FCTUC DEPARTAMENTO DE ENGENHARIA CIVIL
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Development of an algorithm which generates functional floor plans for residential buildings based on modular construction

Dissertation submitted for the degree of Master of Civil Engineering in Structural Mechanics

Author

João Francisco Rodrigues Costa

Supervisors

Prof. Doc. Luís Alberto P. Simões da Silva (FCTUC)

Prof. Doc. Vítor Manuel Bairrada Murtinho (FCTUC)

Coimbra, October, 2015

AGRADECIMENTOS

Gostaria de começar não sem antes deixar o meu mais profundo agradecimento a todos os que de uma forma ou de outra contribuíram para o desenvolvimento desta dissertação, e que me foram deixando pedaços de informação pelo chão até à saída do labirinto: ao André Perdigão, ao João Francisco Sousa, ao Dinis Sarmento, ao André Carvalho, ao Guilherme Pires, ao Diogo Resende, à Catarina Agreira e ao André Tereso. Aos meus orientadores, Professor Doutor Luís Simões da Silva e Professor Doutor Vítor Murtinho, por todo o apoio e tempo disponibilizado. Ao Professor Doutor Paulo Providência, por toda a gentileza com que sempre me tratou. Por fim mas não menos importante, à Arquiteta Tânia Martins, que disponibilizou a sua dissertação de contributo valioso a este trabalho.

Aos meus amigos, e em especial aos colegas de curso, que me acompanharam durante estes longos cinco anos, quer em trabalho quer em farra, em particular ao Daniel Oliveira, ao Francisco Arede, ao Gonçalo Gomes, ao Manuel Porém e ao João Almeida. Ao Filipe Firmo e ao Gonçalo Duarte, por todo o apoio disponibilizado desde o primeiro dia de aulas, e onde recorri sempre que precisei. Por fim à Ângela Lemos. Não consigo precisar em quanto ela me ajudou durante estes anos, quer fosse em apontamentos, conversas, corridas ou explicações.

Aos meus Pais, à minha Irmã, e aos meus Avós, em especial à minha Avó São. Por todas as oportunidades, todo o amor, e todo o apoio. Nunca seria o que sou hoje sem vocês, e sei que posso contar com o vosso apoio. O meu mais profundo agradecimento por tudo. Porém, peço que não se esqueçam,

"Whatever it is you're seeking, won't come in the form you're expecting."
Haruki Murakami

Ao João Portugal.
Ao meu Avô Álvaro.
Saudades.

RESUMO

Um dos objetivos fundamentais da arquitetura e da engenharia civil é a construção e reabilitação de edifícios habitacionais. Estas áreas colaboram entre si para desenvolver soluções que satisfaçam os desejos do cliente incluindo as preferências estéticas e os requisitos técnicos legais, e, obviamente, tendo em conta a segurança, funcionalidade e o conforto da construção. Isto poderá ser atingido através dum processo iterativo que usualmente começa por um leque pequeno de soluções, das quais uma é escolhida e seguidamente modificada até se obter o projeto final da habitação. Este procedimento demora habitualmente um tempo substancial.

Por outro lado, a construção modular constitui um método inovador baseado num conceito de casa adaptável e flexível, que consiste na organização de soluções uniformes em bloco, pré-fabricadas em aço enformado a frio, numa grelha organizacional de múltiplos de uma unidade básica, quase como se de um jogo de Lego® se tratasse. Esta tecnologia é altamente sustentável e permite modificações na construção final durante o seu período de vida útil. Por fim, os custos são minorados e a relação custo-benefício é majorada.

Assim, a presente dissertação objetiva o desenvolvimento dum algoritmo que gere várias soluções de plantas de arquitetura baseado em módulos arquitetónicos de construção modular previamente estudados. Este objetivo foi atingido através da implementação em Java® deste algoritmo que testa e combina 3 módulos de arquitetura modular até encontrar todas as soluções que obedeçam a determinadas restrições.

Palavras-chave: Adaptabilidade; Alocação Espacial; Arquitetura Modular; Construção Metálica; Flexibilidade; Geração de Plantas de Arquitetura; Habitações Económicas; Planeamento Espacial Automático.

