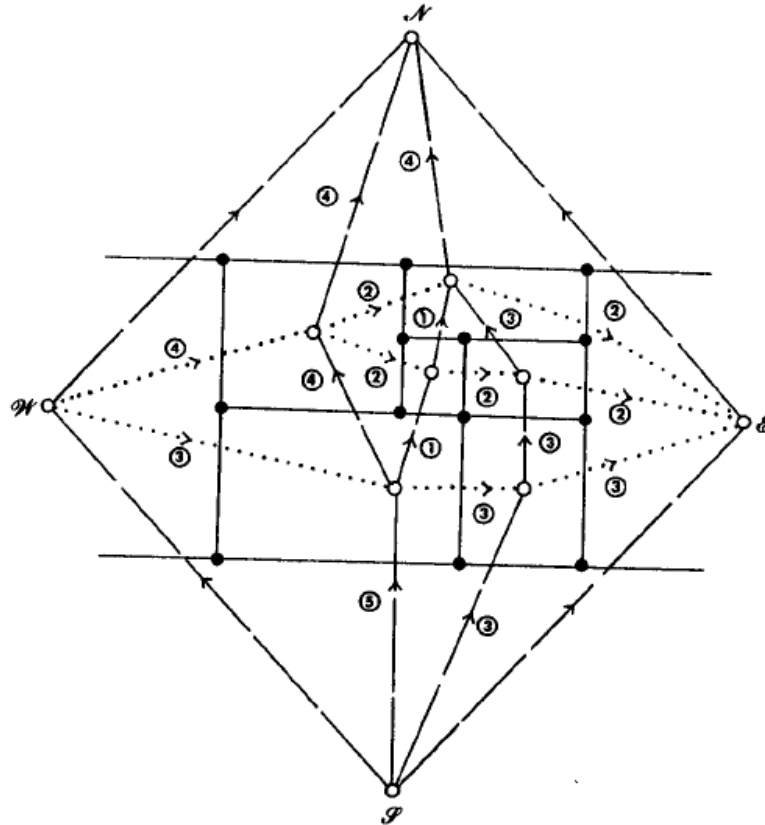


Figure 4- Floor plan graph with dual graph (Grason, 1971)

The incomplete dual graphs that are produced in the intermediate stages of this design process present special problems. Since edges can be colored, directed, and weighted, it is not always clear whether or not there exists at least one physically realizable floor plan satisfying the relationships expressed in the incomplete dual graph. To treat this problem, appropriate properties of the dual graph representation have been developed and are presented in Grason's paper. These include the definitions of "Planarity", "Well-Formed Nodes", "Well-Formed Terminal Regions" and "The Turn Concept". Based on these properties three theorems on physical realizability are established.

The use of these theorems enables the program to detect whether the graph is planar or not. It also makes it possible to generate various possible geometric realizations of the dual graph. A geometric realization of a planar graph is simply one of the possibly many ways in which it can be drawn in a plane. Four different realizations of a particular planar graph are shown in Figure 5.

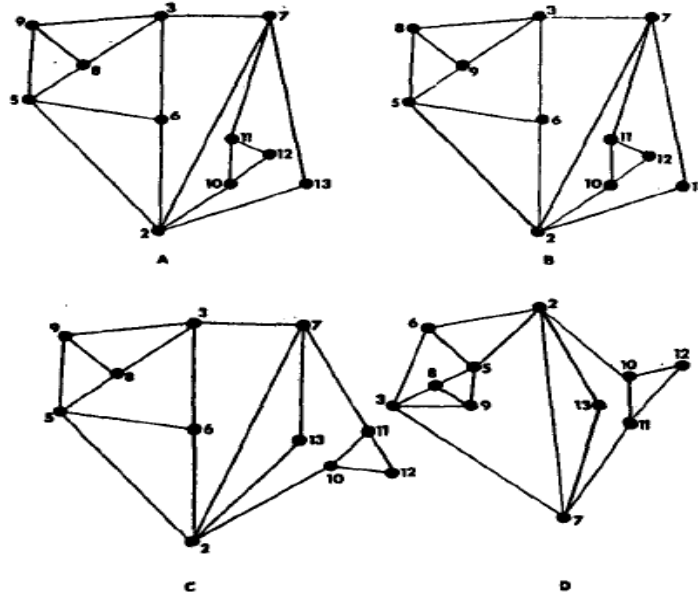


Figure 5- Four planar realizations of a graph (Grason, 1971)

Since in a partially completed dual graph, each planar realization corresponds to a topologically different set of floor plans, it is important to be able to generate realizations with great facility.

To deal with the problems mentioned above, a special graphical grammar called the planar graph grammar (PGG) was developed. The general intent of this grammar is to divide a graph into a set of hierarchically organized sub graphs. Each of these sub graphs can be treated as a separate entity, and if it is connected to the rest of the graph, it is by at most two nodes. These sub graphs are free to rotate around these nodes to create the various realizations of the graph. Other similar “degrees of freedom” in creating realizations are also possible.

The grammar allows the user to deal with planarity in an inductive manner. Given a graph that is planar and described in terms of the planar graph grammar, if a new

edge is to be added to that graph, a test can be developed to tell whether the resultant graph will also be planar. This test assumes that each of the special sub graphs is already planar. It then checks to see if the proposed edge can be connected to the two endpoints without crossing an edge of one of these sub graphs. This method is extremely fast, and its computation time increases only linearly with the size of the graph considered. The computation time for other methods generally increases more than linearly with the size of the graph.

At the next level, the program has two jobs to accomplish. (1) It must satisfy the physical dimension requirements. It does this by selecting nodes on the boundary of the region in question and assigning dimensions to them from the design requirement list. (2) It must “fill” the region with edges representing those adjacencies not specifically requested by the design requirements. Here the program does an exhaustive search of all possible design solutions. The output of the program is shown in Figure 6 .

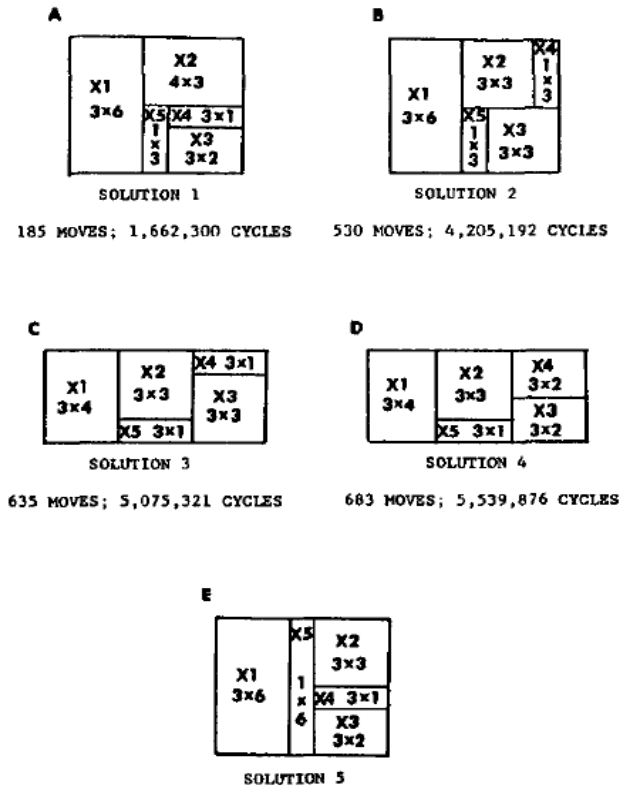


Figure 6- Output of GRAMPA (Grason, 1971)

### 3.1.3. Discussion

Several points should be considered about this prototype. First, the adjacency data provided for this program must be either true or false. This means priorities are not defined in this program. This may constitute a problem in that the program may start processing and eventually may conclude that no planar graph could be drawn based on the input data. In other words, the program does not guarantee to generate a representation even equivalent to a bubble diagram.

Additionally, the long run time of the program hampers the possibility of fixing the input data in multiple runs in an effort to reach to an improved solution.

As mentioned for the case of space allocations, these approaches are often unable to consider more than adjacency objectives. This reduces utility of the program to specific types of buildings (e.g. schools, warehouses).

Not only do these approaches not consider qualitative objectives such as view, but also they usually prevent the main activities from having such advantages. This happens because most highly connected spaces usually end to be placed in the middle of the layout. While this particular program allows users to set the connectivity of spaces with outside regions to true, this decreases the chance of the program to find a planar graph.

Lastly, an exhaustive generation of the floor plan that the program does at the end is one of the weak points of the GRAMPA. As mentioned in the paper (Grason, 1971), it could be replaced with a heuristic method. Otherwise, eliminating that part of the program and ending up with a bubble diagram representation may be more practical.

In sum, as the author also states (Grason, 1971), due to the shortcomings discussed above, GRAMPA is basically a research prototype rather than a practical method to generate automated plan layouts.

### ***3.2. Procedural Methods- Permutational Space Allocation***

ACTLOC (1992) is another example of a program that attempts to computerize space layout planning with procedural methods. The type of applied procedural method in this case is permutational Space Allocation.

#### **3.2.1. History of the program**

In 1965, Thomas Anderson, a student in Civil Engineering at the University of Washington, wrote a program called SLAP1. SLAP1 locates activities in relation to one another in order to minimize the total cost of circulation between the two activities. (This

program is available in a rewritten form as ACTLOC2). SLAR, the direct predecessor of ACTLOC, was written by Dale Bryant, a student in the Department of Architecture at the University of Washington, in December 1967. ACTLOC was substantially rewritten in a modern FORTRAN in 1984, at which time its output and input ‘friendliness’ were improved, and the code was optimized to minimize the cost of execution at the same time as it was enlarged to handle larger projects. This revised program was ported to the VAX/VMS environment about a year later, where it was installed as an interactive, but still file-oriented program. This is the form of the program that is discussed here. (ACTLOC, 1992)

### **3.2.2. Structure of the program (ACTLOC, 1992)**

ACTLOC is a program that provides activity location guidance appropriate to the schematic design phase of a project. It attempts to produce the optimum relationship amongst a number of separate activities. The outputs of this program look pretty similar to bubble diagrams.

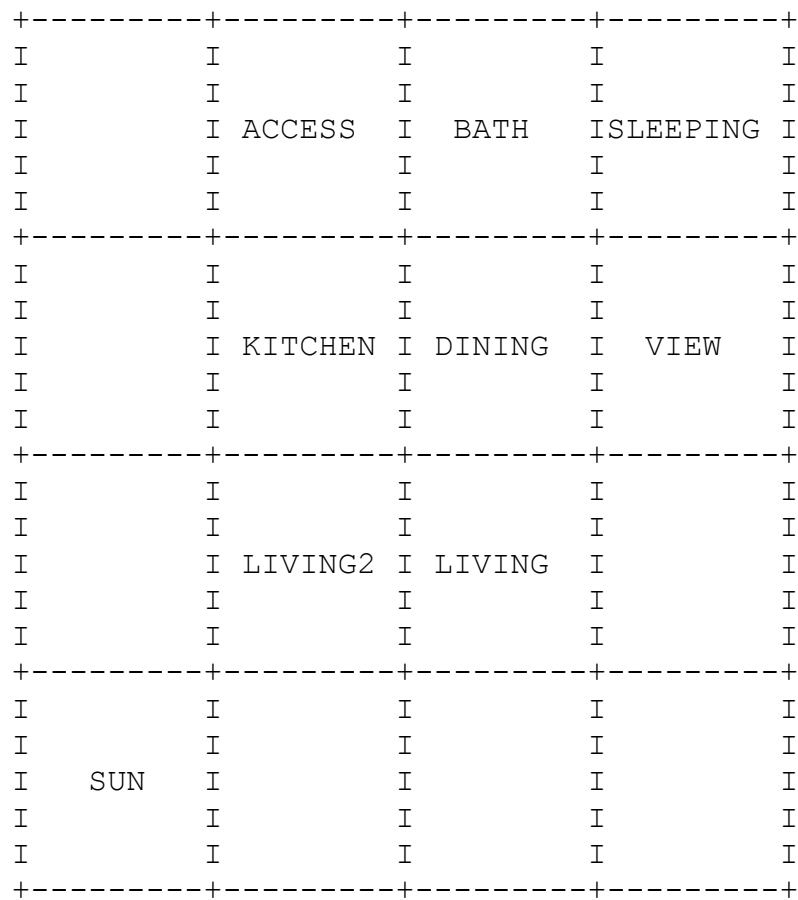
ACTLOC performs the initial layout function based on ‘compatibility’ between activities. Compatibility is calculated from input data in which the needs of each activity have been established in terms of a number of parameters (architectural ‘means’). The procedure which the program follows is based on two assumptions:

- Compatibility between any pair of activities can be considered to be a function of their degree of similarity in terms of the architectural parameters or means which have been entered (i.e., two quiet activities are compatible)
- Basic schematic arrangement can be based on these compatibilities. (i.e., it makes sense to place quiet activities near each other).

ACTLOC calculates the optimum configuration by systematically shifting activities around, keeping the arrangement which results in an improvement in the overall

compatibility; then shifting activities again, starting from the current best. When no more rearrangements can be found to produce more improvement the program stops. Output comes in the form of printed grids showing the relative positions of each activity in the final, optimum positions discovered by the program Figure 7.

TOTAL CONFLICTS=1313.50



**Figure 7- ACTLOC sample output (ACTLOC, 1992)**

Based on this approach this program may not examine a configuration which might be better than the one it considers ‘the best’. In order to improve the chances of

finding the optimal solution the program can randomly rearrange the best and submit that new arrangement to the same sequence of searches as before.

An important consideration about this program is to assign about 1.5 times the number of activities which are identified to the grid cells. That means half of the grid cells should be empty. Considering these empty cells are important because they allow the overall configuration to take on a shape other than the rectangular shape of the grid itself by acting as buffer space around the margins. On the other hand, since the computational cost of running the program is proportional to the size of the grid, beyond a certain size of grid, the extra space adds nothing but cost.

To be able to account for some external conditions like quality of view or solar exposure, a mean for that specific condition should be established (e.g. “SUN” in Figure 7) and placed in its related part of the grid as a “fixed” (immobile) activity. Then, for the activities which would benefit from such exposure, the compatibility rating should be set very high. An example of considering external conditions is depicted in Figure 7 by assigning a south-west grid to the sun.

### **3.2.3. Discussion**

There are several concerns about using this prototype. First, considering the size of activities is important even in producing bubble diagrams. Refusing to consider this important part of design may result in diagrams that are not realizable by assigning the real sizes to them. This could possibly lead to misconduct in the real design situation.

However, as indicated in the program user guide and illustrated in Figure 7 (ACTLOC, 1992), if proportional numbers of grids are specified for different activities based on their expected size (and “bound” together), it is possible to apply an area interpretation to the grids.

On the other hand, applying the size of activities to the program not only increases the cost of the program dramatically, but also prevent activities from freely shifting around due to the fact that only two grid cells exchange their activities at the same time.

Another concern about the program is that it may be trapped in a local optimum, i.e. the program may reach to a layout that is not optimal but cannot be further improved by changing the activities of the two grids. As it is pointed out in the program user guide, in order to improve the chance of finding the best solution, it is possible to run the program in multiple times.

Yet, even if the program reaches the best solution, that doesn't indicate that the best presented solution by the program is the optimum or even semi-optimum practical solution to the real architectural problem. Because there are many other factors (e.g. style) that could not be simulated with the ACTLOC.

### ***3.3. Heuristic Methods***

One of the approaches to computerized space layout planning that is presented by means of analogical methods in the category of heuristic methods is the physically based space planning program, which has borrowed a metaphor from (Mitchell, 1996) mechanical forces and has simulated the problem according to the rules of classical physics.

#### **3.3.1. History of the program**

Physically based modeling was first used to model the realistic behavior of rigid bodies (Barzel and Barr, 1988; Baraff, 1989), deformable models (Terzopoulos et al., 1987), and flocking behavior of a large number of objects (Reynolds, 1987).

House and Kocmoud (1998) use physically based modeling techniques to create what they call “continuous cartograms”.

Physically based dynamics have also been used in geometric design. Qin (Qin et al., Qin and Terzopoulos, 1995; Qin, 1998) and Mandal et al. (Mandal et al., 1997) used physically based techniques to manipulate smooth surfaces of arbitrary topology interactively.

However, Scott A. Arvin and Donald H. House (Arvin and House, 1999) were the first to use physically based techniques to manipulate space layout planning.

### **3.3.2. Structure of the program** (Arvin and House, 1999)

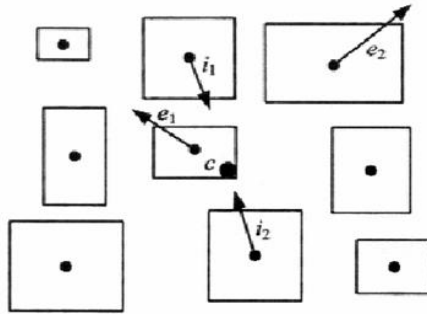
In the physically based space planning program, the designer creates a space plan by specifying and modifying graphic design objectives rather than by directly manipulating primitive geometry. The plan adapts to the changing state of objectives by applying the principles of classical physics to its elements.

In fact, in this approach the architect defines programmatic objectives in the usual manner, and these objectives are then modeled as physical objects and forces used in a dynamic physical simulation.

The spaces and walls are modeled as point masses, with adjacencies between spaces that are modeled as springs connecting the masses. Objectives specified in the architectural program are translated into forces applied to the masses.

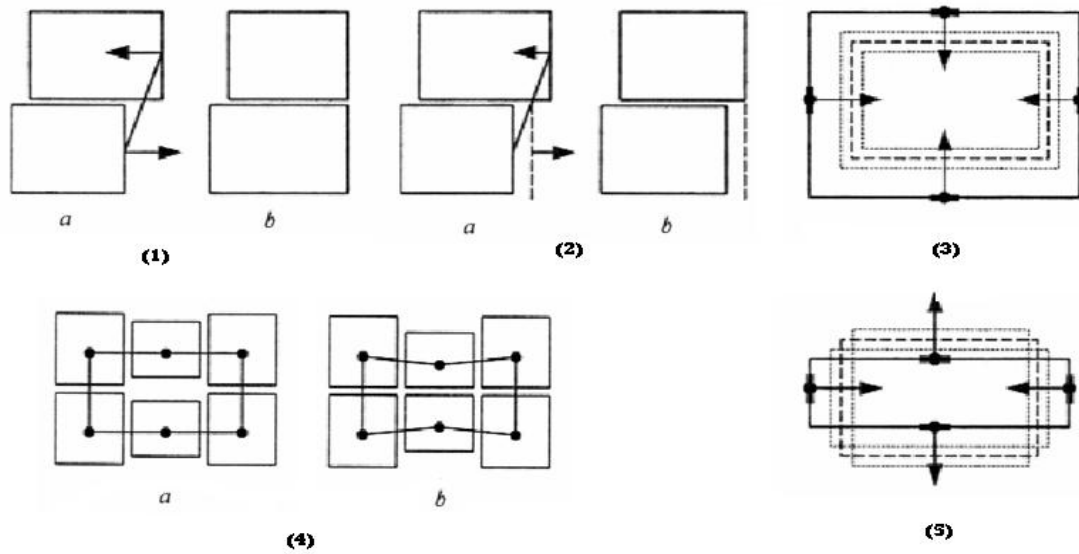
In the first phase the topological objectives apply forces to the center of a space. For collision detection, in this step boundary shapes are treated as circles so spaces are able to slide around each other. The dynamic simulation starts to run until the system is in equilibrium, which is defined as the point in time when the magnitudes of all velocities

are below a small threshold. Examples of the topological objectives are considered in this phase are adjacency, orientation, interior, exterior, and separation objectives.



**Figure 8- Applying topological objectives to center of spaces (Arvin and House, 1999)**

At the second phase topological objectives are turned off and geometric objectives apply forces to the polygonal edges of space boundaries. Also space boundaries are switched from a circular to a polygonal representation in this phase. Collision detection and response then act to keep spaces from overlapping, resulting in an arrangement that is very close to a recognizable building floor plan. Examples of geometrical objectives that are turned on in this phase are alignment, offset, area, gravity and proportion objectives.



**Figure 9- (1) Alignment objectives (2) Offset objectives (3) Area objectives (4) Gravity objectives (5) proportion objectives (Arvin and House, 1999)**

Once a geometric simulation has reached equilibrium, the designer can begin to analyze and interact with the design by directly manipulating the graphic model rather than by respecifying design objectives in the language of the underlying system. The mass-spring representation allows the graphic model to adapt to those changes immediately. The intention here is not simulating the actual behavior of building elements, but simulating the way architects may view and interact with design elements during their conception. Sample pictures for the process are shown in Figure 10.

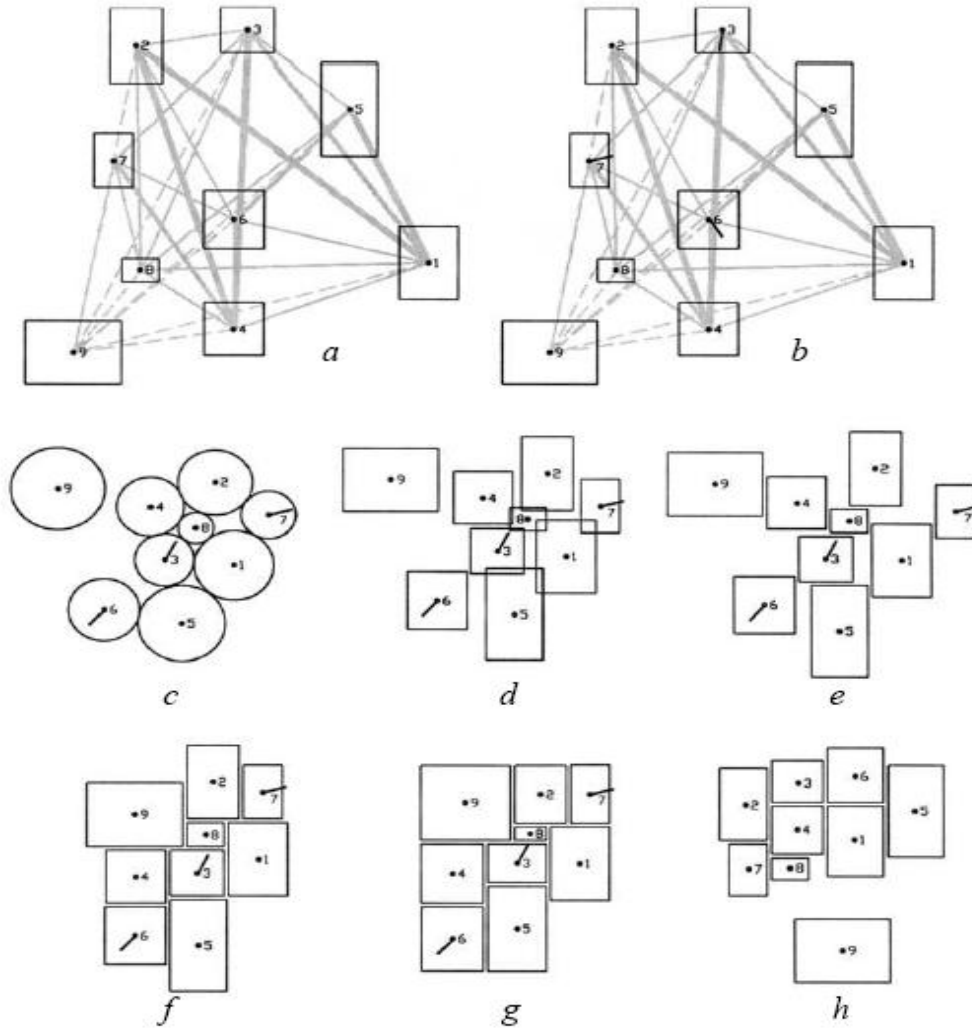


Figure 10- Sample results (Arvin and House, 1999)

### 3.3.3. Discussion

It seems that this prototype system provides a convincing demonstration of the use of physically based design objectives. It appears to help architects to feel the nature of design problem, interacting with this prototype.