ABSTRACT

One of the fundamental objectives of architecture and civil engineering is to construct and rehabilitate residential buildings. Both parties collaborate in order to provide solutions that fit the client's desires, satisfy legal requirements, while granting aesthetic preferences and, obviously, are functional, safe and satisfy serviceability requirements. This may be accomplished by an iterative process which commonly starts from a spectrum of a few solutions, where one is chosen and arranged until achieving the final building design. This process takes a substantial amount of time.

On the other hand, modular construction presents an innovative constructive method based on a flexible and adaptive housing concept, which consists on arranging uniform modular solutions, pre-fabricated on cold formed steel, on an organizational grid of multiple numbers of a basic unit, almost as if it was a Lego® game. This technology is highly sustainable and allows modifications on the solutions during its life cycle. Last but not least, costs are diminished and energy efficiency is improved.

Therefore the objective of the present dissertation is the development of an algorithm that generates multiple floor plan design solutions based on previously studied architectural modules of modular construction. This objective was accomplished by implementing the algorithm in Java® which tests and combines 3 modules of modular construction until it reaches every solutions that obey specific constraints.

Keywords: Adaptability; Affordable Habitations; Automatic Space Planning; Flexibility; Floor Plan Generation; Modular Architecture; Space Allocation; Steel Construction.

TABLE OF CONTENTS

ABREVIATIONS

CAD – Computer Aided-Design;
EPSAP – Evolutionary Program for the Space Allocation Problem;
ES – Evolutionary Strategy;
GA – Genetic Algorithm;
GP – Genetic Programming;
MIT – Massachusetts Institute of Technology;
QAP – Quadratic Assignment Problem;
RGEU – Regulamento Geral das Edificações Urbanas;
SHC – Stochastic Hill Climbing.

1. INTRODUCTION

Sustainability. *"Development which meets the needs of current generations without compromising the ability of future generations to meet their own needs."*
World Commission on Environment - *Brundtland Report* (1987)

1.1. General considerations

The very first step of a construction process begins with a primordial discussion with the client, during which the designer gathers data, such as clients' requirements, some personal living habits, aesthetic preferences, topographical constraints and other relevant information. This step is mostly known as *space planning phase*, and consists on the sketching of the future building where the professional not only considers the constraints mentioned above, but also the ruling legal requirements, producing a few alternative floor plans that will ideally fulfill every constraint and satisfy every participant of the process (Neufert, 2010).

This is a considerably complex trial and error process, time consuming, and highly dependent on the professional's past experience and style (Rodrigues, 2014). Moreover, it is commonly an iterative process, since professionals will start mainly with one solution and perform several arrangements until they reach what they idealize to be the optimum floor plan, or the final building design, which should meet the client's needs and desires. This leads to the discard of potentially better design solutions.

On the other hand, this dissertation takes in mind a new flexible and adaptable housing concept, where a new house may be based on industrialized modules of cold formed steel on an organizational square grid where each unit has fixed dimensions. The modules considered during this dissertation have been previously studied by Martins (2011), who organized them mainly by their function. An algorithm which combined them in order to facilitate both the architect and the inhabitant, to idealize and construct an affordable, flexible and adaptive residence in the shortest period of time possible, would be undoubtedly advantageous.

Modular construction, an obvious result of sustainable conception, will surely play an important role in the future of construction. Several reasons for this may be pointed out, such as: its standardized dimensions, which not only eliminates usual construction errors but also speeds up construction processes (modules may even be assembled in factory and transported to the construction site in 2D or 3D pieces); or its ecologic sustainability (Lawson, 2014). Buildings

and structures are currently erected to last for 50 or 100 years. Which leads to the question: what should we do with them after this period?

Sustainable development answers this question by implementing a sustainable construction cycle which considers extraction of stocks, planning, projecting, building, demolition and management of any resulting solid residues that, until recently, could not be reutilized. Harmony between natural and constructed environment must be strengthened, and at the same time, human clusters should be created to reinforce human dignity and to encourage economical equity (du Plessis, 2002).