However, when borrowing a metaphor from other disciplines, it is always questionable whether the metaphor fits in the problem, and why? How can we make sure that our metaphor doesn't have an aspect that interferes with our problem? Specifically speaking about this case, one may question the similarity of spring forces with design constraints. How should we define the constant of springs proportional to our constraint importance? Is it just a simple proportion or a complicated equilibrium? How do we know that the behavior of spring in all situations is the same as what we need for manipulating the design objectives? These are the questions that need to be addressed.

Yet, since the design problem is always over-constrained (Yoon, 1992), and it is never possible to obtain all the imaginable goals for one design problem, we should ignore some aspects of the design. From this point of view, the fact that this prototype is able to somehow manipulate different constraints of the design, and produce an adequate design solution, makes it valuable.

Nonetheless, this program is still not a truly practical one. There are many other design criteria that should be considered in the program (e.g. cost, circulations, and aesthetics). Some of them may need new heuristic methods to be incorporated into space layout planning, and some just need additional information to be resolved.

In spite of the ability of the program to handle multiple constraints, (in contrast to the space allocation methods), this program is unable to handle multistory building designs. This is not a trivial problem. Additional research is needed to solve these types of design problems.

### ***3.4. Evolutionary Methods***

An example of an approach to computerized space layout planning by means of evolutionary methods is presented by Rosenman and Gero in 1999. This approach benefits from the combinatory power of genetic algorithm and genetic engineering.

### 3.4.1. History of the Program

In 1995 John Gero and Vladimir Kazkov presented a formal evolutionary model of design representations based on genetic algorithms. They used pattern recognition techniques to execute aspects of the genetic engineering (Gero and Kazkov, 1995). Subsequently, Gero and Thorsten Schnier presented a sufficient method to adapt a predefined set of design parameters to automate the design task based on the genetic algorithm (Gero and Schnier, 1995). Later they proposed a different approach, in which a system is given design examples, and in a bottom-up process, it learns stylistic features of the examples. This was achieved by using genetic algorithm and shape grammars that enable the system to change its representation. With the creation of a more and more complex evolved representations, the search space of the evolutionary process is transformed so that the search for new designs is biased towards designs similar to the design examples (Schnier and Gero, 1996).

Attempts to solve design problems using genetic algorithms were focused more on the issue of automated space layout planning by Jagielski and Gero in 1997. They presented the office layout planning problem that was suitable for genetic programming along with some implementation details (Jagielski and Gero, 1997). This office layout planning then improved by Gero and Kazakov by studying the gene evolution which takes place when an algorithm of this type is running. They demonstrated that in many cases, the gene evolution effectively leads to the partial decomposition of the layout problem by grouping some activities together and optimally placing these groups during the first stage of the computation. They showed that the algorithm finds the solution faster than standard evolutionary methods and that evolved genes represent design features that can be re-used later in a range of similar problems (Gero and Kazakov, 1998).

This process continued by Rosenman and Gero in 1999. They presented two examples of work for evolving designs by generating useful complex gene structures. Their work is presented and discussed in the following chapters (Rosenman and Gero, 1999).

Other contributions toward the space layout planning using the genetic algorithm include the works of Gero (1996), Gero and Kazakov (1996a; 1996b) Damski et al. (1997), Jo and Gero (1998).

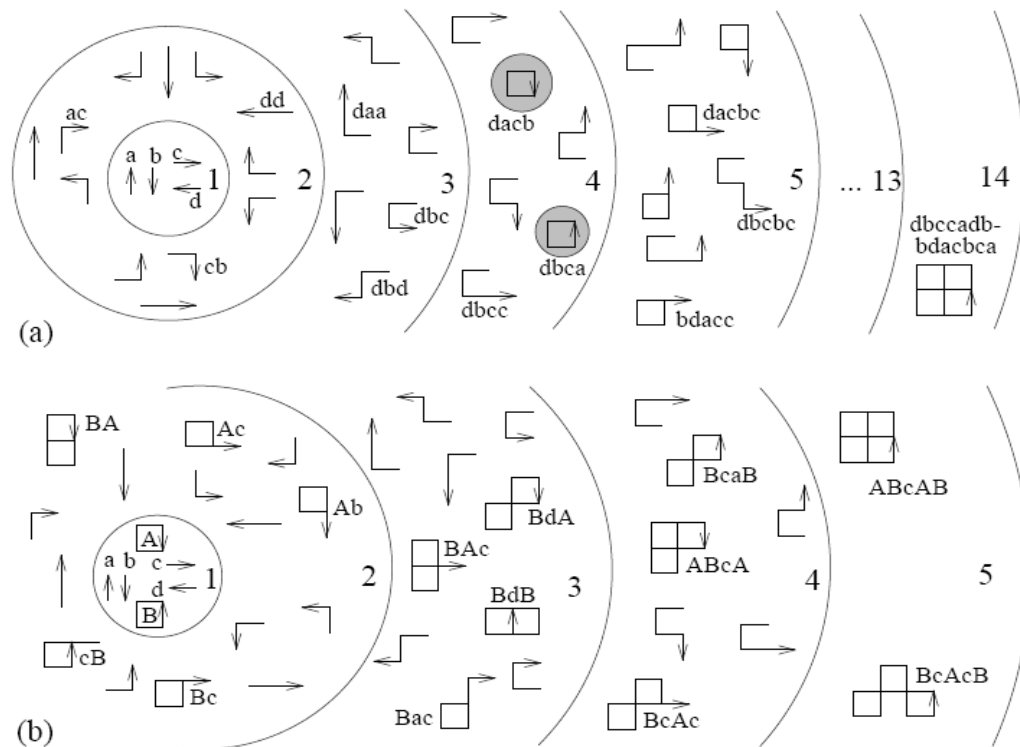
### **3.4.2. Structure of the program (Rosenman and Gero, 1999)**

This paper presents two examples of work for evolving designs by generating useful complex gene structures. The first example uses a genetic engineering approach whereas the other uses a growth model of form.

#### ***Evolving Complex Design Genes Using a Genetic Engineering Approach***

This approach is based on making complex genes from basic genes and using those genes in next generations as evolved genes. The starting point for evolving complex genes is a population of randomly created individuals using a coding with basic genes. This coding has to allow for genotypes of variable length. The individuals are evolved through a number of generations using a given fitness function. At the same time, an additional operation identifies particularly successful combinations of genes. For every gene combination, a new evolved gene is created that represents this combination and is then introduced into the population. At first, the evolved genes are composed of basic genes, but in later cycles most evolved genes are composed of combinations of lower-level, evolved or basic genes. This growing hierarchy of representations gives rise to a more complex and abstract coding, which contains domain specific knowledge that is learnt by the system.

An evolved complex gene subsequently becomes an atom in a new coding and hence the lower-level genes that are represented by it are protected from disturbance from genetic operations such as crossover and mutation. Figure 11 illustrates this process by placing the basic genes in the center of the circles. Consequently, the further away a solution is from the centre, the more complex it is, and the larger the space that has to be searched to find it.



**Figure 11-Example of an evolving representation: (a) original representation and (b) representation with evolved genes (Gero and Schnier, 1995).**

The evolved representation can be now used to solve other, similar, problems. The initial population is generated using both the original basic genes and the evolved genes. The presence of the original basic genes ensures that the whole original search

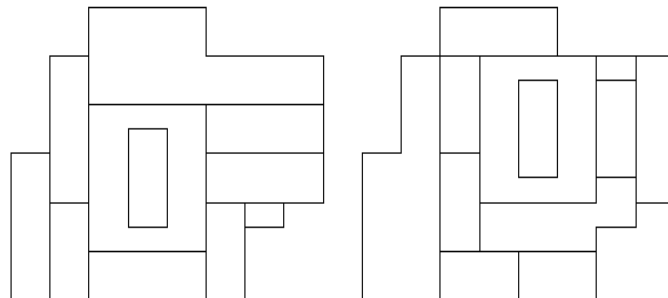
space can still be searched, while the evolved genes restructure the search space in favor of structures that were established as useful when the complex genes were evolved.

The introduction of evolved genes obviously changes the probability that a part of a genotype maps onto a useful feature. While the number of different genes that can be used in the genotypes expands, with the use of evolved genes, the length of the genotype shrinks.

After newly evolved genes were identified and the population became healthier by using micro mutation operations on those genotypes which lack the evolved genes, the standard GA is run for a predetermined number of generations. The cycle is then repeated.

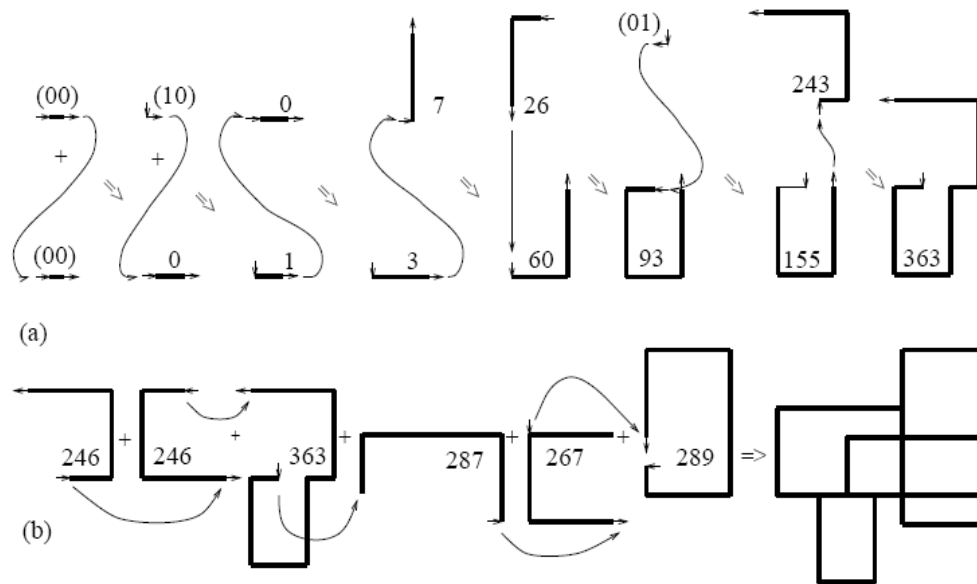
This prototype method is then used in the design of architectural floor plans. As the basic coding, four different basic genes are used that either draw a line in the current direction, move the pen ahead, or change the current direction.

The two floor plans shown in figure are used together as the designs to be represented. The fitness function compares individuals with this drawing, and rewards individuals depending on how much of the drawing they fit.



**Figure 12-Room plans used as design cases (Gero and Schmier, 1995)**

To create evolved genes, successful combinations of genes in the population have to be identified. These evolved genes are then used in the creation of more evolved genes.



**Figure 13- Evolved representation (Gero and Schnier, 1995)**

After a representation based on the examples in Figure 12 has been developed, it is used for new different fitness requirements. A standard evolutionary algorithm is used, where the fitness requirements are coded into the fitness function. The representation is not evolved further. Instead, the set of evolved genes that learned from the examples is used, along with the original basic genes. In an example set out by Rosenman and Gero, the new requirement was to create a floor plan with minimal overall wall length, while at the same time fulfilling the following additional requirements:

1. No walls with open ends, that is, no walls that do not build a closed room;
2. 6 rooms;
3. Room sizes 300, 300, 200, 200, 100 and 100 units.

The additional requirements were given higher priority than the minimization of the wall length. Figure 15 shows the result of one run, after 150,000 crossovers were performed.

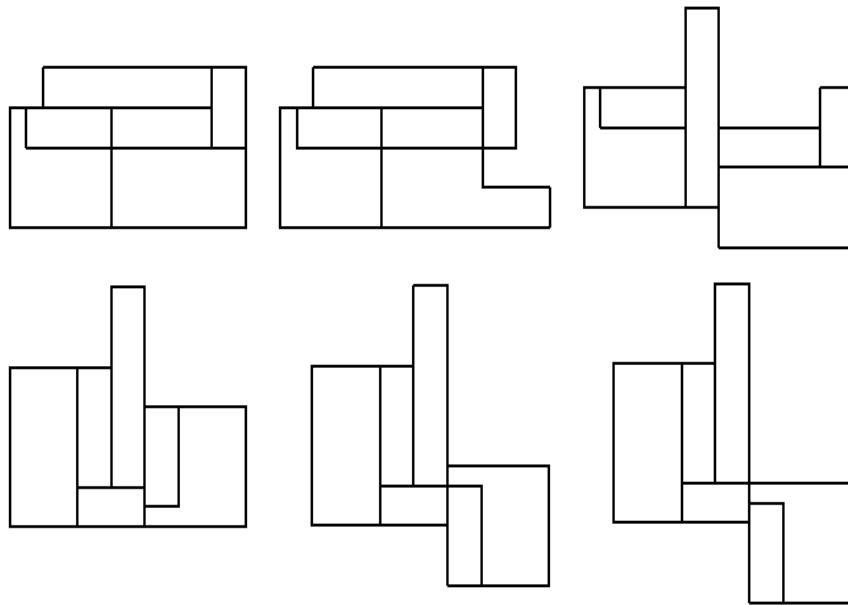


Figure 14- New floor plans using design knowledge from the design cases (Gero and Schnier, 1995)

### *Evolving Complex Design Genes Using a Hierarchical Growth Approach*

The hierarchical growth approach that is used by Rosenman and Gero is a bottom-up multi-level approach where, at each level, a component is generated from a combination of components from the level immediately below. At each level, an initial population is generated and then evolved over a number of generations until a satisfactory population of objects is obtained. Members of that population are then selected as suitable components for generation of the initial population at the next level. The process is repeated for all levels.

The advantages of such a hierarchical approach are that only those factors relevant to that component are considered and factors relevant to the relationships between components are treated at their assembly level. Instead of a single long genotype consisting of a large number of basic genes, the genotype is composed of a set of chromosomes relevant to their particular level. In addition to reducing the combinatorial problem substantially, parallelism is supported since all the different chromosomes (components) at a particular level can be generated in parallel. If the set of possible alternatives of component types is sufficiently large and varied, then many different combinations of members of different such sets are possible. At the next level, there is a good chance of satisfying the criteria and constraints. Only when no possible combination satisfies criteria, there is a need for generation of new alternatives at the lower level.

The aim of the design process in this evolutionary approach is the attainment of a set of instructions (a genotype), that when executed, yields a design description of a product (a phenotype), whose interpreted behaviors satisfy a set of required behaviors (the fitness function). In this approach, a grammar rule is a gene, the plan (sequence of rules) is the genotype and the design solution is the phenotype. The approach taken here is based on the premise that the grammar rules are fundamental operators, which cannot be decomposed or recomposed, that the particular grammar contains all required rules and that the aim of the design process is to find satisfactory sequences of such rules.

In Rosenman and Gero's work, these concepts are exemplified through the generation of 2D plans for single story houses. Previous work had demonstrated that a single-level approach was not able to converge towards satisfactory solutions mainly due to the interactions of the various factors of the fitness function required for the various elements (Jo, 1993; Jo and Gero, 1995).

In this work, a house is considered to be composed of a number of zones, such as living zone, bed zone, utility zone, etc. Each zone is composed of a number of rooms, such as living room, dining room, bedroom, etc. Each room is composed of a number of space units. Generally, in a design such as a house, the space unit is constant. In this formulation, the generation of spaces, basically comes down to locating spatial component units for that level. At the room level, the component unit is a fundamental unit of space. At the zone level, the component unit is a room and at the house level the component unit is a zone.

The design grammar used here is based on the method for constructing polygonal shapes represented as closed loops of edge vectors (Rosenman, 1995). This rule ensures that new cells are always added at the perimeter of the new resultant shape.

To use the spatial hierarchy methods in the evolution of house designs, at each level, different fitness functions apply according to the requirements for that level. While the requirements for designs of houses involve many factors, many of which cannot be quantified or adequately formulated in a fitness function, some simple factors have been used initially to test the feasibility of the approach. For this example, the fitness function for rooms consists of minimizing the perimeter-to-area ratio and the number of angles. This requirement tends to produce compact forms, useful as rooms. For zones, the fitness function consists of minimizing a sum of adjacency requirements between rooms reflecting functional requirements. At the house level, the fitness function consists of minimizing a sum of adjacency requirements between rooms in one zone and rooms in other zones. This has the tendency to select those arrangements of zones where adjacency interrelations are required between rooms of different zones.

Although these criteria have been described in terms of optimizing functions, the aim is not to produce global optimum solutions, but rather to direct the evolutionary process to produce populations of good solutions either as components for higher levels

or at the final level itself. By selecting other non-optimal but good solutions, according to the given criteria, good unexpected results may be achieved for the overall design.

Results are shown in the following figures.

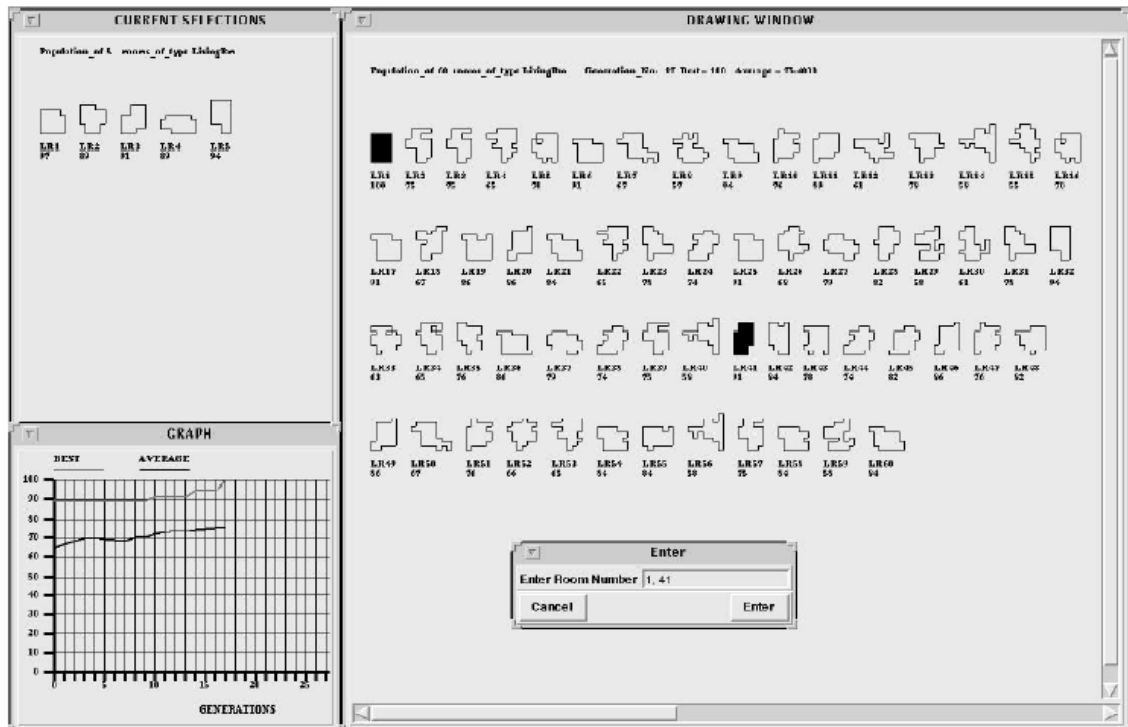


Figure 15- Results of Living room Generation after 17<sup>th</sup> generation (Rosenman and Gero, 1999)

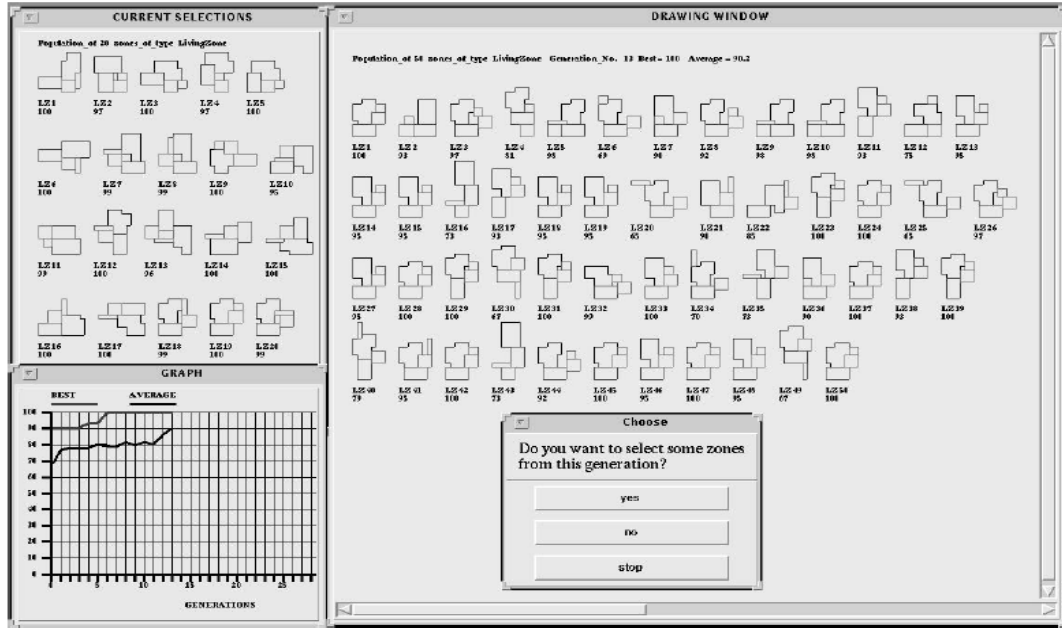


Figure 16- Results of Living Zone Generation (Rosenman and Gero, 1999)

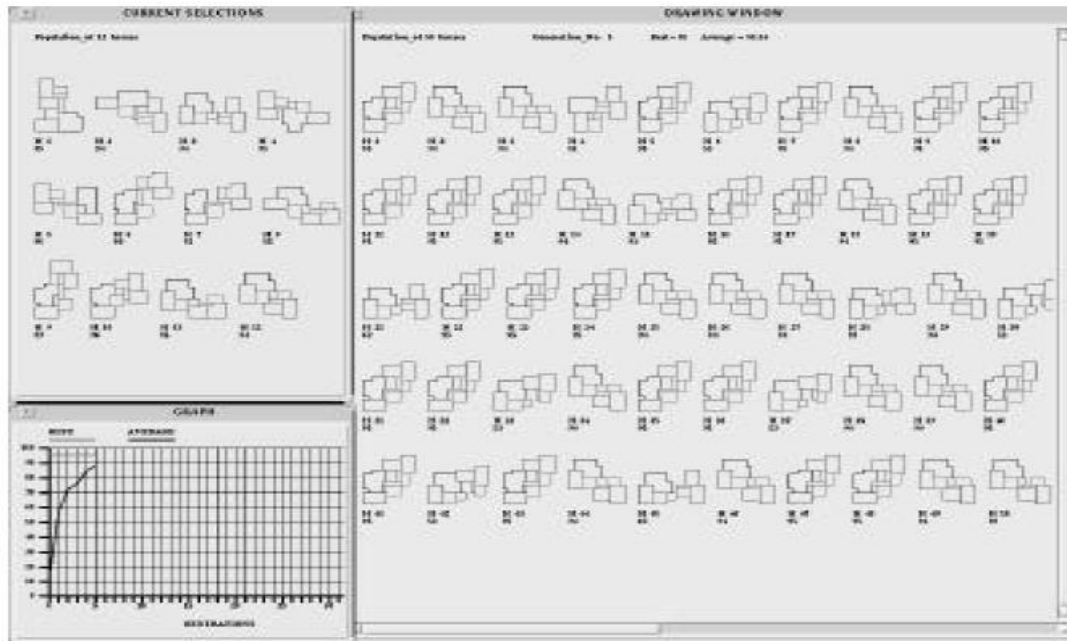


Figure 17- Results of House Generation (Rosenman and Gero, 1999)

### 3.4.3. Discussion

The hierarchical growth model represents the strong method of genetic algorithm approaches to space layout planning. The advent of the hierarchical growth approach helped to organize the generating process and made the design process more meaningful and manageable for the architects. However, there are still some arguments about the use of this system, and the genetic algorithm technique in general, in space layout planning tasks.

Firstly, when a number of fitness functions are defined which address the same thing, but differently and without a weighted value, the system tends to use only the function that is more likely to happen during the mating and mutation process. This would make the design more uniform although that may not be desirable for the user. An example of this issue is depicted well in Figure 14, which shows the phenotypes of the generating process of space planning based on the fitness function that compares individuals with drawings depicted in Figure 12. None of the results in Figure 14 has the frame shape at the middle of drawings simply because it is less likely to happen than the rectangular shapes during the crossover and mutation process.

Secondly, design is a process that redefines itself. That means beside the main goals of each project that are specified from the beginning, based on different decisions that are made during the design process, and different situations that are created, different local goals would be defined and perused. With this definition, we can argue that having a set of predefined goals (or fitness functions) before starting the design (or running the program) may not be the best way of doing the task. The result of the program would have been much adapted to the design if the fitness function could have been changed during the generation process, although this seems like a very different task.

The other problem with this system, which is kind of generalized between all the programs that are based on genetic algorithm approach, is that the program is not capable

of categorizing the solutions and recognizing the novel ones by itself. The software needs the user to go through all solutions and search for a novel design after each level. For big projects like hospitals which consist of many zones and many adjacency rules, verifying all the alternatives that the system generates would be a demanding task, redundant and tedious by itself.

Finally, in contrast with artificial neural network systems that learn from their mistakes and make experiments over time, this approach lacks a learning feature. However, combining the system with a neural network system that lacks a sufficient generating system seems to be a responsible approach to “novel space layout planning”.

### ***3.5. Comparison and Results***

Table 1 shows the strengths and weaknesses of the solution approaches to computerized space layout planning that were studied in this research.

**Table 1- Comparison chart of strengths and weaknesses of the studied approaches**

	Handling different types of constraints	Runtime/ Complexity	Creating Novel Solutions	Evaluating Solutions	User friendliness (with architects)
Additive Space Allocation	poor	acceptable	poor	poor	poor
Permutation al Space Allocation	acceptable	poor	poor	excellent	poor
Analogical Methods	excellent	acceptable	poor	poor	excellent
Genetic Algorithms	excellent	acceptable	excellent	excellent	acceptable

The overall review of the papers and the follow-up comparison chart helps create an overview about the future work in this field:

1. There is a demand for more comprehensive software to support the design process.
2. Currently most architects do not use these computational approaches in the design process.

In fact, despite the increasing need of architects to have a computational assistant in the design task, they do not show much interest in any of the available

programs. To have a better understanding about this issue, we should first get a broader viewpoint about architects jobs and the type of available space planning programs.

There are different types of programs that help architects in the design process. Each has its own domain and ability of supporting the designer. The simplest form of these programs is tools–implements that enhance the individual’s power by providing some assistance in performing the tasks that do not need a human intelligence (Kalay, 2006). Architects and other professionals have shown great willingness to use these programs. Examples of these kinds of programs include CAD systems. These systems mostly help the designers to avoid redundancy and do their job faster. Users are the main designers and the navigators of the system, and they absolutely trust the system.

A more advanced form of support in computer-aided design technologies is assistants–semiautonomous tools which can take an instruction and carry it out, even when the task involves multiple steps and some judgment (Kalay, 2006).

The space layout planning systems are categorized in this form of support in computer-aided design technologies. There are several reasons that currently architects do not show much enthusiasm for these programs:

First, most of these programs are still at a research prototype stage. They still have many bugs and need further improvements. One of the reasons that these programs do not get to the marketing stage is their excessive runtime. On one hand, the runtime of these programs increases exponentially with increasing problem complexity so that the program cannot handle the more complex problems. On the other hand, the best application of such programs are complex projects. In fact, architects usually have enough experience and knowledge to solve the design objectives in small projects that they do not need to use additional helping software. Moreover, none of the current available software is capable of producing a truly novel design as architects do. Neither

they are able to consider many design criteria such as aesthetic values, which are of the most important goals of the design.

Another reason that architects do not show much willingness to use these kinds of programs is that their job is design and therefore most of them need software that helps them through the difficulties of the design task, not to design on their behalf. They want to be the controller of their assistant software not to be limited by the commands and decisions that the software takes. In fact, these programs still are not considered reliable by the designers.

Designers believe that they can design better than the software. Even if the architects are convinced about the performance of these programs, they are not the best user of such software, since it does, to some degree, threaten their job security. Hence, even if these programs were truly working great, meaning that they were truly are capable of design, there would be no need for a designer to use them. A junior architect that is familiar with architectural matters would be the best user for these programs, because these programs enable him to do what he was not able to do before. Or a computer assistant could be employed to take over the role of a junior architect.

Certainly verifying the recent statements requires further case studies and statistical researches that would compose the future works. Finding the right user for each program and specializing the program for its future users would be another job that should be done in future. If we are designing software for architects' assistant, we should respect the desire of architects to design freely and with fortitude.

In short, design is a very complicated process that human beings are able to accomplish. Giving this intricate task to the computer needs a more human-like computer system that can be adapted to the sophistication and elegance of the human mind that is used in the design process.

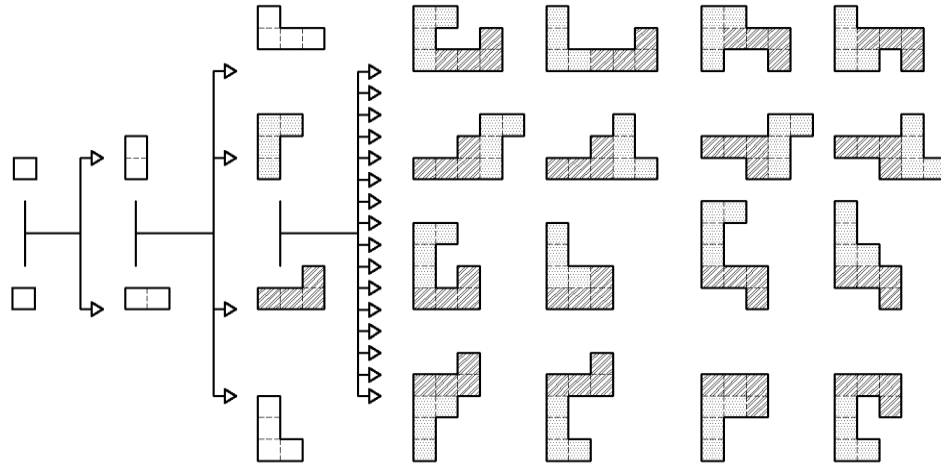
## **4. A genetic engineering approach to Space Layout Planning**

This section describes a new approach to generating floor plan layouts with the help of Genetic Algorithm and Engineering. This approach is mainly derived from the hierarchical growth approach that was developed by Rosenman and Gero (Rosenman and Gero, 1999), discussed in Chapter 3, but benefits from more genetic engineering techniques and strategies.

The program is executed in two main phases, or levels:

1. Room level: in room level for all of the rooms that a user defines, several alternatives that satisfy the defined criteria are generated.
2. Building level: in Building level the alternative rooms are put together, and several alternatives for design of the building layout is presented.

At the room level, the initial populations (the initial genotypes) are defined using one-meter squares equivalent to the span of a normal door. The crossover function is defined as putting one population on top or to the right side of the other. In both cases this function has the random choice of making each of the parents upside down or mirrored. Figure 18 shows the growing model of form in this level, as a result of the crossover function.



**Figure 18- Results of the crossover function at the room level**

After performing the crossover function over the populations in the first generation, the resulted populations are evaluated based on a fitness function at the room level. This fitness function includes four sub-functions, each evaluating different characteristics of the generated rooms:

1. Area fitness: This function ensures that the area of the selected population (room) is close enough to the user defined area.
2. Perimeter fitness: This function evaluates the population based on number of angles that each room has. As a result, this function gives a better (lower) fitness value to the compact forms (lower perimeter/area ratio).
3. Concavity fitness: This function measures the degree of concavity for each room, and gives a high (penalizing) fitness value to the rooms that have a high degree of concavity such that the center of gravity for the room is outside of it.
4. Proportion fitness: This function compares the length/width ratio of each population with that defined by the user and gives a better (lower) fitness

value to those rooms with the closer proportion to the target proportion value.

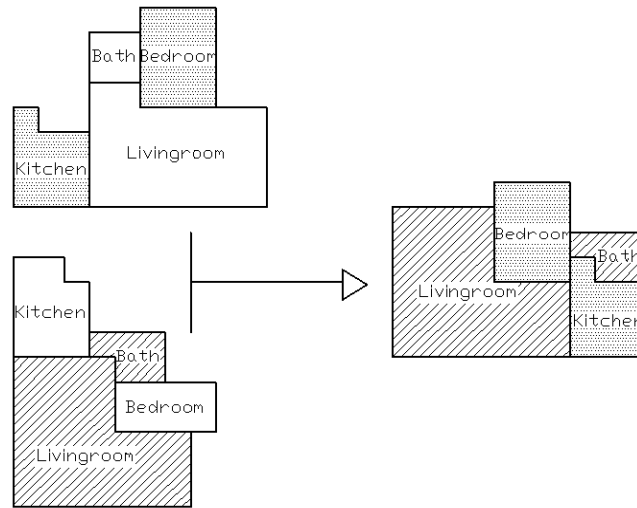
The process of mating populations (genotypes) in the last generation and evaluating the fitness of new populations continues until the satisfactory fitness for the first room is reached (based on the criteria and values that user has defined for that particular room). All the rooms in the final generation that have reached the satisfactory fitness are selected as initial genes for the building level. This process is repeated several times, so that for each room, there are a constant number of alternatives available as an input for the next level.

At the building level, the goal is to create a layout in which the adjacency requirements between the rooms are mostly satisfied. Therefore, one of the main fitness functions at this level is “Adjacency fitness” which evaluates a layout based on the number of adjacency requirements that are not satisfied and the importance of each one based on the user criteria. Other fitness functions in this level are “Area fitness”, “Perimeter fitness” and “Proportion fitness” which could affect the overall fitness value based on the user request.

To create an initial population for the building level, for each genotype (building layout) one room is selected from each room set and all of the selected rooms are put together by a function called “putToGether”. This function adds each room to the collection of previous rooms by putting one of its random corners on one of the exterior corners of the collection without an overlap and in a way that the new room is connected to the collection by at least one unit of a side (which is 1 meter, equivalent to the width of an ordinary door).

After creating the initial population, the crossover function at the building level is executed. During the crossover function, a random number of rooms are selected from the first parent and the rest are selected from the second parent. These rooms are again

combined by means of the “putToGether” function. The process is illustrated in Figure 19.

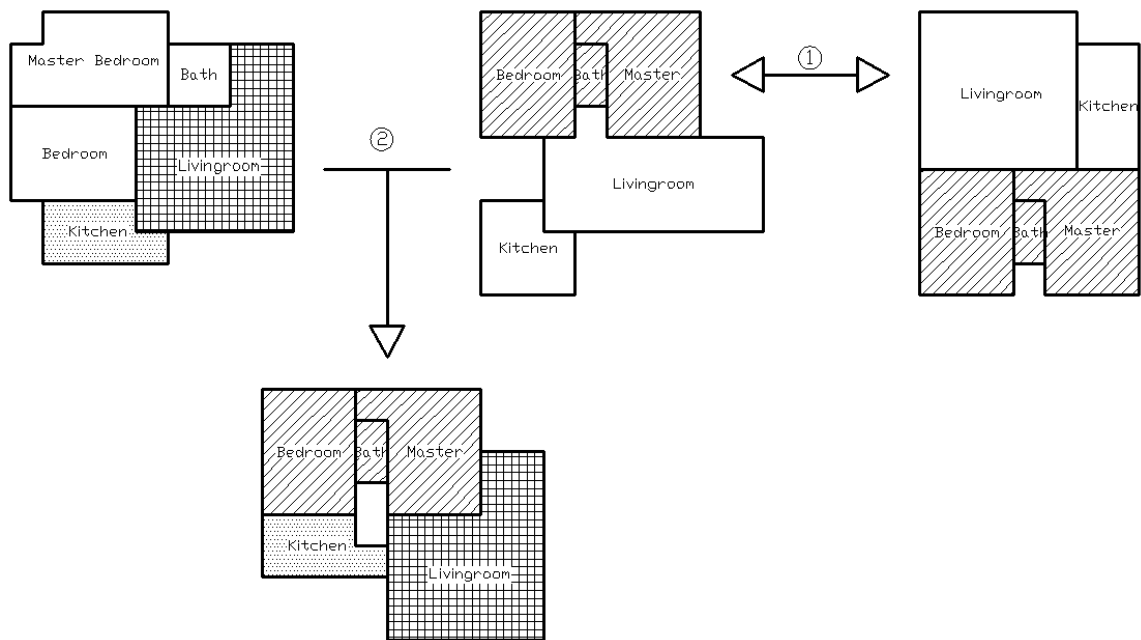


**Figure 19-Crossover function at the building level**

Execution of this function over multiple generations slightly improves the fitness value of the building layouts, but this progress in fact occurs only on the part of the proportion and area fitness. The adjacency and perimeter fitness does not improve distinctly since all of the room connections are broken in this process and new connections are formed.

In order to prevent the program from breaking the adjacencies between good combinations of rooms, the idea of making evolved genes (explained in section 3.4.2) is used. After each generation, the best populations of building layouts are selected and one by one compared together. Combinations of rooms that are literally repeated in a number of best populations are selected and marked as evolved genes so that the perimeter and adjacency fitness value of populations improves in subsequent generations. This method also significantly helps toward the improvement of the area fitness.

Having created evolved genes out of successful combinations of rooms, instead of selecting random rooms from each parent during the crossover function, an evolved gene (if existed) is selected from one of the parents and the rest of rooms that do not exist in the evolved gene are selected from the other parent (who may or may not have an evolved gene). This method increases the use of evolved genes in the new populations and therefore the overall effectiveness of using the evolved genes increases. The process of making evolved genes and their role in the crossover function is illustrated in Figure 20.



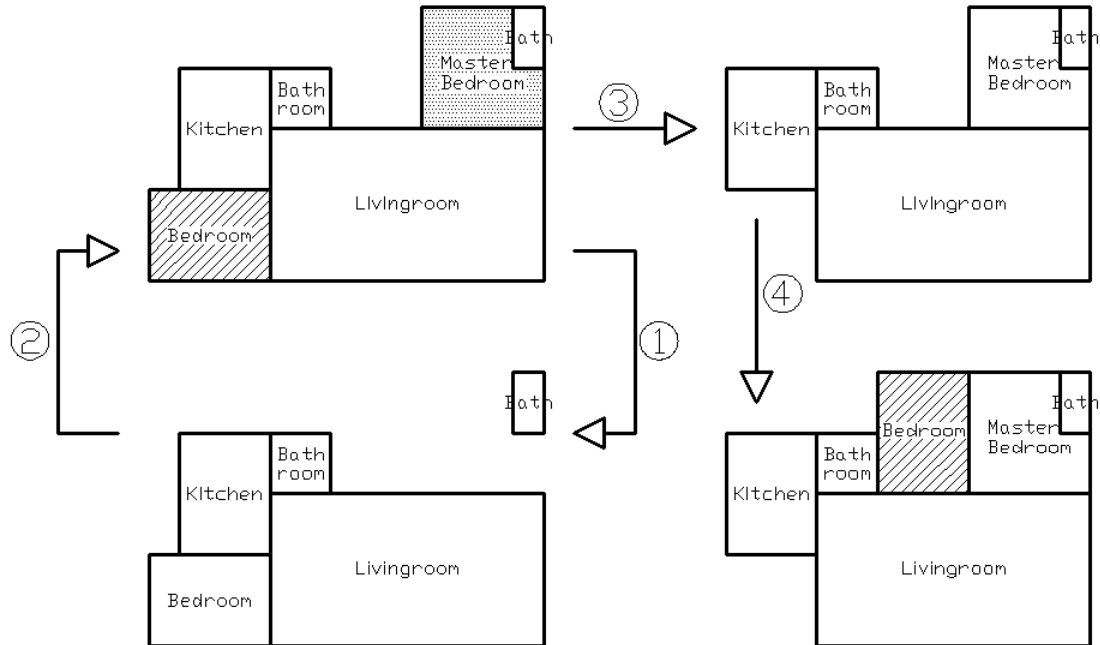
**Figure 20- Part 1 shows the comparison between the two selected populations that results in creating an evolved gene out of the common components. Part 2 illustrates the behavior of crossover function when an evolved gene is involved.**

After performing the crossover function, all of the evolved genes are divided to their original room components and are treated like other rooms in next populations. This division does not reduce the impact of evolved genes since from each evolved gene, at

least 2 would be available in a next population and if the evolved gene truly is a good combination, the two building layouts that contain those evolved genes will show up in best populations and will be selected and evolved again. It is also possible that in the next generation even more rooms will be added to the combination, creating a larger evolved gene. This method helps to prevent mistakenly selected evolved genes from spreading in the populations and decreasing the overall fitness value.

After performing the crossover function, populations are all passed through the Mutation Function. At this point, from each population, one of the exterior rooms is selected and removed from the combination. The continuity of the rooms in the combination is checked afterward and if the combination was discrete, another room is selected and checked until one room is found that the combination remains continuous after its removal.

The selected room is removed from the layout, turned and put back into the layout via “putTogether” function. The mutation function in fact searches around the existing combinations, and tries to make them better without breaking their whole structures. The process of changing a layout via mutation function is depicted in Figure 21.



**Figure 21- An Example of a Mutation process: Part 1 shows the removal of the Master bedroom from the building layout which results to separation of the Bath from the rest of the layout. Part 2 returns the combination to its original layout. Part 3 shows the removal of the Bedroom which keeps the building layout continuous. In Part 4 the removed Bedroom is turned and put back to the combination.**

After performing the mutation function, populations are evaluated with the fitness function and the best results create the next generation. The process of mating, mutation and evolving continues until the satisfactory result is gained. At that time, the most successful populations will be selected and displayed.

#### ***4.1. Implementation and results***

A computer program written in Matlab version 7.0.4 was implemented. The built-in Genetic Algorithm toolkit was adopted, and then extended with genetic engineering methods.

#### **4.1.1. Design objectives**

The main objective in this implementation was designing a program that generates architectural plans according to specific design criteria, allowing a designer/researcher to experiment and manipulate different criteria within the program. The program was intended to produce viable results, in reasonable amount of time, in a way that is competitive in terms of solution quality and variety.

#### **4.1.2. Design layout**

This program is designed with three graphical User Interfaces (GUI) along with built-in genetic algorithm GUIs that are being executed in a sequence. The first GUI is designed to take the general user preferences about the building as a whole. Four questions are asked at this level:

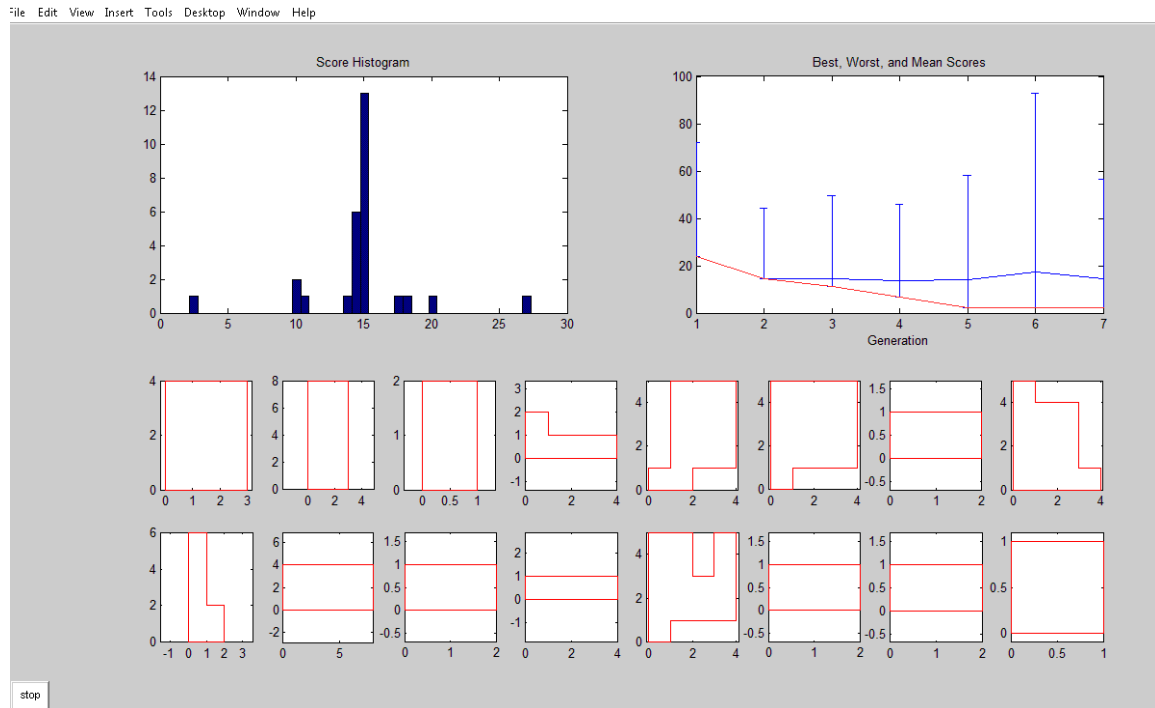
1. How many rooms does your building have?
2. Do you want to minimize the area of the building as much as possible?
3. Do you want to minimize the perimeter of the building?
4. What is the best length/width ratio for the building?

Pressing the submit button brings up the next dialog which is designed to take the information about each room. The user then has to give the program information about the first room of the building. The questions that are asked at this level are:

1. What is the name of this room?
2. What is the best area for the room?
3. Do you prefer compact forms?
4. What is the best proportion for the room?