Some primordial aspects that affect sustainability on the construction sector should be borne in mind and can be improved, such as: sustainable materials – around 50% of extracted materials from earth are to be utilized on construction; buildings' energetic efficiency – around 40% of energy production is spent on construction, operation and buildings' demolition – it also contributes with a similar percentage to greenhouse gases; management of construction and dismantling – residues from construction and demolition are the biggest source of waste in terms of weight in the European Union (Gervásio et al.,2015). This situation must be modified by decreasing the consumption of raw materials and energy and by promoting recycling of residues.

Therefore, this dissertation aims at the development of an algorithm, not only to contribute with a step towards a real time floor plan generation, design and optimization, but also to trigger another step into sustainable and modular construction. The developed solution uses, as a case study, Martins' (2011) proposed solutions of 6x6m modules, and provides an answer for increasing the range of initial design options, diminishes high design prices, decreases conception time consumption, and foments material recycling, while also encouraging sustainability.

1.2. Goals of the dissertation

The present dissertation aims to develop and implement an algorithm that generates floor plan design solutions for modular construction. This algorithm has two objectives in mind: first, to accelerate and partially "automate" the design process, and second, to push modular construction implementation and all the benefits associated with it, as described above. As innovative as it may be, this work aims to take a step further into the future of automatic generation of floor plans based on modular construction.

1.3. Dissertation outline

The dissertation has the following structure:

Chapter 1 presents a brief introduction on the main topic of the present dissertation, briefly describing the reasons and the advantages of the developed algorithm. In addition, it outlines the goals of the essay, and provides a synopsis for each chapter.

Chapter 2 provides a contemporary look on the subject, including previous works, both in architectural regulation, and on floor plan generation algorithms and its mathematical methodologies.

Chapter 3 presents the architectural constraints required, and their transformation into mathematical restrictions. It also describes the implemented inquiry, as well as the program's interface.

Chapter 4 presents the developed algorithm and its implementation in Java®, step by step, for easier comprehension, in case the reader is a layman concerning the programming world. The algorithm contains approximately 2300 lines of code. The most important utilized processes are also described in this section.

Chapter 5 presents the final conclusions from the developed work, and offers the author's thoughts about possible future works and developments on the theme.

2. BACKGROUND

It is usual for professionals to begin a building design by a handmade process. There is currently no available software that offers the designer a first-hand perspective of the building; one which allows him to choose and develop one design solution from a wide range of initial possibilities. Yet, there has been some research developed around the subject which may prove automated generation of floor plans to be fruitful.

Furthermore, modular construction has been growing for the past years simultaneously with the sustainable development sector, offering multiple advantages that should be taken into account, such as environmental friendly materials (that may be reutilized), lower prices, faster construction, or the possibility of remote construction and consequent transportation, among others. Let us take the medical field as an example: it would be extremely advantageous to be an adopter of large-scale modular construction, since the standardization of room environments and of the building itself would greatly improve employees' efficiency and patients' welfare. However that is not the only reason to bear in mind; a modular approach of any building provides benefits not only for the project and construction stage, but also during its lifetime maintenance. With a detailed project, it is possible to know what lies within the walls and confidently solve inefficacies that naturally occur (Ball, 2014). Moreover, a modular approach may even take advantage of space individualization by providing solutions that are mass produced but are different on small details, such as window locations and sizes, room positions or even different floor or wall coverings, leading to extremely similar rooms and/or habitations on a constructive perspective, but unique on an aesthetic perspective: take Siza's Malagueira Houses as an example (Duarte, 2001). To sum up, modular construction companies can currently work with high levels of design and construction sophistication which are definite competition for conventional construction companies. All its benefits are telling.

This chapter explores these subjects and related works.

2.1. Introduction

Computer Aided-Design (CAD) systems – "*the use of computer systems to assist in the creation, modification, analysis, or optimization of a design"* (Groover, 1983) – appeared around the late 1960's, and came up as a response to the need of production and automation of accurate drawings at a real scale (Narayan, 2008). Fairly straightforward small buildings could take two days to be drawn with paper and a slide rule – obviously, engineering knowledge was put aside and was replaced by arithmetic skills. Consequently, with the CAD industry

development, design processes became faster, more precise, definitely cheaper, and automated in a certain way, since even labour force was reduced (Weidsberg, 2008).