Submitting the information at this point brings up the genetic algorithm GUI that shows the populations of rooms in each generation along with two graphs (Figure 22). The first one is a histogram that shows fitness scores in all generations and the second

one tracks the best, worst, and mean fitness value in each generation (The greater the slope of the best scores curve, the better the performance of the program).



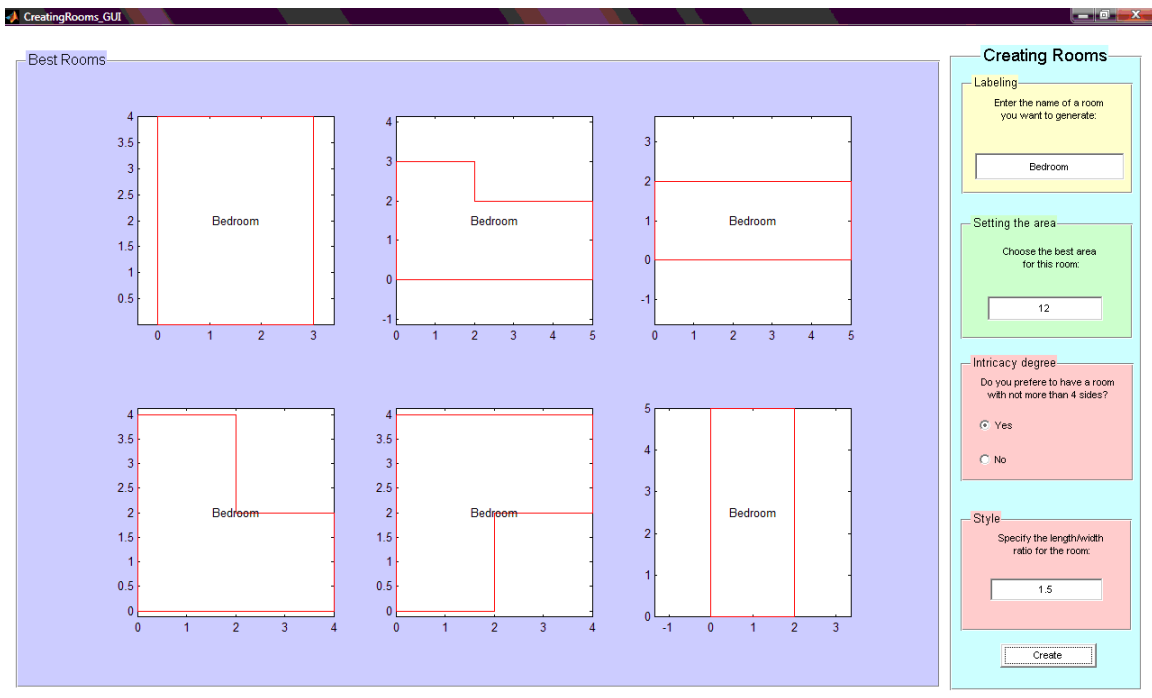
**Figure 22- An example of a GUI with population of rooms in 7<sup>th</sup> generation**

The program stops generating new populations whenever one of the six following conditions is met:

1. The fitness value of a best population is less than a threshold value
2. The maximum number of generations is reached
3. The time limit is exceeded
4. No improvement shows up in the objective function for a defined number of consecutive generations (StallGenLimit).
5. No improvement shows up in the objective function for a defined amount of time (StallTimeLimit).

6. The user requests terminating the process by pressing the stop button.

At that time, all of the rooms in the last generation are checked and the ones that have an acceptable fitness value are printed to the associated GUI. In order to overcome the general merging trait of genetic algorithms, the program is executed multiple times with the same inputs until the predefined number of various rooms is generated (Figure 23).



**Figure 23- An example of a GUI in the room level that shows the results of creating a requested room.**

The same interface is then used to get information about the next room and the process is repeated until all of the rooms the user requested have been created.

At the end of the room level, a third designed GUI appears displaying all of the room names that the user defined previously and asking about the adjacency priorities

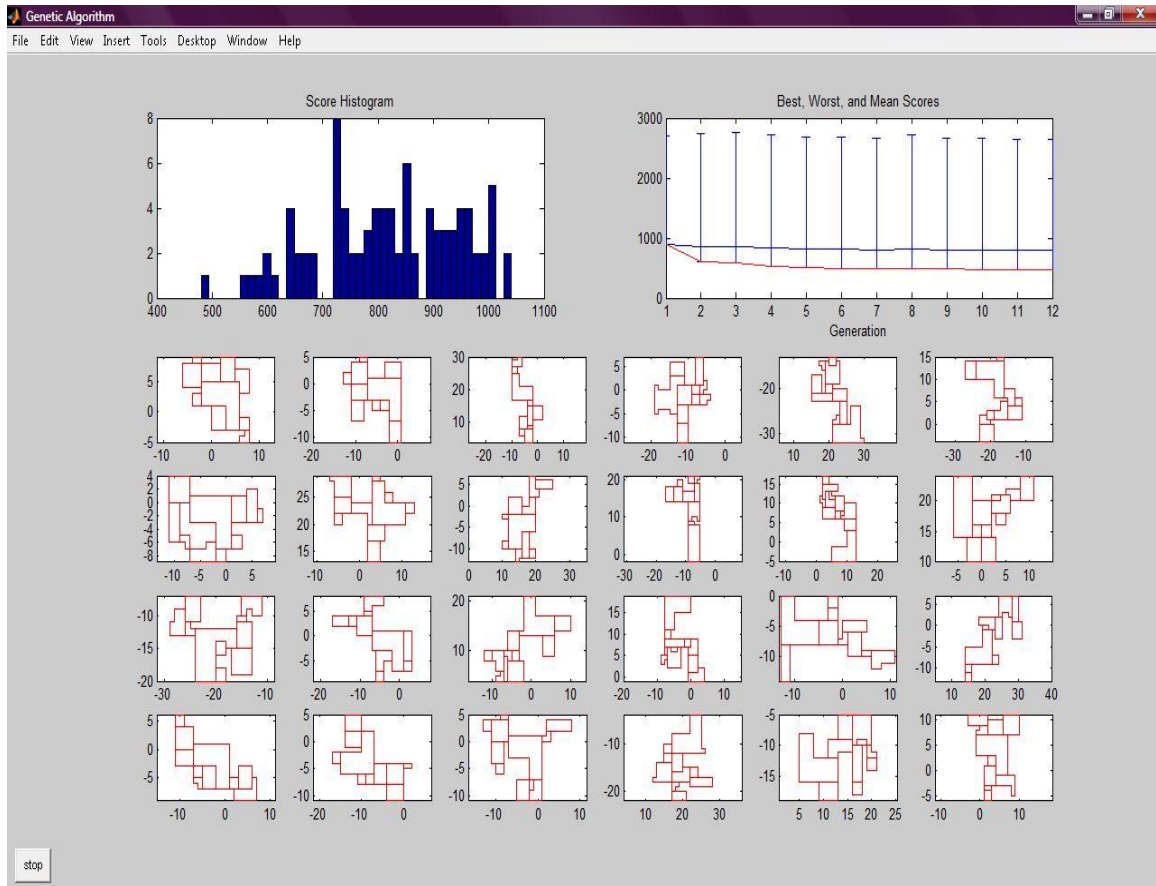
between the rooms, and also between each room and the outside. This GUI is shown in Figure 24.

Specify the adjacency priorities between the rooms by assigning values between 0 and 10 to them:

	outside	Bedroom	Livingroom	Kitchen	Diningroom	Bathroom	MasterBedroom	Bath	Laundry	Office
Bedroom	0									
Livingroom	0	0								
Kitchen	0	0	0							
Diningroom	0	0	0	0						
Bathroom	0	0	0	0	0					
MasterBedroom	0	0	0	0	0	0				
Bath	0	0	0	0	0	0	0			
Laundry	0	0	0	0	0	0	0	0		
Office	0	0	0	0	0	0	0	0	0	
Entrance	0	0	0	0	0	0	0	0	0	0

**Figure 24- Adjacency requirements GUI**

Having accepted the adjacency priorities from the user, the building-level genetic algorithm GUI appears. This GUI is similar to the room-level GUI . It has the two previously explained graphs and a pool of generated populations (Figure 25).



**Figure 25-** An example of a GUI with populations of building layouts in 12<sup>th</sup> generation

At the end of the building-level calculation, the most successful populations of building layouts are printed on their associated GUI.

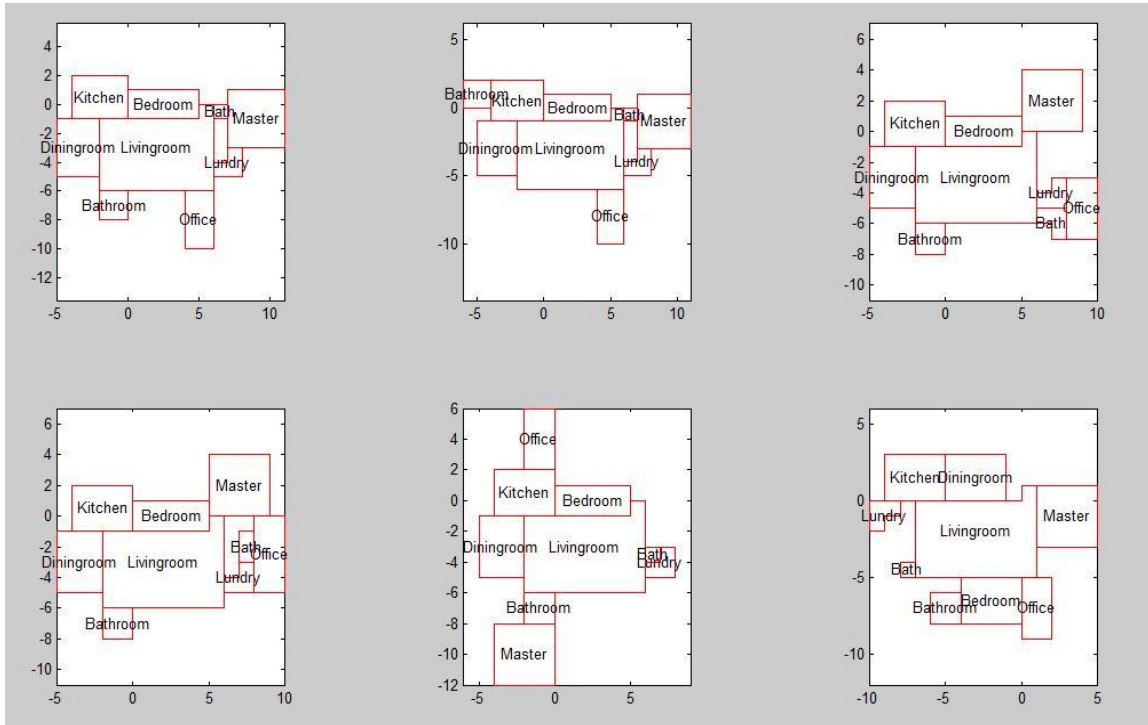


Figure 26- A building-level GUI that shows the results of creating building layouts

## 4.2. Experiments and results

In this section, experimental objectives, setups and results of running the prototype are presented.

### 4.2.1. Experimental objectives

The main objective of this experiment was to optimize the process of generating floor plan layouts in terms of quality of design and program runtime.

Adjusting the value of certain control parameters in this implementation has a direct impact on the performance of the program. These variables include:

- The importance factor of each sub-fitness function in an overall fitness function both in room and building level.
- The threshold fitness value for accepting the population as final results in room level.
- The population size both at the room and building level.
- Number of best populations for comparing and making evolved genes at building level.
- Number of best populations to choose in room level and transfer to the building level.
- The value of terminating variables such as number of generations, time limit and “stall gen” limit in both room and building level.

In order to get the best results, each one of these factors was subject to the experimental setup. Since the program runs much faster at the room level than it does at the building level, and the outcome is rather predictable, there is no need to run the program multiple times with different settings, once we have an acceptable result. However, changing the number of best populations to choose and varying the threshold of the fitness value at the room level have a large impact on the outcome of the building level. The greater the variety of rooms and the higher the threshold of fitness value, the better the rooms go together at the building level.

The question is then about the priority of having well formed rooms or having a layout with not-very-well-shaped rooms that fit together well. The answer could be a mixture of both options. For some rooms (like a bedroom) having a compact form may be important and for some others (like a living room) it may not. Thus, the applicable prototype should have more developed features and ask about this priority for each room. However, in this thesis the focus was to get a convenient result from the prototype under limited circumstances as a base for further studies. As a result, the performance of the

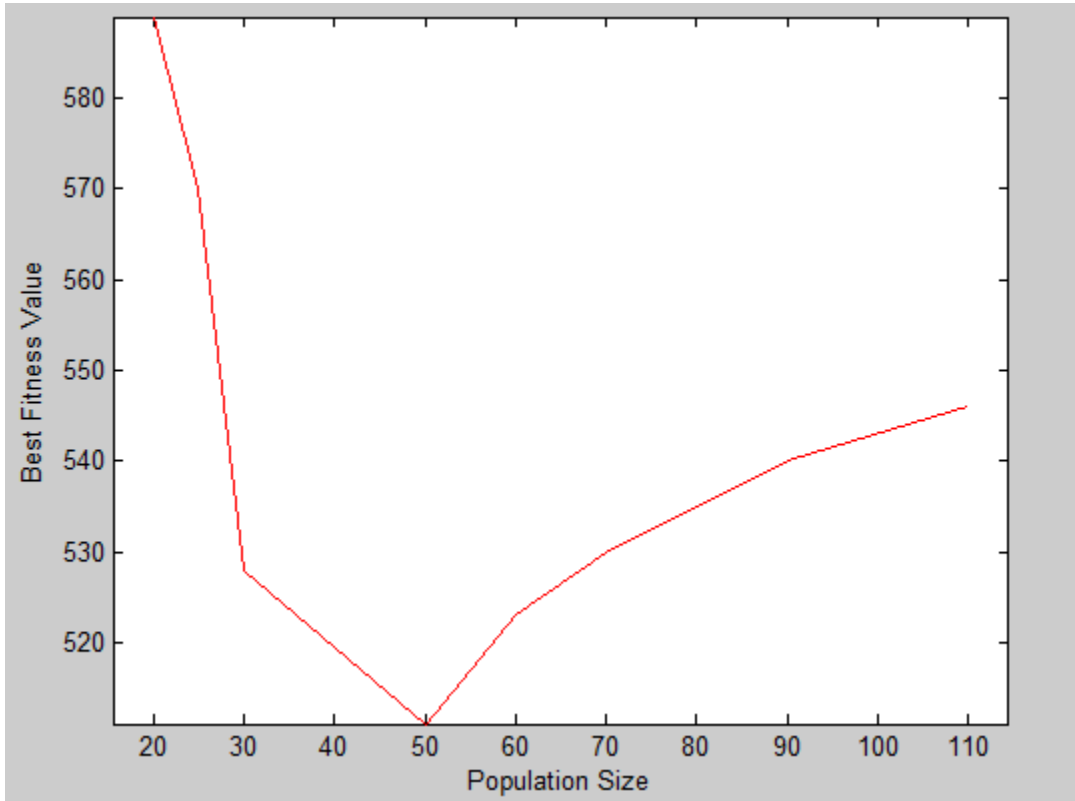
program under the variation of parameters only at the building level is studied and presented.

For each statistic, the program has been run at least four times and the results are presented in a comparison graph. However, some of the resultant patterns in the graphs are pretty erratic as a result of the convergence characteristics of genetic algorithms and their randomness that result in a large span of best fitness values under the same condition. Thus, the precise comparison between the results needs further study. Furthermore, the studied parameters are not independent. Increasing the value of a parameter could be in favor of one function and against another. Therefore, analysis of resulting patterns needs a combinatory analysis on all the functions that have been under the influence of the change.

Despite of all the factors mentioned above, these experiments are still necessary for understanding the overall behavior of the prototype and specifying the direction for achieving the best performance of the prototype.

#### **4.2.2. Experimental setup**

In the first study, the results of creating 10 rooms and defining the adjacency relationship between them are saved and used as an input to the genetic algorithm at the building level in all of the experiments. The number of generations is set to 7, number of best populations for comparing and evolving genes to 20 and all other variables are kept constant throughout the experiment. The graph in Figure 27 shows the result of this experiment in terms of variations of the best fitness values by increasing the population size.

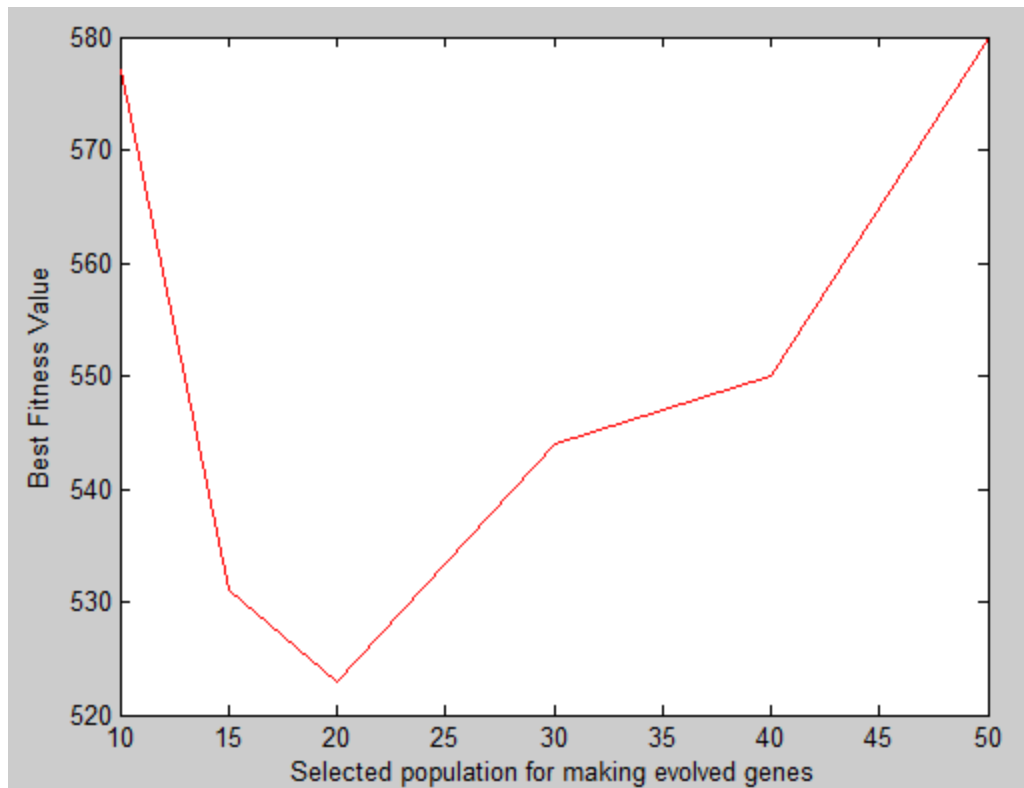


**Figure 27- The graph shows best fitness values for different population sizes**

As explained earlier, the level of accuracy is not clear in any of the graphs. However, the overall behavior of this graph shows that increasing the population size, up to some point, improves the quality of design, and after that, the best fitness value starts to increase (get worse). This variation shows the relationship between the number of selected populations as best populations (for comparison and creating evolved genes) and the population size. When the population size is equal to the number of selected best populations (20 in this experiment), the process of evolving genes, in fact, does not improve the quality of the design, since bad combinations of genes are also evolved and tend to neutralize the virtue of well evolved genes. By increasing the population size, and keeping the number of selected populations constant, evolved genes start to act and affect

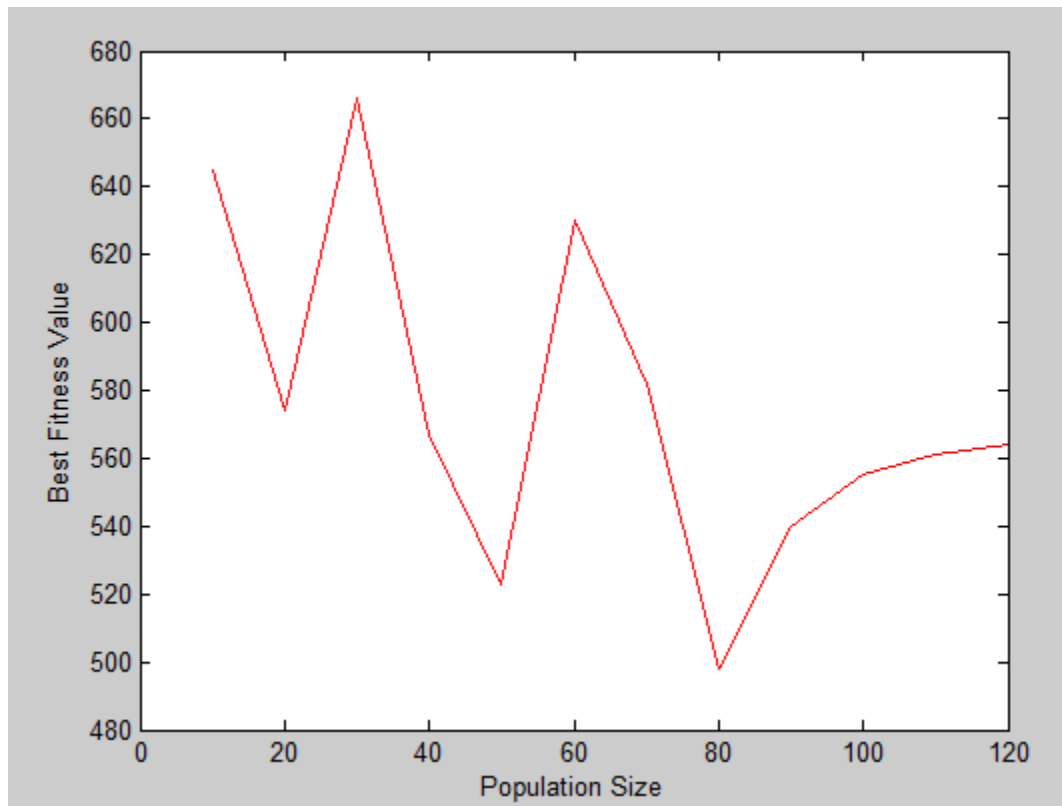
the best fitness values. This improvement continues until the optimum ratio between the selected populations and the population size is reached. After that, increasing the population size results in reducing the effect of evolved genes and as a result the overall fitness value gets worse.

As a result in order to find the best population size for maximizing the performance of the prototype, it is important to establish the ideal percentage value for selecting the best populations. Thus, in the next experiment, the population size is kept constant and the result of variations in the number of selected populations for making evolved genes is studied (Figure 28).



**Figure 28- Variations of the best fitness values by increasing the size of selected populations for comparison and making evolved genes**

The graph shows that by choosing around 20% of populations for comparison and making evolved genes, the best use of creating evolved genes is achieved. Therefore, in the next experiment this percentage is set to 20% and the variations of the best fitness value by changing the population size is studied.