Since then, several programs have been developed for the automated production of spatial layout problems. In order to fully incorporate artificial intelligence into the architectural field, these problems need to be further analysed. Objectives and scope of programs have varied widely and were directed to multiple scientific areas, one of these being floor planning generation (Ligget, 2000). Yet, this type of problem is known for its complexity, since there is a massive number of constraints to be overcome. Also, professionals' styles and preferences are hard to define mathematically, and it should not be forgotten that the majority of space allocation problems will hardly overcome every requirement in the future.

Several methods for solving spatial layout problems started to be developed around the 1980's. In order to create and evaluate alternative layout variations these methods use a generative and an evaluation mechanism respectively (Mitchell, 1998). These methods can be divided into three categories that are relevant to layout systems: procedural, heuristic, and evolutionary (Kalay, 2004). According to Galle, in the early 80s, the first algorithms for the exhaustive generation of building floor plans started being developed (Galle, 1981). A decade later, SEED was born, a new system that would generate layouts through constraint programming (Flemming & Woodbury, 1995), a hierarchical extension of a system named LOOS (Flemming, 1989), which uses orthogonal structures for the representation of loosely packed arrangements of rectangles. Gero was one of the first to make use of evolutionary approaches for solving spatial layout problems in architecture (Gero & Kazakov, 1996; Schnier & Gero, 1996; Jo & Gero, 1998; Rosenman & Gero, 1999). During these years various methods began to appear, such as planar edge-connected squares, rectangle dissection, proxemics dimensions, packing rectangles, graph theory, and space partitioning (Rodrigues, 2014). The main purpose of most of these programs was the exhaustive generation of floor plan solutions. Soon it became obvious that this approach would fail against medium and large problems, since space allocation is a combinatorial problem on its basis, which not only produces an unbearable amount of solutions, but also demands for high computation and memory capacity - this fact may be solved by increasing restrictions and requirements, and also by computer evolution, since computation and memory capacity are continuously increasing.

Therefore, researchers progressed from the idea of enumerating all possible solutions onto the *evolutionary methods'* path. This type of methods are known for solving both constrained and unconstrained optimization problems based on Charles Darwin theory of evolution by natural selection – *"Survival of the fittest".* Thus, starting from an initial set of individuals (floor plan solutions, represented by chromosomes and called population when in a group), "*evolutionary operators select, recombine and mutate the genetic material to produce the offspring. Solutions*

which will mutate into new ones are selected according to their fitness - the more suitable they are the more chances they have to reproduce. The procedure is repeated until some condition is satisfied" (Rodrigues, 2014). This type of algorithm will be discussed is section 2.3.1.

The reviewed approaches seemed to be too specific or too abstract. For instance, some only tackled topologically aspects (Wong and Chan, 2009), others only geometrically ones, and others even tried to imitate some architectural design styles (Jackson, 2002; Serag et al., 2008). The use of basic crossover among different alternative methods may result on a less diverse set of floor plans from a topological and geometric perspective. Hence, in order to maintain the chromosome's coherence, a judicious use of crossover must be carried out as well (Rodrigues, 2014), to avoid problems such as duplication of spaces, or the lack of these (Flack, 2011). Variables also vary according to each approach, and may be exterior and interior doors and windows, wall thicknesses and dimensions, orientation, floor levels, building boundaries, adjacent buildings, among others. Topological features of crucial relevance also disparate between approaches, and they can be the openings' orientations, or adjacencies between spaces, for example. Few approaches dealt with both openings and floor levels. Mostly, research works treat each space as an orthogonal space, which is advantageous to our final objective.

It is important to mention that the proposed algorithm, which would ideally evolve into a user-friendly program for the professionals to use on a first-hand approach of a building design, may be object of rejection by them, since they would possibly argue its conflict with their creative process. Nonetheless, creativity is not intended to be put aside but to be stimulated by substantially increasing the range of initial floor plan solutions.

It is crucial to point out that the algorithm is intended for the connection of pre-fabricated modular spaces, based on typologies studied on previous dissertations of the University of Beira Interior in collaboration with the University of Coimbra.

2.2. Space allocation – Modular construction; Requirements and constraints

"Architecture is a complex amalgamation of science and art" (Flack, 2011). There is a countless number of normative requirements to follow and to add to clients' and professionals' personal aesthetics preferences, which will lead to a vast range of solutions. Nonetheless, professionals are most likely to begin with one solution and iterate it, until the result becomes pleasant for both parties. No solution will surely suit every client. Thus, a large amount of initial alternatives would be of major relevance to assist with a computer algorithm.

On the other hand, the housing concept based on modular construction consists on a structure comparable with a Tetris® game: an agglomeration of pre-fabricated/pre-modelled forms, not

only on the same floor level but also on upper levels (multi-layer buildings). Therefore, it consists on a perfectly flexible and resilient solution, able to meet the needs of mobility and adaptation, whenever there is a necessity of increasing or decreasing the household. Modular construction companies provide a set of elements of multiple shapes, which can be arranged in the way both the designer and the client desire the most, enhancing the formation of multiple typologies and their combination. This solution also allows the adaptation to existing facilities. The modules main advantages are their customization and optimized dimensions, which enable the articulation between elements, the use of a constructive system where the elements are able to be disaggregated, enabling future addition or subtraction of modules, or even their demolition, taking advantage of the elements to be recycled and, last but not least, its substantially lower budget than traditional solutions (Cool Haven@, 2015). Figure 2.1 presents a modular habitation example by Cool Haven.

Figure 2.1 – Coolhaven's example of a modular habitation (Cool Haven@, 2015).

Tânia Martins, a former architecture student from the Faculty of Civil Engineering and Architecture of the University of Beira Interior, performed a research on modular typologies on an organizational grid of measures multiple of 0.6m, in cooperation with the University of Coimbra. This dimension is an estimate of the most favourable structural rhythm, i.e., which

optimizes construction, and from which resulted Martins' customized modules of 6x6m – advantageous due to their square shape and their larger stiffness and structural strength, enabling their future adaptation to other forms in different ways. Martins grouped those typologies according to their context and interior arrangement, taking into account some crucial factors, such as circulation minimization, space optimization, legal requirements, variety of solution for each typology and respective function, and even its free interior space (Martins, 2011).

Each module has a total area of $36m^2$ and $29m^2$ of usable space, approximately – with 0.3 and 0.1m thickness for exterior and interior walls, respectively. There are three kinds of modules, A, B and C. Generally, A refers to kitchens with stair cases allocated, with the possibility of adding a pantry, a machine room or even a small service toilet, not forgetting that corridors may vary its size; B refers mainly to rooms which may have variations such as different room sizes, a single large room with private bathroom, the possibility of adding wardrobes, and even a small exterior space, with a small garden or even a balcony; finally, C modules refer to living rooms, one of the house's larger spaces, which has a bathroom included that may be just a service toilet or a bathroom that fulfills accessibility requirements. Variations on the last one would be the addition of a small office or an exterior space, or even changing the living room area to a L-shape (Martins, 2011).

Bearing in mind Martins' conclusions (2011), one must mention the following details:

- Toilets must always be placed alongside with exterior walls;
- Living rooms must be placed near to, at least, one service toilet;
- In the case of a two-floor building, the lower floor must not exceed 4 modules, and the upper floor 3 modules.

Also, RGEU imposes minimum and maximum space areas, minimum floor height, stair steps sizes, areas' positioning, ventilation, among others (RGEU, 2009).

This dissertation aims to turn Martins' idea of adaptability and flexibility that modular construction may offer into reality by developing an algorithm and implementing a user-friendly program for the generation of floor plan alternatives based on modular construction. "*One habitation may have several possible organizations"* (Martins, 2011). Martins' conclusions, RGEU and other regulations altogether should be translated into mathematical restrictions and inserted into the algorithm. All these restrictions are fully described in section 3.2. After a preliminary inquiry from the algorithm, it will generate a number of floor plan alternatives depending on user's selection choices.