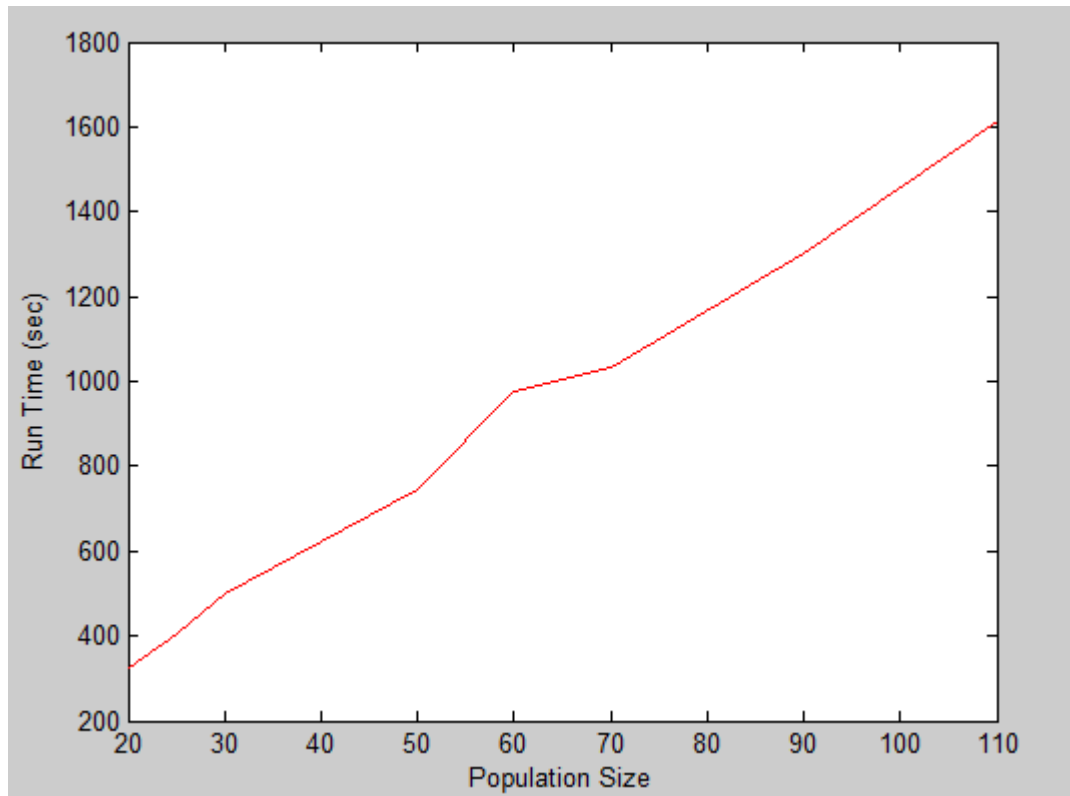


**Figure 29- Variations of best fitness values by increasing the population size**

As discussed earlier, because of the random character of genetic algorithms, even with the same input values, fluctuations of the best fitness value is high. Therefore, studying the variations of best fitness values by changing another variable gets complicated. Hence, the inference from this experiment is based on the overall

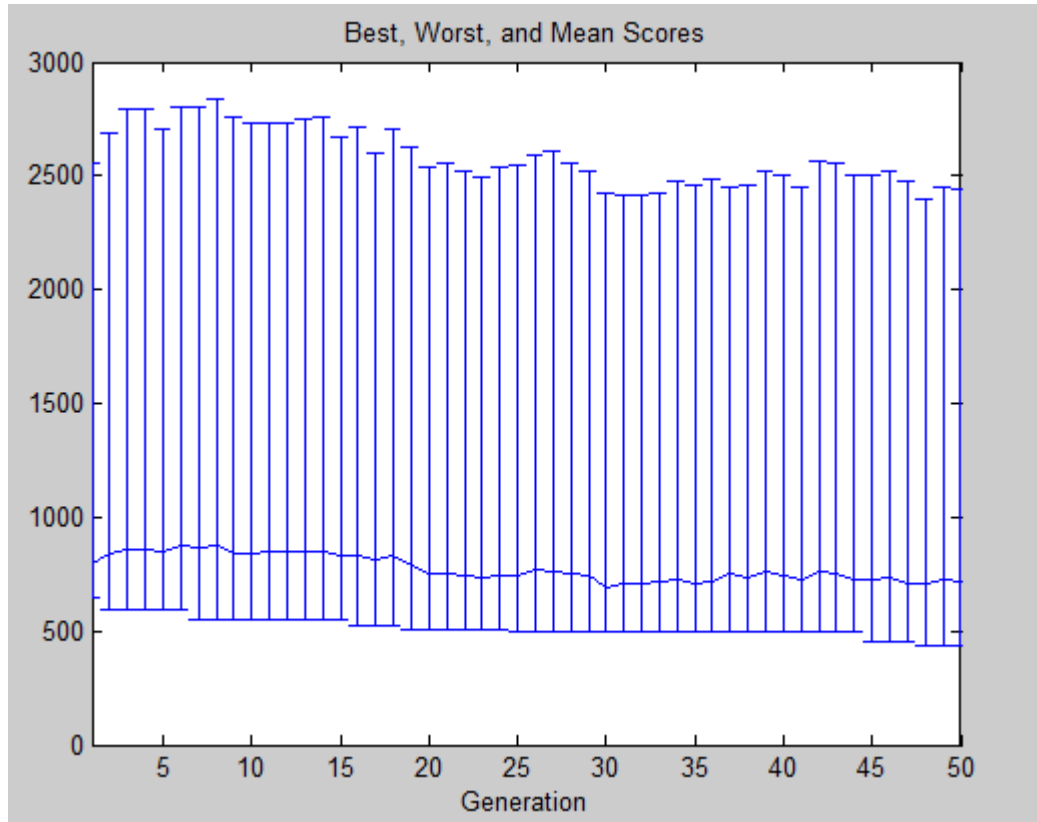
fluctuations of the graph (Figure 29) which shows the positive influence of increasing the population size on the best fitness value.

However, studying the results of the same experiments in terms of variations of time by increasing the population size (Figure 30) shows that the program runtime grows linearly with a quite high slope when increasing the population size. Therefore, improving the performance of the program by increasing the population size is likely to conflict with the desire to minimize the runtime of the program.



**Figure 30-** The graph shows the program runtime versus the population size

Another effective way in optimizing the performance of the program is changing the number of generations along with other variations that are used for stopping the generations.

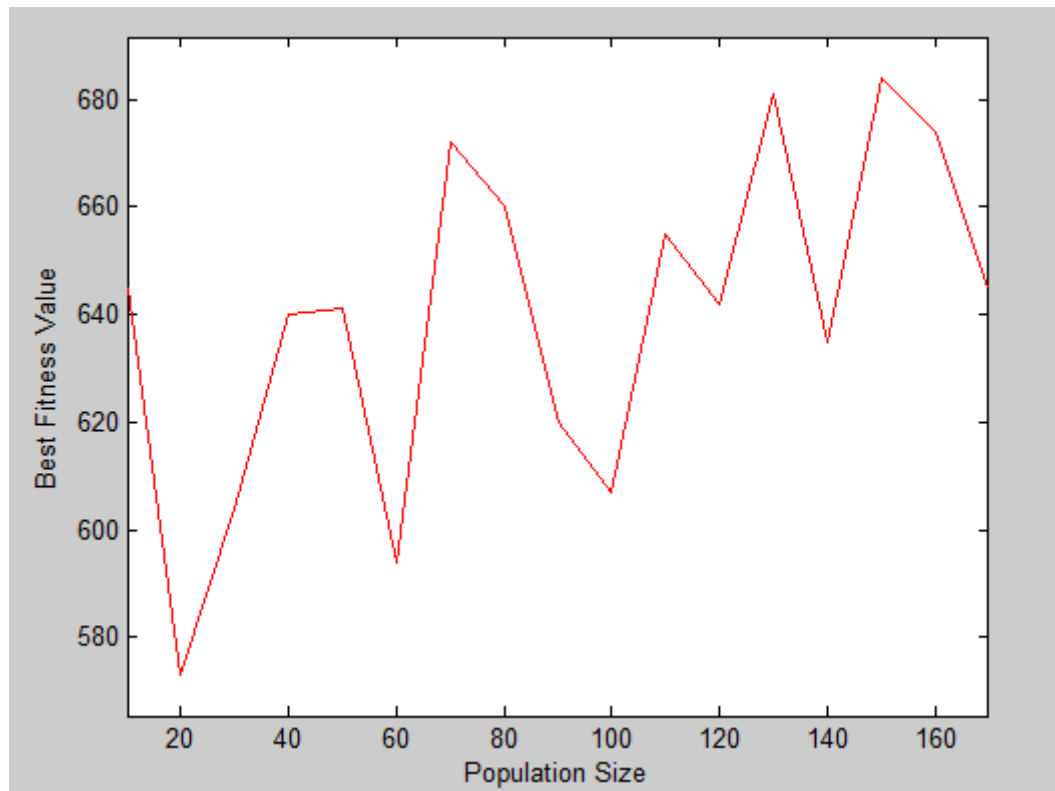


**Figure 31- Improving the best fitness value by increasing the number of generations**

As shown in Figure 31, the process of improving the fitness value continues throughout the generations. This is one of the great virtues of genetic algorithms with a properly defined mutation function. The mutation function stops the program from stocking in a local optimum and thus, the program always has the chance of finding a better solution.

Similar to the case of increasing the population size, since the program runtime directly increases by increasing the number of generations and stopping criteria, selecting the best value for the number of generations and applying it to all the situations is not optimal. Therefore, the best way to optimize the performance of the program based on the user need is to pick a high value for the number of generations, and let the user to choose the length of the runtime by defining the “time limit” and “stall time limit”.

In order to define the optimal population size in terms of time efficiency, another experiment was performed which shows the variation of the best fitness value achieved by increasing the population size while the time limit is set to 600 seconds (Figure 32).



**Figure 32- Variations of the best fitness value by increasing the population size with the time limit of 600 seconds.**

Comparing the process of generating layouts with small and large population sizes shows that the program has better performance with fewer populations when the time limit is rather short, since the program converges toward the closest local optimum and has enough time to search different configurations around the area and find a close result to the local optimum. When the population size is rather large, and the time limit is short, the program has more chance of locating the final optimum area, but doesn't have enough time for searching that area and finding the final optimum. Therefore, when the time limit is short, the program has a better performance with a small population size and when the time limit is rather large, the performance of the program raises by increasing the population size.

### **4.2.3. Results**

Results of running the program with 10 rooms and population size of 20 after 100 generations are shown in Figure 33, Figure 34, Figure 35, Figure 36 and Figure 37. These layouts have mostly satisfied their given topological objectives, but still their geometrical objectives could be improved. However, once the program reaches a local optimum with a low fitness value for adjacency requirements, it needs much more time to change the achieved configuration in favor of topological objectives since the topological objectives are given more weights in this program. Yet, it is possible to ask the user about this priority and direct the populations toward more compact forms.

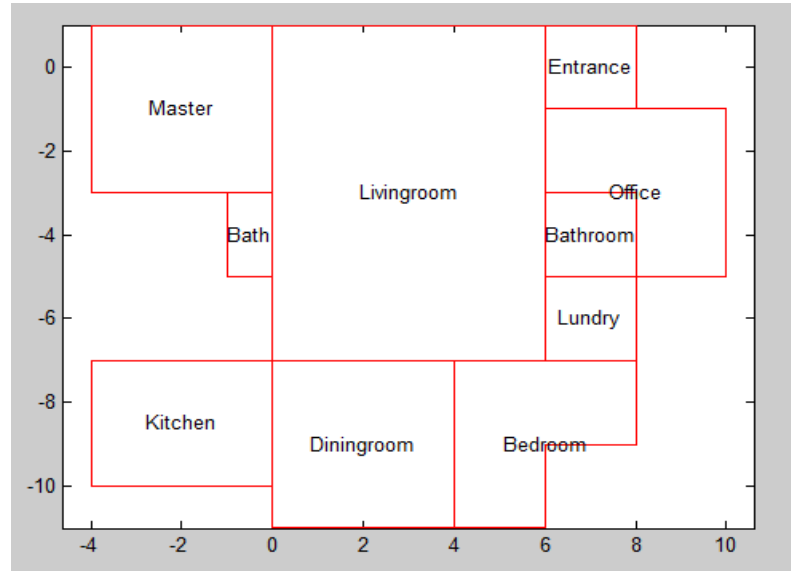


Figure 33- A result of running the program with 10 rooms (first example)



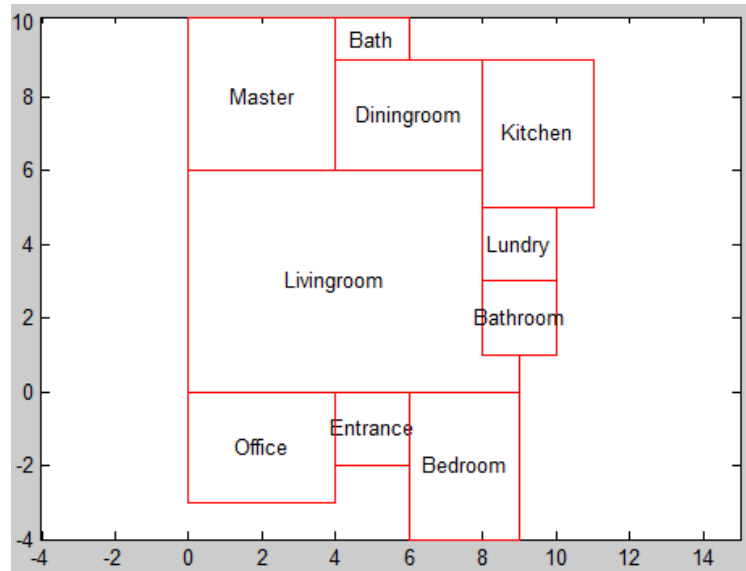
Figure 34- A result of running the program with 10 rooms (second example)



Figure 35- A result of running the program with 10 rooms (third example)



Figure 36- A result of running the program with 10 rooms (4<sup>th</sup> example)



**Figure 37- A result of running the program with 10 rooms (5<sup>th</sup> example)**

## 5. Future Directions

As discussed earlier, the prototype application was developed as a framework for investigation, and it presents many opportunities for future work:

- **Compatibility with more fitness functions:** It is convenient to define most of the design criteria by means of a sub-fitness function. Designers then would be able to direct the design toward their favorite style, standards, etc. by selecting and prioritizing different sub-fitness functions.
- **Improving the performance of the program:** Since the main objective for developing this research prototype was to solve complex design problems with a large number of rooms and adjacency relationships, the performance of the program still needs to be improved, using more genetic engineering techniques, varying the mutation function, and performing more parameter settings on variables that have impacts on the performance of the program.
- **Interactive environment:** Sometimes, the program generates a layout that the user can easily improve it by turning or slightly shifting a room in a layout while it may take many generations for the program to find that position. Thus, the performance of the program would be increased if the user can stop the generation at will, make the desirable changes, and start running the genetic algorithm with the new inputs.
- **Solving 3 dimensional layout problems:** By defining the basic element of space as a one-meter cube, and adjusting the fitness functions, the program can actually address spatial relationship between spaces in section as well as plan.

- Integrating more heuristic methods: The idea of using a database of expert systems and attachment rules in shape grammars could be used in this prototype to replace some of the random processes of generation. This would provide the users with a set of predefined standard items and rules which decrease the program runtime and create a knowledge-base program that embraces both design rules and generalized topological layouts.

## 6. Conclusions

In this work we explored different approaches to automatic space layout planning. Amongst different procedural, heuristic and evolutionary methods that have been presented for this problem, the genetic algorithm and engineering techniques were chosen due to their ability to optimize over constrained problems and generate novel solutions.

The genetic algorithm technique was adapted to the space layout planning problem in two phases: room level and building level. In the room level, we follow a genetic engineering approach to create space by combining basic blocks of one square foot to build larger blocks and ultimately a room. The user can provide inputs to the program in this phase to specify area, perimeter and proportion requirements for each room. In the building level, the best rooms generated from the first phase for different rooms are put together based on area, perimeter, proportion and adjacency requirement for the building layout provided by the user.

The results of running the program multiple times shows that the program is capable of generating multiple optimized solutions to a single problem while some of the solutions may seem novel to the user. At this stage, the user can establish the general direction of the design by choosing from presented solutions. The selected design layout can then be further improved by an architect.

This approach was implemented using MATLAB and its genetic algorithms toolbox. The current implementation can give up to 6 different practical designs for a building with 10 rooms after a users input in 10-15 minutes. The performance of the application can be further improved using more sophisticated programming languages and genetic engineering methods.

## Bibliography

- ACTLOC Users Guide (1999). CD601. College of Architecture and Urban planning, University of Washington.
- Arvin, S. A. and House, D. H. (1999). Modeling Architectural Design Objectives in Physically Based Space Planning. *Media and design process*, no. 2: 212-225.
- Balachandran, M. and Gero, J. S. (1987). Dimensioning of Architectural Floor Plans under Conflicting Objectives. *Environment and Planning B* (14), 29-37.
- Barzel, R., and Barr A. H. (1988). A Modeling System Based on Dynamic Constraints. *Computer Graphics, Proceeding of SIGGRAPH 88*, Ed. J. Dill, ACM SIGGRAPH, Atlanta, Georgia, pp 179-188.
- Beasley J.E, Bull D.R, Martin R.R. (1993). An Overview of Genetic Algorithms: Part I Fundamentals. *University Comp* 15:170.
- Cervone, G. and Kaufman K. and Michalski R. (2000). Experimental Validations of the learnable evolution model. *Evolutionary Computation CEC00*, pp 1064-1071.
- Casalaina, V. and Rittel, H. (1967). Morphologies of Floor Plans. *Paper for the Conference on Computer-Aided Building Design*.
- Damski, J. C. and Gero, J. S. (1997). An Evolutionary Approach to Generating Constraint-Based Space Layout Topologies, *CAAD Futures 1997 [Conference Proceedings / ISBN 0-7923-4726-9] München (Germany)*, pp. 855-864.
- Gero, J. S. (1996). Design Tools that Learn: A Possible CAD Future, in B. Kumar (ed.), *Information Processing in Civil and Structural Design*, Civil-Comp Press, Edinburgh, pp. 17-22.
- Gero, J. S. and Kazakov, V. (1995). Evolving Building Blocks for Design Using Genetic Engineering: A formal approach, in J. S. Gero and F. Sudweeks (eds), *Advances in Formal Design Methods for CAD*, IFIP-University of Sydney, Sydney, pp. 29-48.
- Gero, J. S. and Kazakov, V. (1996a). Evolving Building Blocks for Design Using Genetic Engineering: A formal approach, in Gero, J. S. (ed.) *Advances in Formal Design Methods for CAD*, Chapman and Hall, London, pp. 31-50.

- Gero, J. S. and Kazakov, V. (1996b). Spatial Layout Planning Using Evolved Design Genes, in C. Dagli, M. Akay, C. Chen, B. Fernandez and J. Ghosh (eds), *Smart Engineering Systems: Neural Networks, Fuzzy Logic and Evolutionary Programming*, ASME Press, New York, pp. 379-384.
- Gero, J. S. and Kazakov, V. (1998). Evolving Design Genes in Space Layout Problems, *Artificial Intelligence in Engineering* 12 (3):163-176.
- Gero, J. S. and Schnier, T. (1995). Evolving Representations of Design Cases and Their use in creative Design, in J. S. Gero, M. L. Maher and F. Sudweeks (eds), *Preprints Computational Models of Creative Design*, Key Centre of Design Computing, University of Sydney, pp. 343-368.
- Grason, J. (1970a). A Dual Linear Graph Representation for Space-Filling Location Problems of the Floor Plan Type. Gary T. Moore(ed.) *Emerging Methods in Environmental Design and Planning*, M.I.T. Press, pp. 170-178.
- Grason, J. (1970b). Fundamental Description of a Floor Plan Design Program. Sanoff, Henry, and Cohn, Sidney (eds.), EDRA1, *Proceedings of the 1st Annual Environmental Design Research Association Conference*, North Carolina State University, pp. 175-182.
- Grason, J. (1970c). Methods for the Computer-Implemented Solution of a Class of 'Floor Plan' Design Problems.
- Grason, J. (1971). An Approach to Computerized Space Planning using Graph Theory.
- Holland. J.H. (1975). *Adaptation in Natural and Artificial Systems*. MIT Press.
- House, D. H. and Kocmoud C. J. (1998). Continuous Cartogram Construction, *Proceedings of Visualization 98*, Research Triangle Park, North Carolina, pp 197-204.
- Jagielski, R. and Gero, J. S. (1997). A Genetic Programming Approach to the Space Layout Planning Problem, in R. Junge (ed.), *CAAD Futures 1997*, Kluwer, Dordrecht, pp. 875-884.
- Jo, J. H. (1993). A Computational Design Process Model Using a Genetic Evolution Approach, PhD Thesis, Department of Architectural and Design Science, University of Sydney, (unpublished).
- Jo, J. H. and Gero, J.S. (1995). A Genetic Search Approach to Space Layout Planning, *Architectural Science Review*, 38(1): 37-46.

- Jo, J. and Gero, J. S. (1996). Space Layout Planning Using an Evolutionary Approach, in M. Tan and R. Teh (eds), *Representation and use of design knowledge in evolutionary design*, National University of Singapore, pp.189-203.
- Jo, J. and Gero, J. S. (1998). Space Layout Planning Using an Evolutionary Approach, *Artificial Intelligence in Engineering* 12(3): 149-162.
- Kalay, E.Y. (2004). Architecture's New Media. Principles, Theories, and Methods of Computer-Aided Design. The MIT Press, Cambridge, Massachusetts.
- Krejcirik, M. (1969). Computer-Aided Plant Layout. *Computer-Aided Design*, pp. 7-19.
- Levin, P. H. (1964). Use of Graphs to Decide the Optimum Layout of Buildings. *Architects' Journal*.
- Michalek, J. (2001). Interactive Layout Design Optimization. MS Thesis, University of Michigan.
- Mitchell, M. (1996). An Introduction to Genetic Algorithm. The MIT Press, Cambridge, Massachusetts.
- Qin, H. (1998), A Physics-based Geometric Modeling and Design System.
- Qin, H., Mandal C., Vemuri B. C. (1998). Dynamic Catmull-Clark Subdivision Surfaces. *IEEE Transactions on Visualization and Computer Graphics* 4(3) 215-229.
- Qin, H. and Terzopoulos, D. (1995). "Dynamic NURBS Swung Surfaces for Physics-based Shape Design. *Computer-Aided Design* 27(2) 91-108.
- Reynolds, C. (1987). Flocks, herds, and schools: A Distributed Behavioral Model, Computer Graphics, *Proceedings of SIGGRAPH 87*, Ed M Stone, ACM SIGGRAPH, Anaheim, California, pp 17-24.
- Rosenman, M. A. (1995). An Edge Vector Representation for the Construction of 2 Dimensional Shapes, Environment and Planning B: *Planning and Design*, 22:191-212.
- Rosenman, M. A. and Gero, J.S. (1999). Evolving Designs by Generating Useful Complex Gene Structures, P. Bentley (Ed.), *Evolutionary Design by Computers*, Morgan Kaufmann, San Francisco, pp. 345-364.