Creativity, a much valued argument by architects, will not be put aside, but instead promoted as it offers a broader range of initial floor plan solutions for the professional to choose. This happens as the algorithm will not decide which alternative should be chosen.

2.3. Mathematical methodologies

As mentioned in the introduction above, a large amount of algorithmic search methods have been used for the automatic generation of floor plans. Rodrigues has aggregated them into 6 groups according to the type and purpose of each approach, which are presented below (Rodrigues, 2014)[1]:

Area assignment approach – The problem is treated as a *Quadratic Assignment Problem* in which *"a number of department unit areas must be assigned to an equal or larger building area."* This approach consists in assigning a minimum cost for allocating objects to locations. The costs are calculated as the sum of the product between distance and flow. Therefore, instead of dealing with spaces – house rooms such as the kitchen or bedrooms –, the objective is *"to assign unit areas of each department to a building floor area"*. Therefore, this approach abdicates *"individual spaces, circulation or openings, which are decided at a future stage of the design, or are implied constraints"*. The methods used were *Genetic Algorithms* and *Genetic Programming.*

Space allocation approach – Allocation of *"spaces with a defined shape according to their topological relations and geometric restrictions"*. In order to determine the best topological arrangement of the spatial layouts and to resize the floor plan according to particular constraints, the used approach was to analyse relations between these spaces through graph theory.

Hierarchical construction approach – One may treat a floor plan as a hierarchical system of elements. Considering a discrete unit of space as the most basic element, one may create a room by joining several units, several rooms to create a zone, and finally regarding several zones as a floor plan. In order to evaluate the relation of elements, different performance assessment criteria may be used depending on the hierarchy levels.

Conceptual exploration approach – The main objective of this approach is to foster creativity in cases where requirements or constraints are not a concern for the designer. As such, professionals are encouraged to generate novel shapes and configurations using a set of solutions for conceptual exploration of ideas, not necessarily to draft final floor plans.

[1] Rodrigues' work (2014) was profoundly valuable and as such adapted for section 2 and 2.3.2. To further explore the evolution on spatial layout algorithms and Rodrigues' EPSAP, consult his work.

Design adaptation approach – This is designed to adapt "*previously stored floor plans for new requirements and constraints that have not been taken into account when the first drafts were produced"*. New solutions are generated using these stored designs and go through consecutive iterations until a satisfactory design is obtained. There is no guarantee though, that for any given solution, the algorithm may not be stuck in a local optimum or even that a single solution can be obtained according to these new requirements and constraints. Basically, "*it is the pursuit for the fittest individual in a very small search region of an already drafted design"*.

Area partitioning approach – When a floor plan is divided into smaller areas, the rooms of the layout are afterwards assigned to those areas according to topological requirements. "Given the fact that "*partitioning is made before the assignment of the design program, these approaches do not guarantee that the geometry will comply with topological requirements"*. In order to solve this issue, several techniques were used such as *Genetic Algorithms, Evolutionary Strategy, Genetic Programming*, or even *Rectangle Dissections with Non-Dominated Sorted Multi-Objective Genetic Algorithm*. This method appears to be the most adequate one to the intended algorithm.

Depending on the researcher's intentions, the algorithms may have different scopes: geometric, topological, related to energy efficiency, the walking space between areas or even the generation of novel shapes to foster creativeness; and variables: such as wall dimensions, communications between areas, wall thickness, door dimensions and thickness, floor levels, furniture, orientations, building boundaries or even adjacent buildings . Topological features, which will be discussed on section 3.2, may refer to opening orientation or relation between spaces.

Usually, *"research works treat each space as an orthogonal shape […], and therefore the use of non-convex shapes is absent"*. This dissertation will treat L-shape floor plans as non-convex.

Problems involving multi-floor space allocation can also be solved by QAP, considering elevators and staircases as means for vertical circulation between floor levels. Researchers first tried to use QAP to solve this issue since it provided an efficient method for designing large spaces with various floor levels such as office buildings, hospitals or big facilities. Methodologies vary from *Heuristic Search Methods, Genetic Algorithms and Programming*, or even *Mixed-Integer Programming*. This type of solution proved to be of difficult implementation, since the building envelope is generally pre-determined and vertical circulations are fixed elements and as such, need to be treated as variables.

2.3.1. Genetic algorithms & genetic programming

"Natural Selection almost inevitably causes much Extinction of the less improved forms of life and induces what I have called Divergence of Character"
Charles Darwin, *The Origin of Species*

Based on Charles Darwin's *Theory of Evolution*, genetic algorithms (GA) are one of the most commonly utilized evolutionary techniques. Transposed to programming language, genetic algorithms simulate natural selection using a heuristic search process which is routinely used to generate useful solutions to optimization and search problems (Mitchell, 1996). It is not guaranteed that results from these algorithms are optimal, as virtually all of the steps involve random elements. On the other hand, this stochastic nature means that it can quickly hone in on good solutions in the population as it does not need to search everything (Flack, 2011).

These algorithms usually begin by creating a list of randomly generated individuals or chromosomes, which represent solutions to the problem. Chromosomes are typically represented as strings of characters which can be transformed into problem solutions. These solutions are evaluated according to some defined fitness function. There is some termination criteria that is checked at this point to see if the GA should continue running. Usually this is a fixed number of generations, a specific target fitness, or the lack of change in fitness for some number of generations. If the termination criteria has not been met, a new population is created. This is done through the use of genetic operators such as crossover and mutation. These stochastically select the fittest chromosomes from the current population, perform some alterations to or combinations between them, and insert them into a new population for the next generation. Since the fittest chromosomes are favoured in the selection process, the population tends to have more and more fit chromosomes with each generation. Eventually, this process is said to converge. That is, the fitness levels of the chromosomes in the new population are no better than those in the current population. There are many causes for this. Most often it is due to the population being clustered around similar or even identically good solutions (Flack, 2011).

"Genetic programming can be considered as an extension of genetic algorithms: instead of using fixed-length strings of characters as the chromosomes or individuals, a more complex computer program is used" (Flack, 2011).

2.3.2. Evolutionary Program for the Space Allocation Problem

Rodrigues (2014) opted for a space allocation approach combined with an evolutionary strategy (ES). *"In addition to being a population-based technique that allows"* to globally search the

space solution, *"it is traditionally known for not having a crossover genetic operator, which has been reported to disrupt or converge the population to similar arrangement of floor plan in other evolutionary algorithms such as genetic algorithms."* The idea is to apply transformation operators which will randomly alter specific elements of the floor plan individual; using genetic programming, these operators will self-adapt and adjust their behaviour according to the evolving stage of the search process, "evolving" the individuals, and discarding poor solutions. This will result on an expected reduction of the computation intensity (Rodrigues, 2014).

Therefore, Rodrigues proposes a technique named *Evolutionary Program for the Space Allocation Problem* (EPSAP), which combines evolutionary strategy (ES) with stochastic hill climbing (SHC), resulting on a hybrid evolutionary technique. According to Rodrigues, *"By combining the two, it is possible to benefit from the known capabilities of a global search by the former and a local search by the latter consisting in a two-stage approach."*

In the first stage, ES randomly generates the initial population of individuals which will consist on a floor plan design with randomly assigned objects. Thus, the genetic operators are *"applied and the fittest individuals are selected for the next generation. The fitness of the individual must be better than the average fitness of the population."*

The second stage, called SHC, is called upon every time a new population of individuals is generated, whereas the individuals are subjected to a set of transformation operations. "*If a given transformation improves or maintains the individual fitness, it is accepted, otherwise, it is rejected. These operators adapt their behaviour according to the evolving stage of each floor plan design."* This process is represented on Figure 2.2, which presents the diagram of Rodrigues' proposed technique.