- Schnier T., and Gero J.S., (1996). Learning Genetic Representations as Alternative to Hand-Coded Shape Grammars, in J. S. Gero and F. Sudweeks (eds), *Artificial Intelligence in Design'96*, Kluwer, Dordrecht, pp.39-57.
- Tsang EPK, Borrett JE, Kwan ACM. (1995). An Attempt to Map the Performance of a Range of Algorithm and Heuristic Combinations. Hybrid problems, hybrid solutions. *In Proceedings of AISB*. IOS Press . p. 203-16.
- Teague, Lavette C., Jr. (1970). Network Models of Configurations of Rectangular Parallelepipeds. Gary T. Moore (ed.), *Emerging Methods in Environmental Design and Planning*, M.I.T. Press, pp. 162-169.
- Terzopoulos D., Platt J., Barr A. H., Fleischer K. (1987). Elastically Deformable Models, Computer Graphic, *Proceeding of SIGGRAPH 87*, Ed. M. Stone, ACM SIGGRAPH, Anaheim, California, pp 205-214.
- Yoon, K. B. (1992). A Constraint Model of Space Planning. Southampton, UK: *Computational Mechanics Publications*.

## Appendix: Application source code

**Table 2- "add2right" function**

Function Purpose	This function takes “argBlock2” which is a genotype in the room level and adds it to the right side of “argBlock1” which is another genotype in the room level. “newBlock” is the result of this combination which makes a genotype in a next generation.
Application area	Room level
<pre> function newBlock=add2right(argBlock1, argBlock2) argBlock1=arrangedBlock(argBlock1); argBlock2=arrangedBlock(argBlock2); block1size=size(argBlock1); block2size=size(argBlock2); newBlock2=argBlock2; s=block1size(2); shift=1;  for j=1:s     for k=1:block2size(2)         if argBlock1(:,j)==newBlock2(:,k)             overlap=1;             break;         else             overlap=0;         end     end     if overlap==1         newBlock2=[argBlock2(1,:)+shift;argBlock2(2,:)];     end     if overlap==1,         shift=shift+1;     end end  blockSize=block1size(2)+block2size(2); newBlock=zeros(2,blockSize); newBlock(:,1:block1size(2))=argBlock1; newBlock(:,block1size(2)+1:end)=newBlock2; </pre>	

**Table 3- "add2up" function**

Function Purpose	This function takes “argBlock2” which is a genotype in
------------------	--

	the room level and adds it to the upper side of “argBlock1” which is another genotype in the room level. “newBlock” is the result of this combination which makes a genotype in a next generation.
<b>Application area</b>	Room level
<pre> function newBlock=add2up(argBlock1, argBlock2) argBlock1=arrangedBlock(argBlock1); argBlock2=arrangedBlock(argBlock2); block1size=size(argBlock1); block2size=size(argBlock2); newBlock2=argBlock2; s=block1size(2); shift=1;      for j=1:s,         for k=1:block2size(2),             if argBlock1(:,j)==newBlock2(:,k)                 overlap=1;                 break;             else                 overlap=0;             end         end         if overlap==1             newBlock2=[argBlock2(1,:);argBlock2(2,:)+shift];         end         if overlap==1,             shift=shift+1;         end     end  blockSize=block1size(2)+block2size(2); newBlock=zeros(2,blockSize); newBlock(:,1:block1size(2))=argBlock1; newBlock(:,block1size(2)+1:end)=newBlock2; </pre>	

**Table 4- "addLabel" function**

<b>Function Purpose</b>	This function takes the room coordinates (argBlock) and the room name (roomName) and prints the room name roughly in the center of gravity of the room.
<b>Application area</b>	Room & Building level
<pre> function named=addLabel(argBlock,roomName) length=argBlock(1,:); </pre>	

```

maxLength=max(length);
minLength=min(length);
x=(maxLength+minLength+1)/2;
height=argBlock(2,:);
maxHeight=max(height);
minHeight=min(height);
y=(maxHeight+minHeight+1)/2;
roomName=text(x,y,roomName);
set(roomName,'HorizontalAlignment','Center')

```

**Table 5- “Adjacancies\_GUI” function**

<b>Function Purpose</b>	This function raises a graphical user interface for taking the adjacency priorities from the user. For the sake of abbreviation, a repetitive part of this function is cut and replaced by three vertical dots.
<b>Application area</b>	Building level
<pre> function varargout = Adjacancies_GUI(varargin) % ADJACANCIES_GUI M-file for Adjacancies_GUI.fig %     ADJACANCIES_GUI, by itself, creates a new ADJACANCIES_GUI or %     raises the existing singleton.  gui_Singleton = 1; gui_State = struct('gui_Name',       mfilename, ...                   'gui_Singleton',  gui_Singleton, ...                   'gui_OpeningFcn', @Adjacancies_GUI_OpeningFcn, ...                   'gui_OutputFcn',  @Adjacancies_GUI_OutputFcn, ...                   'gui_LayoutFcn',  [] , ...                   'gui_Callback',   []); if nargin &amp;&amp; ischar(varargin{1})     gui_State.gui_Callback = str2func(varargin{1}); end  if nargout     [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:}); else     gui_mainfcn(gui_State, varargin{:}); end  % --- Executes just before Adjacancies_GUI is made visible. function Adjacancies_GUI_OpeningFcn(hObject, eventdata, handles, varargin) handles.output = hObject;  AllObjects = guihandles(hObject); NumberOfRooms = varargin{1}; </pre>	

```
RoomNames = varargin{2};

if NumberOfRooms > 0
    set(AllObjects.text22, 'String', RoomNames{1});
    set(AllObjects.text32, 'String', RoomNames{1});
end

if NumberOfRooms > 1
    set(AllObjects.text23, 'String', RoomNames{2});
    set(AllObjects.text33, 'String', RoomNames{2});
end

.
.
.

if NumberOfRooms > 9
    set(AllObjects.text41, 'String', RoomNames{10});
end

handles.one_0=0;
handles.two_0=0;
handles.two_1=0;
handles.three_0=0;
handles.three_1=0;
handles.three_2=0;
handles.four_0=0;
handles.four_1=0;
handles.four_2=0;
handles.four_3=0;
handles.five_0=0;
handles.five_1=0;
handles.five_2=0;
handles.five_3=0;
handles.five_4=0;
handles.six_0=0;
handles.six_1=0;
handles.six_2=0;
handles.six_3=0;
handles.six_4=0;
handles.six_5=0;
handles.seven_0=0;
handles.seven_1=0;
handles.seven_2=0;
handles.seven_3=0;
handles.seven_4=0;
handles.seven_5=0;
handles.seven_6=0;
handles.eight_0=0;
handles.eight_1=0;
handles.eight_2=0;
```

```

handles.eight_3=0;
handles.eight_4=0;
handles.eight_5=0;
handles.eight_6=0;
handles.eight_7=0;
handles.nine_0=0;
handles.nine_1=0;
handles.nine_2=0;
handles.nine_3=0;
handles.nine_4=0;
handles.nine_5=0;
handles.nine_6=0;
handles.nine_7=0;
handles.nine_8=0;
handles.ten_0=0;
handles.ten_1=0;
handles.ten_2=0;
handles.ten_3=0;
handles.ten_4=0;
handles.ten_5=0;
handles.ten_6=0;
handles.ten_7=0;
handles.ten_8=0;
handles.ten_9=0;

guidata(hObject, handles); % Update handles structure

% UIWAIT makes Adjacancies_GUI wait for user response
uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = Adjacancies_GUI_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;

function edit2_Callback(hObject, eventdata, handles)

handles.one=str2double(get(hObject,'String'));
guidata(hObject, handles);

function edit2_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUiControlBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function edit3_Callback(hObject, eventdata, handles)
function edit3_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUiControlBackgroundColor'))
    set(hObject,'BackgroundColor','white');

```

```

end

.
.
.

function edit111_Callback(hObject, eventdata, handles)
handles.ten_0=str2double(get(hObject,'String'));
guidata(hObject, handles);
function edit111_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)

adjPriorities=zeros(10);

adjPriorities(1,1)=handles.one_0;
adjPriorities(2,1)=handles.two_0;
adjPriorities(2,2)=handles.two_1;
adjPriorities(3,1)=handles.three_0;
adjPriorities(3,2)=handles.three_1;
adjPriorities(3,3)=handles.three_2;
adjPriorities(4,1)=handles.four_0;
adjPriorities(4,2)=handles.four_1;
adjPriorities(4,3)=handles.four_2;
adjPriorities(4,4)=handles.four_3;
adjPriorities(5,1)=handles.five_0;
adjPriorities(5,2)=handles.five_1;
adjPriorities(5,3)=handles.five_2;
adjPriorities(5,4)=handles.five_3;
adjPriorities(5,5)=handles.five_4;
adjPriorities(6,1)=handles.six_0;
adjPriorities(6,2)=handles.six_1;
adjPriorities(6,3)=handles.six_2;
adjPriorities(6,4)=handles.six_3;
adjPriorities(6,5)=handles.six_4;
adjPriorities(6,6)=handles.six_5;
adjPriorities(7,1)=handles.seven_0;
adjPriorities(7,2)=handles.seven_1;
adjPriorities(7,3)=handles.seven_2;
adjPriorities(7,4)=handles.seven_3;
adjPriorities(7,5)=handles.seven_4;
adjPriorities(7,6)=handles.seven_5;
adjPriorities(7,7)=handles.seven_6;
adjPriorities(8,1)=handles.eight_0;
adjPriorities(8,2)=handles.eight_1;
adjPriorities(8,3)=handles.eight_2;

```

```
adjPriorities(8,4)=handles.eight_3;
adjPriorities(8,5)=handles.eight_4;
adjPriorities(8,6)=handles.eight_5;
adjPriorities(8,7)=handles.eight_6;
adjPriorities(8,8)=handles.eight_7;
adjPriorities(9,1)=handles.nine_0;
adjPriorities(9,2)=handles.nine_1;
adjPriorities(9,3)=handles.nine_2;
adjPriorities(9,4)=handles.nine_3;
adjPriorities(9,5)=handles.nine_4;
adjPriorities(9,6)=handles.nine_5;
adjPriorities(9,7)=handles.nine_6;
adjPriorities(9,8)=handles.nine_7;
adjPriorities(9,9)=handles.nine_8;
adjPriorities(10,1)=handles.ten_0;
adjPriorities(10,2)=handles.ten_1;
adjPriorities(10,3)=handles.ten_2;
adjPriorities(10,4)=handles.ten_3;
adjPriorities(10,5)=handles.ten_4;
adjPriorities(10,6)=handles.ten_5;
adjPriorities(10,7)=handles.ten_6;
adjPriorities(10,8)=handles.ten_7;
adjPriorities(10,9)=handles.ten_8;
adjPriorities(10,10)=handles.ten_9;

handles.output=adjPriorities;

guidata(hObject, handles);

uiresume;
function text21_CreateFcn(hObject, eventdata, handles)
function text23_CreateFcn(hObject, eventdata, handles)
function text24_CreateFcn(hObject, eventdata, handles)
function text25_CreateFcn(hObject, eventdata, handles)
function text26_CreateFcn(hObject, eventdata, handles)
function text27_CreateFcn(hObject, eventdata, handles)
function text28_CreateFcn(hObject, eventdata, handles)
function text29_CreateFcn(hObject, eventdata, handles)
function text30_CreateFcn(hObject, eventdata, handles)
function text32_CreateFcn(hObject, eventdata, handles)
function text33_CreateFcn(hObject, eventdata, handles)
function text34_CreateFcn(hObject, eventdata, handles)
function text35_CreateFcn(hObject, eventdata, handles)
function text36_CreateFcn(hObject, eventdata, handles)
function text37_CreateFcn(hObject, eventdata, handles)
function text38_CreateFcn(hObject, eventdata, handles)
function text39_CreateFcn(hObject, eventdata, handles)
function text40_CreateFcn(hObject, eventdata, handles)
function text41_CreateFcn(hObject, eventdata, handles)
function result= areAdjacent(room1,room2)
perimeter1=buildPerimeter(room1);
```

```

perimeter2=buildPerimeter(room2);
commonPerim=0;
for k=1:size(perimeter1,2)-1
    thisSide=perimeter1(:,k:k+1);
    thisSide2=perimeter1(:,k+1:-1:k);
    for m=1:size(perimeter2,2)-1
        thatSide=perimeter2(:,m:m+1);
        if thisSide==thatSide | thisSide2==thatSide
            commonPerim=1;
            break;
        end
    end
end
end
if commonPerim==1
    result=1;
else
    result=0;
end
function scores = areaFitness(x,area,smoothness,proportion)
%smoothness value should be either 0 or 1
%proportion should be a number that represents the ratio of the desired
length to width

scores = zeros(size(x,1),1);

for j = 1:size(x,1)
    correctBlock=arrangedBlock(x{j});
    sizeX=size(correctBlock);
    AreaX=sizeX(2);
    if smoothness==1
        smoothnessScore=smoothnessFitness(x{j});
    else
        smoothnessScore=0;
    end
    convexityScore=convexityFitness(x{j});
    proportionScore=abs(proportionFitness(x{j}));
    if proportionScore==0 | proportion==0
        proportionScore=50;
    elseif proportionScore<1
        proportionScore=1/proportionScore;
    end
    proportionScore=proportionScore/proportion;

    if proportionScore>3
        proportionScore=50;
    elseif proportionScore>=2
        proportionScore=10;
    end
    proportionScore=proportionScore*1;%This is for changing the
importance of proportion fitness as we like
    AreaScore=abs(AreaX-area);

```

```

    AreaScore=AreaScore*1.3; %This is for changing the importance of
area fitness as we like
    smoothnessScore=smoothnessScore*1.6;
    scores(j)=AreaScore+smoothnessScore+proportionScore+convexityScore;
end
function newBlock=arrangedBlock(argBlock)
s=size(argBlock);
length=s(2);
newBlock=[];
%Putting Xs in order
for j=0:length-1
    for i=0:length-1
        if argBlock(i*2+1)==j
            newBlock=[newBlock,argBlock(:,i+1)];
            argBlock(:,i+1)=[-0.5,-0.5];
        end
    end
end
%Putting Ys in order
i=0;
while i<=length-2
    if newBlock(i*2+1)==newBlock(i*2+3)
        if newBlock(i*2+2)>newBlock(i*2+4)
            x=newBlock(i*2+2);
            newBlock(i*2+2)=newBlock(i*2+4);
            newBlock(i*2+4)=x;
            i=0;
        elseif newBlock(i*2+2)==newBlock(i*2+4)
            newBlock(:,i+1)=[];
            length=length-1;
        else
            i=i+1;
        end
    else
        i=i+1;
    end
end
%Filling the blanks between Xs
fixedBlock=[];
i=0;
while i<=length-2
    if newBlock(i*2+3)>1+newBlock(i*2+1)
        fixedBlock=[fixedBlock,newBlock(:,i+1)];
        fixedBlock=[fixedBlock,newBlock(:,i+2)];
        fixedBlock(i*2+3)=newBlock(i*2+3)-1;
        fixedBlock=[fixedBlock,newBlock(:,i+2:end)];
        newBlock=fixedBlock;
        length=length+1;
        i=i+1;
    else
        fixedBlock=[fixedBlock,newBlock(:,i+1)];

```

```

        i=i+1;
    end
end

%Filling the blanks between Ys
for i=0:length-2
    if newBlock(i*2+1)==newBlock(i*2+3)
        if newBlock(i*2+2)+1<newBlock(i*2+4)
            newBlock=[newBlock(:,1:i+1),newBlock(:,i+1:end)];
            newBlock(i*2+4)=newBlock(i*2+4)+1;
            i=i-1;
        end
    end
end
end
end

```

**Table 6- "arrangedBlock3" function**

<b>Function Purpose</b>	This function rearranges the basic elements of space in a room and puts them in order. It also fills the possible wholes inside the rooms.
<b>Application area</b>	Room level
<pre> function newBlock=arrangedBlock3(argBlock) s=size(argBlock); length=s(2); newBlock=[]; m=min(argBlock,[],2); minX=m(1); %Putting Xs in order for j=minX:length-1+minX     for i=0:length-1         if argBlock(i*2+1)==j             newBlock=[newBlock,argBlock(:,i+1)];             argBlock(:,i+1)=[-0.5,-0.5];         end     end end end %Putting Ys in order i=0; while i&lt;=length-2     if newBlock(i*2+1)==newBlock(i*2+3)         if newBlock(i*2+2)&gt;newBlock(i*2+4)             x=newBlock(i*2+2);             newBlock(i*2+2)=newBlock(i*2+4);             newBlock(i*2+4)=x;             i=0;         elseif newBlock(i*2+2)==newBlock(i*2+4) </pre>	

```

        newBlock(:,i+1)=[];
        length=length-1;
    else
        i=i+1;
    end
else
    i=i+1;
end
end
%Filling the blanks between Xs
fixedBlock=[];
i=0;
while i<=length-2
    if newBlock(i*2+3)>1+newBlock(i*2+1)
        fixedBlock=[fixedBlock,newBlock(:,i+1)];
        fixedBlock=[fixedBlock,newBlock(:,i+2)];
        fixedBlock(i*2+3)=newBlock(i*2+3)-1;
        fixedBlock=[fixedBlock,newBlock(:,i+2:end)];
        newBlock=fixedBlock;
        length=length+1;
        i=i+1;
    else
        fixedBlock=[fixedBlock,newBlock(:,i+1)];
        i=i+1;
    end
end
%Filling the blanks between Ys
for i=0:length-2
    if newBlock(i*2+1)==newBlock(i*2+3)
        if newBlock(i*2+2)+1<newBlock(i*2+4)
            newBlock=[newBlock(:,1:i+1),newBlock(:,i+1:end)];
            newBlock(i*2+4)=newBlock(i*2+4)+1;
            i=i-1;
        end
    end
end
end
end

```

**Table 7- "buildBlock" function**

<b>Function Purpose</b>	This function takes the coordinates of the defined basic elements of space in a room and prints the outline of the room.
<b>Application area</b>	Room level
<pre> function Block=buildBlock(argDownLeft)     Size=size(argDownLeft); </pre>	

```

temp=argDownLeft;
MinDown=[];
while size(temp)>[0,0]
    Min=min(temp,[],2);
    MinRow=Min(1);
    tempSize=size(temp);
    rowNum=tempSize(2);
    temp2=[];
    i=1;
    while rowNum>=i
        if temp(1,i)==MinRow
            temp2=[temp2 temp(:,i)];
            temp(:,i)=[];
            i=i-1;
            rowNum=rowNum-1;
        end
        i=i+1;
    end
    MinDown=[MinDown, min(temp2,[],2)];
    clear temp2;
    clear Min;
    clear MinRow;
    clear tempSize;
    clear rowNum;
    clear temp2;
    s=size(MinDown);
    down=[];
    for i=1:s(2)
        down=[down,MinDown(:,i),MinDown(:,i)+1];
        tempSize=size(down);
        down(2,tempSize(2))=down(2,tempSize(2)-1);
    end
end
clear s;
clear tempSize;
clear temp;
temp=[argDownLeft(1,:);argDownLeft(2,:)+1];
MaxUp=[];
while size(temp)>[0,0]
    Max=max(temp,[],2);
    MaxRow=Max(1);
    tempSize=size(temp);
    rowNum=tempSize(2);
    temp2=[];
    i=1;
    while rowNum>=i
        if temp(1,i)==MaxRow
            temp2=[temp2 temp(:,i)];
            temp(:,i)=[];
            i=i-1;
            rowNum=rowNum-1;
        end
    end
end

```

```

        end
        i=i+1;
    end
    MaxUp=[MaxUp, max(temp2, [], 2)];
    clear temp2;
    clear tempSize;
    clear rowNum;
    clear temp2;
    s=size(MaxUp);
    Up=[];
    for i=1:s(2)
        Up=[Up, MaxUp(:, i)+1, MaxUp(:, i)];
        tempSize=size(Up);
        Up(2, tempSize(2)-1)=Up(2, tempSize(2));
    end
end
Block=[down, Up];
%eliminating repatative columns
s=size(Block);
blockSize=s(2);
i=1;
while blockSize>i
    if Block(:, i)==Block(:, i+1)
        Block(:, i+1)=[];
        i=i-1;
        blockSize=blockSize-1;
    end;
    i=i+1;
end
%eliminating extra points in horizontal lines
s=size(Block);
blockSize=s(2);
i=1;
while blockSize-1>i
    if Block(2, i)==Block(2, i+1) & Block(2, i+1)==Block(2, i+2)
        Block(:, i+1)=[];
        i=i-1;
        blockSize=blockSize-1;
    end;
    i=i+1;
end
Close=CloseFigure(Block);
plotBlock(Close);

```

**Table 8- "buildPerimeter" function**

Function Purpose	This function takes the coordinates of the defined basic elements of space in all the rooms in a layout and figures
------------------	---

	out the perimeter of the building.
Application area	Building level
<pre> function Block=buildPerimeter(argDownLefts) s=size(argDownLefts); Size=s(2); allSides=cell(0,0); %figuring out the perimeter sides for i=1:Size     currentPoint=argDownLefts(:,i);     DownSide=[currentPoint(1), currentPoint(1)+1; currentPoint(2), currentPoint(2)];     LeftSide=[currentPoint(1), currentPoint(1); currentPoint(2), currentPoint(2)+1];     UpSide=[currentPoint(1), currentPoint(1)+1; currentPoint(2)+1, currentPoint(2)+1];     RightSide=[currentPoint(1)+1, currentPoint(1)+1; currentPoint(2), currentPoint(2)+1];     allSides=cat(2,allSides,DownSide);     allSides=cat(2,allSides,LeftSide);     allSides=cat(2,allSides,UpSide);     allSides=cat(2,allSides,RightSide); end  tempSize=size(allSides); cellSize=tempSize(2);  for i=1:cellSize-1     for j=i+1:cellSize         if allSides{i}==allSides{j},             allSides{j}=[-1];             allSides{i}=[-1];             break,         end     end end perimeterSides=cell(0,0); for i=1:cellSize     if allSides{i}==[-1],     else         perimeterSides=cat(2,perimeterSides,allSides{i});     end end  %building the perimeter s=size(perimeterSides); cellSize=s(2); perimeter=[]; for k=1:cellSize     if perimeterSides{k}==[-1], </pre>	



```

eFound=0;
blockFound=0;
while blockFound==0;
    if iscell(these{number-a})
        that=getRoom(these{number-a},eFound);
        blockFound=1;
    elseif strcmp(these{number-a},'empty')
        eFound=eFound+1;
        a=a+1;
    else
        a=a+1;
    end
end
else
    that=these{number};
end

```

**Table 10- "checkBlocks" function**

<b>Function Purpose</b>	This function takes a number and a cell array that contains all of the rooms (including evolved ones) and returns the room in a combination that the number refers to.
<b>Application area</b>	Building level
<pre> function that=checkBlocks(these,number) if iscell(these{number})     that=getRoom(these{number},-1); elseif size(these{number})==[0 0]%which means block content is []     a=1;     eFound=0;     blockFound=0;     while blockFound==0;         if iscell(these{number-a})             that=getRoom(these{number-a},eFound);             blockFound=1;         elseif size(these{number-a})==[0 0]             eFound=eFound+1;             a=a+1;         else             a=a+1;         end     end else     that=these{number}; end </pre>	

**Table 11- "CloseFigure" function**

<b>Function Purpose</b>	This function takes the coordinate of the first corner of a room and adds it to the end of the coordinate matrix to close the form while it is being printed.
<b>Application area</b>	Room level
<pre>function newFigure = CloseFigure(argFigure) dim = size(argFigure); newFigure = zeros(dim + [0 1]); newFigure(:,1:dim(2)) = argFigure; newFigure(:,dim(2)+1) = argFigure(:,1);</pre>	

**Table 12- "combinationFitness" function**

<b>Function Purpose</b>	This function takes the building layout and the user preferences about the building and evaluates the layout accordingly.
<b>Application area</b>	Building level
<pre>function scores = combinationFitness(x,area,smoothness,proportion,RoomNames, adjacancies) %smoothness value should be either 0 or 1 %proportion should be a number that represents the ratio of the desired length to width %area value should be either 0 or 1 scores = zeros(size(x,1),1); for j = 1:size(x,1)     thisComb=x{j};     rooms=thisComb{2};     names=thisComb{1};     cellSize=size(rooms);     numberOfRooms=cellSize(2);     GroupedRooms=[];     for i=1:numberOfRooms %creating a matrix that contains all of the rooms continually         GroupedRooms=cat(2,GroupedRooms,rooms{i});     end      builtGroupedRooms=buildBlock4(GroupedRooms);     perimeter=buildPerimeter(GroupedRooms);     adjacencyScore= combsAdjacencyFitness(RoomNames,adjacancies,thisComb,perimeter);      if smoothness==1         smoothnessScore=CombsSmoothFitness(GroupedRooms);</pre>	

```

else
    smoothnessScore=0;
end

if area==1
    AreaScore=CombsAreaFitness (rooms);
else
    AreaScore=0;
end

proportionScore=abs (CombsProFitness (rooms)-proportion);

proportionScore=proportionScore*0;%This is for changing the
importance of proportion fitness as we like
AreaScore=AreaScore*1; %This is for changing the importance of area
fitness as we like

scores (j)=AreaScore+3*smoothnessScore+proportionScore+6*adjacencyScore;
end

```

**Table 13- "combiningRooms" function**

<b>Function Purpose</b>	This function runs the genetic algorithm in the building level and prints out the best results.
<b>Application area</b>	Building level
<pre> function result = combiningRooms (numberOfRooms, area, smoothness, proportion, AllTheRooms, RoomNames, adjacancies) %area value should be either 0 or 1 %smoothness value should be either 0 or 1 %proportion should be a number that represents the ratio of the desiered %length to width save('savings2', 'numberOfRooms', 'area', 'smoothness', 'proportion', 'AllTheRooms', 'RoomNames', 'adjacancies') pSize=90; options = struct( ... 'PopulationType','custom', ... 'PopInitRange',[0;1], ... 'PopulationSize',pSize, ... 'EliteCount',1, ... 'CrossoverFraction',.5, ... 'MigrationDirection','forward', ... 'MigrationInterval',1, ... 'MigrationFraction',0.1, ... 'Generations',12, ... 'TimeLimit',4000, ... </pre>	

```

'FitnessLimit',-Inf, ...
'StallGenLimit',10, ...
'StallTimeLimit',2000, ...
'InitialPopulation',[], ...
'InitialScores',numberOfRooms*90*ones(pSize,1), ...
'PlotInterval',1, ...
'CreationFcn',@(NVARs,FitnessFcn,options) FirstPopulation
(NVARs,FitnessFcn,options,numberOfRooms,AllTheRooms,RoomNames), ...
'FitnessScalingFcn',@fitscalingrank, ...
'SelectionFcn',@selectionuniform, ...
'CrossoverFcn',@CrossOverCombinations, ...
'MutationFcn',@mutate_permutation3, ...
'HybridFcn',[], ...
'Display','final', ...
'PlotFcns',@plotCombinations, ...
'OutputFcns',[], ...
'Vectorized','on');

options = gaoptimset(options,'PlotFcns',{@gaplotscorediversity
@gaplotrange @SpacePlot @SpacePlot2 @plotCombinations });

gaproblem = struct( ...
'fitnessfcn',@(x)
combinationFitness(x,area,smoothness,proportion,RoomNames,adjacancies),
...
'nvars',1, ...
'options',options);

[x fval reason output population scores] = ga(gaproblem);
bestScores=cell(0,0);
result=cell(0,0);
lastScore=-1;
thisScore=-1;
index=[1:pSize];
index=index';
scoresWithIndex=[scores,index];
sortedresult=sortrows(scoresWithIndex,1);

k=0;
for j=1:6
    i=sortedresult(pSize+j+k);
    thisScore=sortedresult(j+k);

    while thisScore==lastScore
        k=k+1;
        lastScore=thisScore;
        i=sortedresult(pSize+j+k);
        thisScore=sortedresult(j+k);
    end
    lastScore=thisScore;
    result{j}=population{i};

```

```

end

function score=combsAdjacencyFitness(NamesInOrder, adjacancies,
combination, housePerim)
rooms=combination{2};
names=combination{1};
s=size(adjacancies,1);
z=zeros(s,1);
adjacancies=[adjacancies,z];
score=0;

for i=2:size(NamesInOrder,2)
    for j=2:i
        if adjacancies(i,j)%if the two rooms are better to be adjacent
            %checking their adjacancies in the combination
            shouldEnd=0;
            for k=1:size(names,2)
                %finding the room's place in the combination
                if strcmp(names{k},NamesInOrder{i})
                    for l=1:size(names,2)
                        if strcmp(names{l},NamesInOrder{j-1})
                            if areAdjacent(rooms{k},rooms{l})
                                thisScore=0;
                            else
                                thisScore=adjacancies(i,j);
                            end
                            shouldEnd=1;
                        end
                    end
                    if shouldEnd
                        break
                    end
                end
            end
            if shouldEnd
                break
            end
        end
        score=score+thisScore;
    end
end

%Now we are checking which rooms are adjacent to the outside...
score3=1;
for i=1:size(NamesInOrder,2)
    for j=1:size(names,2)
        shouldBreak=0;
        if strcmp(names{j},NamesInOrder{i})
            if adjacancies(i,1)

```



```

for i=1:numberOfRooms
    GroupedRooms=cat(2,GroupedRooms,argGroupedRooms{i});%creating a
matrix that contains all of the rooms continually
end
arrangedGroupedRooms=arrangedBlock3(GroupedRooms);
s=size(arrangedGroupedRooms);
area=s(2);

score=area;

```

**Table 15- "CombsProFitness" function**

<b>Function Purpose</b>	This function determines the proportions of a given building layout
<b>Application area</b>	Building level
<pre> function value=CombsProFitness(argGroupedRooms) cellSize=size(argGroupedRooms); numberOfRooms=cellSize(2); GroupedRooms=[];  for i=1:numberOfRooms     GroupedRooms=cat(2,GroupedRooms,argGroupedRooms{i});%creating a matrix that contains all of the rooms continually end  builtGroupedRooms=buildBlock4(GroupedRooms);  width=max(builtGroupedRooms(1,:)); length=max(builtGroupedRooms(2,:)); if width==0     value=10; %this would never happen... else     value=length/width; end </pre>	

**Table 16- "CombsSmoothFitness" function**

<b>Function Purpose</b>	This function evaluates a building layout based on its compactness
<b>Application area</b>	Building level
<pre> function score = CombsSmoothFitness(builtGroupedRooms)  s=size(builtGroupedRooms); sides=s(2); </pre>	

```
score=sides-4;
```

**Table 17- "convexityFitness" function**

<b>Function Purpose</b>	This function evaluates a room based on its degree of concavity
<b>Application area</b>	Room level
<pre>function fitnessScore=convexityFitness(argRoom) s=size(argRoom,2); maximumX=argRoom(1); for i=0:s-1     if argRoom(2*i+1)&gt;maximumX         maximumX=argRoom(2*i+1);     end end maximumY=argRoom(2); for j=0:s-2     if argRoom(2*j+2)&gt;maximumY         maximumY=argRoom(2*j+2);     end end xCenter=floor(maximumX/2); yCenter=floor(maximumY/2); fitnessScore=10; for k=1:s     if argRoom(:,k)==[xCenter;yCenter]         fitnessScore=0;         break;     end end end</pre>	

**Table 18- "CreatBuilding\_GUI" function**

<b>Function Purpose</b>	This function raises a graphical user interface for taking the criteria of creating a building from the user and prints out the results
<b>Application area</b>	Building level
<pre>function varargout = CreatBuilding_GUI(varargin) % CREATBUILDING_GUI M-file for CreatBuilding_GUI.fig %CREATBUILDING_GUI, by itself, creates a new CREATBUILDING_GUI or raises the existing singleton. gui_Singleton = 1; gui_State = struct('gui_Name',      mfilename, ...                   'gui_Singleton',  gui_Singleton, ...</pre>	

```

        'gui_OpeningFcn', @CreatBuilding_GUI_OpeningFcn, ...
        'gui_OutputFcn', @CreatBuilding_GUI_OutputFcn, ...
        'gui_LayoutFcn', [], ...
        'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end

% --- Executes just before CreatBuilding_GUI is made visible.
function CreatBuilding_GUI_OpeningFcn(hObject, eventdata, handles,
varargin)
handles.user_rooms = 7;
handles.user_area = 0;
handles.user_perimeter = 0;
handles.user_ratio=1;

handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes CreatBuilding_GUI wait for user response (see UIRESUME)
uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = CreatBuilding_GUI_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;

function edit1_Callback(hObject, eventdata, handles)
user_rooms=str2double(get(hObject,'String'));
handles.user_rooms=user_rooms;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function edit1_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUiControlBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function edit2_Callback(hObject, eventdata, handles)
user_ratio=str2double(get(hObject,'String'));
handles.user_ratio=user_ratio;

```

```

guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function edit2_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUiControlBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
user_area = handles.user_area;
user_ratio = handles.user_ratio;
user_perimeter = handles.user_perimeter;
user_rooms=handles.user_rooms;

AllTheRooms=cell(0,0);
for j=1:user_rooms
    AllTheRooms{j}=CreatingRooms_GUI;
end
close;

RoomNames=cell(0,0);
for j=1:user_rooms
    currentRoom=AllTheRooms{j};
    RoomNames{j}=currentRoom{2};
end

adjacancies=Adjacancies_GUI(user_rooms,RoomNames);
close;

result=combinigRooms(user_rooms,user_area,user_perimeter,user_ratio,Al
lTheRooms,RoomNames,adjacancies);
S=size(result);
nOfCombs=S(2);
close;

for i = 1:6
    if nOfCombs>=1,
        subplot(2,3,i,'align');
        genotype = result{i};
        names=genotype{1};
        rooms=genotype{2};
        s=size(rooms);
        for j=1:s(2)
            thisRoom=rooms{j};
            buildBlock(thisRoom);
            hold on;
            addLabel(thisRoom,names{j});
            hold on;
        end
    end
end

```

```

        hold off;
        nOfCombs=nOfCombs-1;
    end
end

handles.output=result;
guidata(hObject, handles);
uiresume;
function uipanel3_CreateFcn(hObject, eventdata, handles)
function radiobutton1_Callback(hObject, eventdata, handles)
handles.user_area=1;
guidata(hObject, handles);
function radiobutton3_Callback(hObject, eventdata, handles)
handles.user_area=0;
guidata(hObject, handles);

function radiobutton4_Callback(hObject, eventdata, handles)
handles.user_perimeter=1;
guidata(hObject, handles);
function radiobutton5_Callback(hObject, eventdata, handles)
handles.user_perimeter=0;
guidata(hObject, handles);
function uipanel12_CreateFcn(hObject, eventdata, handles)
function axes1_CreateFcn(hObject, eventdata, handles)
function uipanel11_CreateFcn(hObject, eventdata, handles)
function uipanel6_CreateFcn(hObject, eventdata, handles)
function uipanel9_CreateFcn(hObject, eventdata, handles)
function uipanel8_CreateFcn(hObject, eventdata, handles)

```

**Table 19- "CreatePopulation" function**

<b>Function Purpose</b>	This function creates basic elements of space as initial populations for the room level.
<b>Application area</b>	Room level
<pre> function population = CreatePopulation(NVARS,FitnessFcn,options)  totalPopulationSize = sum(options.PopulationSize);  population = cell(totalPopulationSize,1); for i=1:totalPopulationSize     population{i}=[0; 0]; end </pre>	

**Table 20- “creatingRooms” function**

Function Purpose	This function runs the genetic algorithm in the room level and prints out the best results.
Application area	Room level
<pre> function result=creatingRooms(area,smoothness,proportion) %smoothness value should be either 0 or 1 %proportion should be a number greater than 0 options = struct( ... 'PopulationType','custom', ... 'PopInitRange',[0;1], ... 'PopulationSize',28, ... 'EliteCount',1, ... 'CrossoverFraction',.5, ... 'MigrationDirection','forward', ... 'MigrationInterval',1, ... 'MigrationFraction',0.1, ... 'Generations',7, ... 'TimeLimit',18, ... 'FitnessLimit',0, ... 'StallGenLimit',3, ... 'StallTimeLimit',18, ... 'InitialPopulation',[], ... 'InitialScores',area*2*ones(28,1), ... 'PlotInterval',1, ... 'CreationFcn',@CreatePopulation, ... 'FitnessScalingFcn',@fitscalingrank, ... 'SelectionFcn',@selectionstochunif, ... 'CrossoverFcn',@CrossOverFcn, ... 'MutationFcn',@mutate_permutation, ... 'HybridFcn',[], ... 'Display','final', ... 'PlotFcns',@plotRooms, ... 'OutputFcns',[], ... 'Vectorized','on');  options = gaoptimset(options,'PlotFcns',{@gaplotscorediversity @gaplotrange @plotRooms });  gaproblem = struct( ... 'fitnessfcn',@(x) areaFitness(x,area,smoothness,proportion), ... 'nvars',1, ... 'options',options);  [x fval reason output population scores] = ga(gaproblem); bestScores=[]; result=cell(1,0); for i=1:28 </pre>	

```

    if scores(i)<area/5+2.5 %This number could vary depend on the level
of accuracy we want
        bestScores=[bestScores,scores(i)];
        result=[result,population{i}];
    end
end

```

**Table 21- “CreatingRooms GUI” function**

Function Purpose	This function raises a graphical user interface for taking the criteria of creating a room from the user
Application area	Room level
<pre> function varargout = CreatingRooms_GUI(varargin) % CREATINGROOMS_GUI M-file for CreatingRooms_GUI.fig %     CREATINGROOMS_GUI, by itself, creates a new CREATINGROOMS_GUI or raises the existing singleton. gui_Singleton = 1; gui_State = struct('gui_Name',       mfilename, ...                   'gui_Singleton',  gui_Singleton, ...                   'gui_OpeningFcn', @CreatingRooms_GUI_OpeningFcn, ...                   'gui_OutputFcn',  @CreatingRooms_GUI_OutputFcn, ...                   'gui_LayoutFcn',  [], ...                   'gui_Callback',    []); if nargin &amp;&amp; ischar(varargin{1})     gui_State.gui_Callback = str2func(varargin{1}); end  if nargout     [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:}); else     gui_mainfcn(gui_State, varargin{:}); end  % --- Executes just before CreatingRooms_GUI is made visible. function CreatingRooms_GUI_OpeningFcn(hObject, eventdata, handles, varargin) handles.user_area = 12; handles.user_ratio = 1.5; handles.user_smoothness = 1; handles.user_roomName='Bedroom'; handles.FinalRooms=cell(1,6); handles.output = cell(1,6); % Update handles structure guidata(hObject, handles);  % UIWAIT makes CreatingRooms GUI wait for user response (see UIRESUME) </pre>	

```

uiwait(handles.figure1);

function varargout = CreatingRooms_GUI_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;

function edit1_Callback(hObject, eventdata, handles)

user_roomName=get(hObject,'String');
handles.user_roomName=user_roomName;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function edit1_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function slider1_Callback(hObject, eventdata, handles)
function slider1_CreateFcn(hObject, eventdata, handles)
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end

function slider2_Callback(hObject, eventdata, handles)
function slider2_CreateFcn(hObject, eventdata, handles)
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end

function edit2_Callback(hObject, eventdata, handles)

area=get(hObject,'String');
areaNum=str2double(get(hObject,'String'));
handles.user_area=areaNum;
guidata(hObject, handles);
function edit2_CreateFcn(hObject, eventdata, handles)
function edit4_Callback(hObject, eventdata, handles)
user_ratio=get(hObject,'String');
handles.user_ratio=str2double(user_ratio);
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function edit4_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

```

```

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
user_area = handles.user_area;
user_ratio = handles.user_ratio;
user_smoothness = handles.user_smoothness;
user_roomName=handles.user_roomName;

best_Rooms=creatingRooms(user_area,user_smoothness,user_ratio);
s=size(best_Rooms);
numberOfRooms=s(2);
close;

FinalRooms=cell(0,0);
i=1;
j=0;
k=0;%This is for keeping track of redundant generated Rooms.
l=0;% This is for stopping the program from running the GA more than 3
times for each room.
repetition=0;
nOfRooms=0;
while i<=6 & l<7,
    if numberOfRooms>=i-j+k,
        Room = best_Rooms{i-j+k};
        pR=arrangedBlock(Room);
        for m=1:5
            if nOfRooms>=m
                pF=arrangedBlock(FinalRooms{m});
                if size(pF)==size(pR),
                    if pF==pR,
                        repetition=1;
                    end
                end
            end
        end
        if repetition==0,
            subplot(2,3,i,'align');
            buildBlock(Room);
            addLabel(Room,user_roomName);
            FinalRooms{i}=Room;
            nOfRooms=nOfRooms+1;
            i=i+1;
        else
            k=k+1;
            repetition=0;
        end
    else
        j=i-1;
        l=l+1;
        k=0;
        best Rooms=creatingRooms(user_area,user_smoothness,user_ratio);

```

```

        s=size(best_Rooms);
        numberOfRooms=s(2);
        close;
    end
end
fs=size(FinalRooms);
n=fs(2);
h=0;
while n<6 & n>0
    subplot(2,3,n+1,'align');
    Room=FinalRooms{h+1};
    buildBlock(Room);
    addLabel(Room,user_roomName);
    FinalRooms{n+1}=Room;
    n=n+1;
    h=h+1;
end
RoomsNname=cell(1,2);
RoomsNname{1}=FinalRooms;
RoomsNname{2}=user_roomName;
handles.output=RoomsNname;
guidata(hObject, handles);
uiresume;

% --- Executes on button press in radiobutton1.
function radiobutton1_Callback(hObject, eventdata, handles)
handles.user_smoothness=1;
guidata(hObject, handles);
function radiobutton2_Callback(hObject, eventdata, handles)
handles.user_smoothness=0;
guidata(hObject, handles);
function axes1_CreateFcn(hObject, eventdata, handles)

```

**Table 22- “CrossOverCombinations” function**

<b>Function Purpose</b>	This function creates evolved genes out of the good combinations of rooms and performs the crossover function
<b>Application area</b>	Building level
<pre> function xoverKids = CrossOverCombinations(parents,options,NVARS, ...     FitnessFcn,thisScore,thisPopulation) %finding the 6 best parents bestPops=20; newBlocks=0; existingBlocks=0; deleteBlocks=0; pNum=length(parents); </pre>	

```

indxS=[1:pNum];
indxS=indxS';
scoresWithIndex=[thisScore,indxS];
sortedresult=sortrows(scoresWithIndex,1);
bestPups=cell(0,0);
for j=1:pNum
    i=sortedresult(pNum+j);
    Pups{j}=thisPopulation{i};
end
bestPups=Pups(1:bestPops);
%finding a common pattern between the best parents
for i=1:bestPops-1
    moreThan2=0;
    thisP=bestPups{i};
    NofRooms=size(thisP{1},2);
    thisNames=thisP{1};
    thisRooms=thisP{2};
    mFound=0;
    for j=i+1:bestPops
        thatP=bestPups{j};
        thatNames=thatP{1};
        thatRooms=thatP{2};
        %this is to make sure that the combinations that we are
        %comparing to make blocks are not exactly the same
        if isequal(thisP,thatP)
        elseif isequal(thisRooms{1},thatRooms)
        elseif isequal(thisRooms,thatRooms{1})
        else
            Found=0;
            for k=1:NofRooms-1
                %this is for reaching out to block contents
                thisN=checkBlockNames(thisNames,k);
                thisR=checkBlocks(thisRooms,k);
                %this is to move all roomes to the center of coordinanates
                %to be able to compare them....
                [thisRmoved, xToMove, yToMove]= moveRoom(thisR,[0;0]);
                for l=1:NofRooms-1
                    thatN=checkBlockNames(thatNames,l);
                    thatR=checkBlocks(thatRooms,l);
                    if size(thisN)==size(thatN)
                        if strcmp(thisN,thatN)
                            if size(thisR)==size(thatR)
                                [thatRmoved, xToTMove, yToTMove]=moveRoom (thatR, [0;0]);
                                if thisRmoved==thatRmoved
                                    thisCell=cell(0,0);
                                    thisCell{1}=thisN;
                                    auxiCell=cell(0,0);
                                    auxiCell{1}=thisR;
                                    alikeRooms=1;
                                    for m=k+1:NofRooms
                                        checkedThisName=checkBlockNames(thisNames,m);

```

```

nFound=0;
for n=1+1:NofRooms
    if strcmp (checkedThisName,checkBlockNames (thatNames,n))
        checkedThisRoom=checkBlocks (thisRooms,m);
        checkedThatRoom=checkBlocks (thatRooms,n);
        if size (checkedThisRoom)==size (checkedThatRoom)
            thisRoom2=moveRoom2 (checkedThisRoom, [xToMove;yToMove]);
            thatRoom2=moveRoom2 (checkedThatRoom, [xToTMove;yToTMove]);
            if thisRoom2==thatRoom2
                %making an evolved gen and replace it in the population
                nFound=1;
                mFound=1;
                alikeRooms=alikeRooms+1;
                Found=Found+1;
                if moreThan2==1
                    for y=m:NofRooms
                        if strcmp (thisNames{y}, 'empty')
                            thisNames{y}=checkBlockNames (thisNames,y);
                            thisRooms{y}=checkBlocks (thisRooms,y);
                        end
                    end
                    for y=n:NofRooms
                        if strcmp (thatNames{y}, 'empty')
                            thatNames{y}=checkBlockNames (thatNames,y);
                            thatRooms{y}=checkBlocks (thatRooms,y);
                        end
                    end
                end
                thisCell{1+Found}=checkedThisName;
                thisNames{m}='empty';
                thatNames{n}='empty';
                auxiCell{1+Found}=checkedThisRoom;
                thisRooms{m}=[];
                thatRooms{n}=[];
                thisNames{k}=thisCell;
                thisRooms{k}=auxiCell;
                thisP=cell (0,0);
                thisP{1}=thisNames;
                thisP{2}=thisRooms;
                index=sortedresult (pNum+i);
                thisPopulation{index}=thisP;
                if alikeRooms==NofRooms
                    DeleteBlocks=deleteBlocks+1;
                    thatP=cell (0,0);
                    thatNames=thisNames{1};
                    thatRooms=thisRooms{1};
                    thatP{1}=thatNames;
                    thatP{2}=thatRooms;
                    index=sortedresult (pNum+j);
                    thisPopulation{index}=thatP;
                else

```

```

        newBlocks=newBlocks+1;
        thatNames{1}=thisCell;
        thatRooms{1}=auxiCell;
        thatP=cell(0,0);
        thatP{1}=thatNames;
        thatP{2}=thatRooms;
        index=sortedresult(pNum+j);
        thisPopulation{index}=thatP;
    end
end
end
end
    if nFound==1
        break
    end
end
end
end
end
end
end
end
end
end
    if Found>0
        newBlocks=newBlocks+1;
        break,
    end
end
    if Found>0
        break,
    end
end
end
end
    if mFound==1
        break
    end
end
end
    %Making a block out of the best combinations in last population
    elite=0;
    if thisScore==NofRooms*70*ones(pNum,1)
    else
    bests=cell(0,0);
    for i=1:elite
        TheBest=bestPups{i};
        NofRooms=size(TheBest{1},2);
        bestNames=TheBest{1};
        bestRooms=TheBest{2};
        cellNames=cell(0,0);
        cellRooms=cell(0,0);
        for j=1:NofRooms
            Name=checkBlockNames(bestNames,j);
            Room=checkBlocks(bestRooms,j);

```

```

    cellNames{j}=Name;
    cellRooms{j}=Room;
end
BestFixed=cell(0,0);
BestFixedNames=cell(0,0);
BestFixedRooms=cell(0,0);
BestFixedNames{1}=cellNames;
BestFixedRooms{1}=cellRooms;
for k=2:NofRooms
    BestFixedNames{k}='empty';
    BestFixedRooms{k}=[];
end
BestFixed{1}=BestFixedNames;
BestFixed{2}=BestFixedRooms;
bests{i}=BestFixed;
end
for j=0:elite-1
    i=sortedresult(pNum+j+1);
    thisScore(i)=sortedresult(j+1);
    thisPopulation(i)=bests(j+1);
end
end
%checking if the common pattern includes the whole combination to be
%deleted
nKids = pNum/2;
xoverKids = cell(nKids,1);
index = 1;
for i=1:nKids
    parent1 = thisPopulation(parents(index));
    parent2 = thisPopulation(parents(index+1));
    argMom = parent1{1};
    argDad = parent2{1};
    index = index + 2;
    rooms=argMom{1};
    roomsCoord=argMom{2};
    dadRooms=argDad{1};
    dadRoomsCoord=argDad{2};
    s=size(rooms);
    numberOfRooms=s(2);
    Xmin=1;
    Xmax=numberOfRooms-1;
    random=floor(rand.*(Xmax-Xmin+1)+Xmin);
    Xmax=numberOfRooms;
    selectedRooms=cell(0,0);
    j=1;
    while j<=random
        cellFound=0;
        sz=size(rooms,2);
        for t=1:sz-1
            if iscell(rooms{t})
                cellFound=1;
            end
        end
        if cellFound
            selectedRooms{j}=rooms;
            j=j+1;
        end
    end
end

```

```

        blockSize=size(rooms{t},2);
        break;
    end
end
if cellFound==1
    selectedRooms=rooms{t};
    random=blockSize;
    j=random+1;
    combWithName=cell(0,0);
    combWithName{1}=selectedRooms;
    movedRooms=cell(0,0);
    [movedRooms{1}, xToMove, yToMove]=moveRoom(roomsCoord{t}{1},[0;0]);
    for w=2:blockSize
        movedRooms{w}=moveRoom2(roomsCoord{t}{w},[xToMove;yToMove]);
    end
    combWithName{2}=movedRooms;
else
    %This is because we want to maximize our benefit from blocks
    for u=1:sz-1
        if iscell(dadRooms{u})
            cellFound=1;
            blockSize=size(dadRooms{u},2);
            break;
        end
    end
    if cellFound==1
        selectedRooms=dadRooms{u};
        random=blockSize;
        j=random+1;
        combWithName=cell(0,0);
        combWithName{1}=selectedRooms;
        movedRooms=cell(0,0);
        [movedRooms{1},xToMove,yToMove]=
moveRoom(dadRoomsCoord{u}{1},[0;0]);
        for w=2:blockSize
            movedRooms{w}=moveRoom2(dadRoomsCoord{u}{w},[xToMove;yToMove]);
        end
        combWithName{2}=movedRooms;
        auxiliary=argMom;
        argMom=argDad;
        argDad=auxiliary;
    else
        rnd=floor(rand.*(Xmax-Xmin+1)+Xmin);
        selectedRooms{j}=checkBlockNames(rooms,rnd);
        rooms(rnd)=[];
        Xmax=Xmax-1;
        j=j+1;
    end
end
end
momRoomNames=argMom{1};

```

```

momRooms=argMom{2};
if cellFound==1
    existingBlocks=existingBlocks+1;
else
    combWithName=cell(0,0);
    for r=1:random
        selected=selectedRooms{r};
        for l=1:numberOfRooms
            momN=momRoomNames{l};
            momRN=momRooms{l};
            if strcmp(momN,selected),
                combination=putToGether(combWithName,momRN,momN);
                bestcomb=combination{2};
                combWithName=cell(0,0);
                combWithName{1}=combination{1};
                combWithName{2}=bestcomb;
                break,
            end
        end
    end
    dadRoomNames=argDad{1};
    dadRooms=argDad{2};
    for q=1:numberOfRooms
        exist=0;
        for t=1:random
            if strcmp(checkBlockNames(dadRoomNames,q),selectedRooms{t})
                exist=1;
                break,
            end
        end
        if exist==0,
            combination=putToGether(combWithName,checkBlocks(dadRooms,q),checkBlock
Names(dadRoomNames,q));
            bestcomb=combination{2};
            combWithName=cell(0,0);
            combWithName{1}=combination{1};
            combWithName{2}=bestcomb;
        end
    end
    child=combWithName;
    xoverKids{i} = child;
end

```

**Table 23- “CrossOverFcn” function**

Function Purpose	This is the cross over function in the room level
------------------	---

Application area	Room level
<pre> function xoverKids = CrossOverFcn(parents,options,NVARS, ...     FitnessFcn,thisScore,thisPopulation) nKids = length(parents)/2; xoverKids = cell(nKids,1); index = 1;  for i=1:nKids     parent1 = thisPopulation(parents(index));     parent2 = thisPopulation(parents(index+1));      argMom = parent1{1};     argDad = parent2{1};      index = index + 2;     random=rand();     if random&gt;0.5         argMom=upSideDown(argMom);     end;     random=rand();     if random&gt;0.5         argDad=upSideDown(argDad);     end;     random=rand();     if random&gt;0.5         argDad=leftSideRight(argMom);     end;     random=rand();     if random&gt;0.5         argDad=leftSideRight(argDad);     end;     random=rand();     if random&gt;0.5         child=add2right(argMom,argDad);     else         child=add2up(argMom,argDad);     end;     xoverKids{i} = child; end </pre>	

**Table 24- “DeleteRoom” function**

Function Purpose	This function deletes a room from the building layout
Application area	Building level
<pre>function newBlock=DeleteRoom(these,number)</pre>	

```

if iscell(these{number})
    block=these(number);
    s=size(block{1},2);
    b=0;
    emptyFound=0;
    while emptyFound<=s-2
        if strcmp(these(number+b+1),'empty')
            these(number+b+1)=[];
            emptyFound=emptyFound+1;
        else
            b=b+1;
        end
    end
    these(number)=[];
    newBlock=these;
elseif strcmp(these{number},'empty')
    a=1;
    eFound=0;
    blockFound=0;
    while blockFound==0;
        if iscell(these{number-a})
            blockFound=1;
            blockFound=1;
        else
            a=a+1;
        end
    end
    block=these{number-a};
    s=size(block,2);
    emptyFound=0;
    b=1;
    while emptyFound<=s-2
        if strcmp(these(number-a+b),'empty')
            these(number-a+b)=[];
            emptyFound=emptyFound+1;
        else
            b=b+1;
        end
    end
    these(number-a)=[];
    newBlock=these;
else
    these(number)=[];
    newBlock=these;
end

```

**Table 25- “FirstPopulation” function**

<b>Function Purpose</b>	This function creates an initial populations of building layouts
<b>Application area</b>	Building level
<pre> function population = FirstPopulation (NVARS, FitnessFcn, options, numberOfRooms, AllTheRooms, RoomNames)  combinedRooms=cell(0,0); for k=1:6     combination=cell(0,0);     combWithName=cell(0,0);     for l=1:numberOfRooms         thisRoom=AllTheRooms{1};         Rooms=thisRoom{1};         combination=putTogether(combWithName, Rooms{k}, thisRoom{2});         bestcomb=combination{2};         combWithName=cell(0,0);         combWithName{1}=combination{1};         combWithName{2}=bestcomb;     end     combinedRooms{k}=combWithName; end  totalPopulationSize = sum(options.PopulationSize); population = cell(totalPopulationSize,1); j=1;  for i=1:totalPopulationSize     if j&lt;=6         population{i}=combinedRooms{j};         j=j+1;     else         j=j-6;         population{i}=combinedRooms{j};         j=j+1;     end end end </pre>	

**Table 26- “isContinuous” function**

<b>Function Purpose</b>	This Boolean function checks if a given building layout is continuous
<b>Application area</b>	Building level

```

function connected=isContinious(GroupedRooms)
queue=GroupedRooms(:,1);
remainedN=GroupedRooms;
remainedN(:,1)=[];

stack=[];
while size(queue,2)>0
    [queue,thisNode]=pop(queue);
    stack=[stack,thisNode];
    x=thisNode(1);
    y=thisNode(2);
    y1=y+1;
    y2=y-1;
    x1=x+1;
    x2=x-1;
    i=0;
    while i<=size(remainedN,2)-1
        if remainedN(i*2+1)==x
            if remainedN(i*2+2)==y1 | remainedN(i*2+2)==y2
                queue=push(queue,remainedN(:,i+1));
                remainedN(:,i+1)=[];
                i=i-1;
            end
        elseif remainedN(i*2+2)==y
            if remainedN(i*2+1)==x1 | remainedN(i*2+1)==x2
                queue=push(queue,remainedN(:,i+1));
                remainedN(:,i+1)=[];
                i=i-1;
            end
        end
        i=i+1;
    end
end

if size(stack,2)==size(GroupedRooms,2)
    connected=1;
else
    connected=0;
end

function [newQueue,poped]=pop(argQueue)
s=size(argQueue,2);
poped=argQueue(:,s);
argQueue(:,s)=[];
newQueue=argQueue;

function newQueue=push(argQueue,data)
newQueue=[argQueue,data];

```

**Table 27- "isOutSide" function**

<b>Function Purpose</b>	This Boolean function checks if a given room is an exterior room in the given layout
<b>Application area</b>	Building level
<pre> function TorF=isOutSide(Room, housePerim) TorF=0; shouldBreak=0; roomPerim=buildPerimeter(Room); for k=1:size(roomPerim,2)-1     if roomPerim(:,k)==housePerim(:,1)         n=size(housePerim,2);         if roomPerim(:,k+1)==housePerim(:,2)   roomPerim(:,k+1)==housePerim(:,n-1);             TorF=1;             shouldBreak=1;             break;         end     end end for l=2:size(housePerim,2)-1     if roomPerim(:,k)==housePerim(:,l)         if roomPerim(:,k+1)==housePerim(:,l+1)   roomPerim(:,k+1)==housePerim(:,l-1)             TorF=1;             shouldBreak=1;             break;         end     end end end if shouldBreak     break end end end </pre>	

**Table 28- "leftSideRight" function**

<b>Function Purpose</b>	This function mirrors a given population
<b>Application area</b>	Room level
<pre> function newBlock=leftSideRight(argBlock) y=argBlock(1,:); maxWidth=max(y); width=argBlock(1,:); heights=argBlock(2,:); </pre>	

```
width=maxWidth-width;
newBlock=[width;heights];
```

**Table 29- “minimizingArea” function**

Function Purpose	This function evaluates a building layout based on its area
Application area	Building level
<pre>function fitnessScore=minimizingArea(argGroupedRooms) cellSize=size(argGroupedRooms); numberOfRooms=cellSize(2); GroupedRooms=[]; for i=1:numberOfRooms     GroupedRooms=cat(2,GroupedRooms,argGroupedRooms{i});%creating a matrix that contains all of the rooms continually end arrangedGroupedRooms=arrangedBlock(GroupedRooms); s=size(arrangedGroupedRooms); area=s(2); coveredArea=0; for i=1:area     for j=1:numberOfRooms         room=argGroupedRooms{j};         rS=size(room);         rArea=rS(2);         for k=1:rArea             if arrangedGroupedRooms(:,i)==argGroupedRooms{j}(:,k),                 coveredArea=coveredArea+1;             end         end     end end end fitnessScore=s(2)-coveredArea;</pre>	

**Table 30- “moveRoom” function**

Function Purpose	This function puts the first corner of a given room to a given point and determines the movement vector
Application area	Building level
<pre>function [movedRoom, xToMove, yToMove]=moveRoom(argRoom, pointToMove) builtRoom=buildBlock4(argRoom); firstCorner=argRoom(:,1); movedRoom=shiftRoom(argRoom, builtRoom, firstCorner, pointToMove);</pre>	

```

firstPoint=movedRoom(:,1);
xToMove=firstPoint(1)-firstCorner(1);
yToMove=firstPoint(2)-firstCorner(2);

function movedRoom = moveRoom2(argRoom, distanceToMove)
s=size(argRoom,2);
for i=1:s
    movedRoom(:,i)=argRoom(:,i)+distanceToMove;
end

```

**Table 31- "mutate\_permutation3"function**

<b>Function Purpose</b>	This function takes one of the exterior rooms in the layout and puts it back in a new position
<b>Application area</b>	Building level
<pre> function mutationChildren = mutate_permutation3(parents ,options,NVARS, ...     FitnessFcn, state, thisScore,thisPopulation,mutationRate) index=1; n=length(parents); mutationChildren=cell(n,1); for i=1:n     parent=thisPopulation(parents(index));     index=index+1;     child=parent{1};     Names=child{1};     Rooms=child{2};     s=size(Names,2);     random=floor(rand.*(s)+1);     for j=random:s         RoomsP=Rooms;         NamesP=Names;         Room=turnBlock(Rooms{j});         RoomsP(j)=[];         Name=Names{j};         NamesP(j)=[];         GroupedRooms=[];         for k=1:size(RoomsP,2)             %creating a matrix that contains all of the rooms continually             GroupedRooms=cat(2,GroupedRooms,RoomsP{k});         end         house=arrangedBlock3(GroupedRooms);         if isContinious(GroupedRooms) j==s             combWithName=cell(0,0);             combWithName{1}=NamesP;             combWithName{2}=RoomsP;             child=putTogether(combWithName,Room,Name);             break, </pre>	

```

        end
    end
    mutationChildren{i}=child;
end;

```

**Table 32-“plotBlock” function**

<b>Function Purpose</b>	This function sets the plot settings to an appropriate format for printing the rooms in an architectural layout
<b>Application area</b>	Both room and building levels
<pre> function plotBlock(argFigure) plot(argFigure(1,:),argFigure(2:),'r'); axis equal; </pre>	

**Table 33- “plotCombinations” function**

<b>Function Purpose</b>	This function plots all the building layouts in a population
<b>Application area</b>	Building level
<pre> function state =plotCombinations(options,state,flag) totalPopulationSize = sum(options.PopulationSize); for i = 1:24     subplot(6,6,i+12);     genotype = state.Population{i};     names=genotype{1};     rooms=genotype{2};     s=size(rooms);     for j=1:s(2)         thisRoom=rooms{j};         buildBlock(thisRoom);         hold on;     end     hold off; end </pre>	

**Table 34- “plotRooms” function**

<b>Function Purpose</b>	This function plots all rooms in a population.
<b>Application area</b>	Room level

```

function state =plotRooms (options, state, flag)
for i=1:16
    subplot(4,8,i+16);
    genotype = state.Population{i};
    buildBlock(genotype);
end

```

**Table 35- “proportionFitness” function**

<b>Function Purpose</b>	This function determines the proportions of a given room in a population
<b>Application area</b>	Room level
<pre> function value=proportionFitness(argBlock) width=max(argBlock(1,:))+1; length=max(argBlock(2,:))+1; if width==0     value=50; % This would never happen! elseif length==0     value=50; else     value=length/width; end </pre>	

**Table 36- “putTogether” function**

<b>Function Purpose</b>	This function puts all of the given rooms together without any overlap and in a continuous arrangement.
<b>Application area</b>	Building level
<pre> function newGroupedRoomsCell=putTogether(argGroupedRooms, argRoom, RoomName) %argGroupedRooms should be a 1 by N cell containing rooms and arg room should be a matrix %newGroupedRoomsCell is a M by N cell that contains all of the possible combinations of argGroupedRooms with agrRoom S=size(argGroupedRooms); N=S(2); if N==0,     newGroupedRoomsCell=cell(0,0);     name=cell(0,0);     name{1}=RoomName;     newGroupedRoomsCell{1}=name; </pre>	

```

room=cell(0,0);
room{1}=argRoom;
newGroupedRoomsCell{2}=room;
else
Rooms=argGroupedRooms{2};
cellSize=size(Rooms);
numberOfRooms=cellSize(2);
GroupedRooms=[];
for i=1:numberOfRooms
    GroupedRooms=cat(2,GroupedRooms,Rooms{i});%creating a matix
that contains all of the rooms continually
end
builtGroupedRooms=buildBlock4(GroupedRooms);%figuring out the
borders of the rooms together
realizableCombs=0;
placedRoom=cell(0,0);
s=size(builtGroupedRooms);
nOfCorners=s(2);
%Finding combinations of rooms that don't have overlaps
SR=size(argRoom);
RoomArea=SR(2);
fixedRooms=arrangedBlock3(GroupedRooms); %might be extra
SG=size(fixedRooms);
GroupArea=SG(2);
builtRoom=buildBlock4(argRoom);
BRS=size(builtRoom);
corners=BRS(2);
l=floor(rand.*(4-1+1)+1);%selecting a random number between 1 and 4
if l==1,
    CurrentRoom=argRoom;
elseif l==2,
    argRoom2=leftSideRight(argRoom);
    CurrentRoom=argRoom2;
elseif l==3,
    argRoom3=upSideDown(argRoom);
    CurrentRoom=argRoom3;
elseif l==4,
    argRoom2=leftSideRight(argRoom);
    argRoom4=upSideDown(argRoom2);
    CurrentRoom=argRoom4;
end
builtRoom=buildBlock4(CurrentRoom);
found=0;
while found==0
    i=floor(rand.*(nOfCorners-1+1)+1);%selecting one of the corners
cornerToPut =builtGroupedRooms(:,i);
m=floor(rand.*(corners-1+1)+1);%selecting one of the corners
cornerOfRoom=builtRoom(:,m);
[newroom1, builtNewRoom1]=
shiftRoom(CurrentRoom,builtRoom,cornerOfRoom,cornerToPut);
overlap=0;

```

```

    for j=1:RoomArea
        for k=1:GroupArea
            if newroom1(:,j)==fixedRooms(:,k)
                overlap=1;
                break,
            end
            if overlap==1
                break,
            end
        end
    end
    if overlap==0,
        %checking if the rooms have common perimeter
        room1Perimeter=buildPerimeter(newroom1);
        fixedRoomPerim=buildPerimeter(fixedRooms);
        commonPerim=0;
        for k=1:size(room1Perimeter,2)-1
            thisSide=room1Perimeter(:,k:k+1);
            thisSide2=room1Perimeter(:,k+1:-1:k);
            for m=1:size(fixedRoomPerim,2)-1
                thatSide=fixedRoomPerim(:,m:m+1);
                if thisSide==thatSide | thisSide2==thatSide
                    commonPerim=1;
                    break;
                end
            end
        end
        if commonPerim==1
            found=1;
        end
    end
    end
    newGroupedRoomsCell=cell(0,0);
    names=cell(0,0);
    oldNames=argGroupedRooms{1};
    s=size(oldNames);
    nameNumbers=s(2);
    for j=1:nameNumbers
        names{j}=oldNames{j};
    end
    names{j+1}=RoomName;
    newGroupedRoomsCell{1}=names;
    GroupedRoomsCell=cat(2,argGroupedRooms{2},newroom1);
    newGroupedRoomsCell{2}=GroupedRoomsCell;
end

```

**Table 37- “SelectingTheBest” function**

Function Purpose	This function selects the populations among the building layouts.
------------------	---

Application area	Building level
<pre> function bestCombs=SelectingTheBest(GroupedRoomsCell) cellSize=size(GroupedRoomsCell); numberOfCombs=cellSize(2); bestScore=SmoothPerimeter(GroupedRoomsCell{1})+1; bestCombs=cell(0,0); for i=1:numberOfCombs     fitnessScore=SmoothPerimeter(GroupedRoomsCell{i});     if fitnessScore&lt;bestScore,         bestScore=fitnessScore;         bestCombs=GroupedRoomsCell{i};     end end end </pre>	

Table 38-“shiftRoom” function

Function Purpose	This function takes a room and a coordinate of a point and moves the room to the taken coordinates.
Application area	Building level
<pre> function [newRoom,builtRoom]=shiftRoom(argRoom, argbuiltRoom, cornerOfRoom, cornerT oPut) s=size(argbuiltRoom); roomCorners=s(2); xRoom=cornerOfRoom(1); yRoom=cornerOfRoom(2); xToPut=cornerToPut(1); yToPut=cornerToPut(2); xDifference=xToPut-xRoom; yDifference=yToPut-yRoom; builtRoom=argbuiltRoom;  for i=0:roomCorners-1     builtRoom(i*2+1)=argbuiltRoom(i*2+1)+xDifference;     builtRoom(i*2+2)=argbuiltRoom(i*2+2)+yDifference; end  s=size(argRoom); area=s(2); newRoom=argRoom;  for i=0:area-1     newRoom(i*2+1)=argRoom(i*2+1)+xDifference;     newRoom(i*2+2)=argRoom(i*2+2)+yDifference; end end </pre>	

**Table 39-“smoothnessFitness” function**

<b>Function Purpose</b>	This function evaluates a room based on its degree of compactness.
<b>Application area</b>	Room level
<pre>function score = smoothnessFitness(x) room=buildBlock(x); s=size(room); sides=s(2); score=sides-4;</pre>	

**Table 40-“SmoothPerimeter” function**

<b>Function Purpose</b>	This function evaluates a building layout based on its degree of compactness.
<b>Application area</b>	building level
<pre>function fitnessScore=SmoothPerimeter(argGroupedRooms) cellSize=size(argGroupedRooms); numberOfRooms=cellSize(2); GroupedRooms=[]; for i=1:numberOfRooms     GroupedRooms=cat(2,GroupedRooms,argGroupedRooms{i});%creating a     matix that contains all of the rooms continually end builtGroupedRooms=buildBlock5(GroupedRooms);%figuring out the borders of the rooms together s=size(builtGroupedRooms); nOfCorners=s(2); fitnessScore=nOfCorners-4;</pre>	

**Table 41- “turnBlock” function**

<b>Function Purpose</b>	This function turns a given room 90 degree counterclockwise
<b>Application area</b>	Room level
<pre>function newBlock=turnBlock(argBlock) ies=argBlock(1,:); jes=argBlock(2,:); newBlock=[jes;ies];</pre>	

**Table 42- “upSideDown” function**

<b>Function Purpose</b>	This function makes a given room upside down
<b>Application area</b>	Room level
<pre>function newBlock=upSideDown(argBlock) y=argBlock(2,:); maxHeight=max(y); width=argBlock(1,:); heights=argBlock(2,:); heights=maxHeight-heights; newBlock=[width;heights];</pre>	