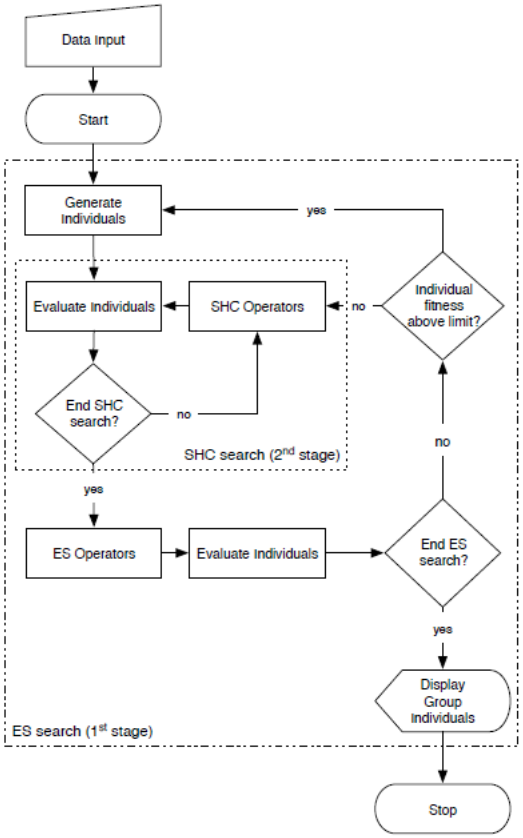


Figure 2.2 – Rodrigues' scheme of his EPSAP algorithm (Rodrigues, 2014).

According to Rodrigues (2014), the assessed objectives may be collected into the following groups depending on the evaluators and area functions:

1. *Connectivity/Adjacency Evaluator objectives:*
- To set connectivity (interior doors);
- To set adjacency.

2. *Overlap Evaluator objectives:*
- To avoid objects overlapping other objects;
- To avoid objects overlapping adjacent buildings.

3. *Space Location Evaluator objective:*
- To place spaces in a specific place in the floor plan.

4. *Opening Overlap Evaluator objectives:*
- To avoid openings overlapping other openings;
- To avoid openings overlapping objects;

- To avoid openings overlapping adjacent buildings.

5. *Opening Orientation Evaluator objective:*
- To set the orientation of openings.

6. *Dimensions Evaluator objectives:*
- To dimension the spaces;
- To dimension the openings.

7. *Compactness Evaluator objective:*
- To set the floor plan compactness among objects.

8. *Overflow Evaluator objective:*
- To avoid the overflow of the building boundary by the objects.

9. *Construction Area Function objective:*
- To limit the floor plan to the maximum construction area.

10. *Gross Area Function objective:*
- To limit the floor plan to the gross area.

Rodrigues method is the algorithm that presents the best solutions within the studied cases.

3. ARCHITECTURAL SPECIFICATIONS – MATHEMATICAL CONSTRAINTS

3.1. Introduction

As already mentioned above, the main objective of this dissertation is to come up with the first steps towards an algorithm that generates floor plans for modular habitations, taking in mind the user's preferences and any companies' pre-fabricated modules solutions. The proposed algorithm uses, as a case study, Martins' proposed solutions of 6x6m modules (Martins, 2011). Nonetheless, these modules may be changed afterwards according to the products of a specific company. This chapter intends to clarify the imposed restrictions implemented so far into the algorithm and inherent to the program, and to illustrate how the user is supposed to interact with the latter, by explaining the proposed inquiry interface.

3.2. Constraints

First and foremost, it is essential to dissect the algorithm as a mathematical problem in three parts: variables to consider, mathematical restrictions, and expected result.

Firstly, one must bear in mind that the implemented algorithm will operate exclusively with square modules of $6x6m^2$ for the time being. Martins has already studied the interior architectonic arrangement of the square modules (Martins, 2011), and their architectures are exclusively studied bearing in mind a two bedroom habitation typology (T2). From her conclusions a group of ten samples were selected: two kitchens, three living rooms with accessible bathrooms, and five rooms with service toilets, see Figure 3.1. The darker areas refer to possible circulation spaces. In the case of the kitchens it refers to an ante-camera/entrance hall.

Figure 3.1 – Set of 10 modules used in the implementation of the algorithm (Martins, 2011).

One of the main advantages of modular construction is the distribution of elements inside each module which is strictly associated to a squared grid whose subdivisions have a fixed dimension *x*. The modules were studied taking in mind an organizational grid of multiples of 0,6m. This way, fitting these modules is undoubtedly facilitated. For instance, openings – doors and, possibly, windows – are confined to one of three places that each side has available for it – let us call them the possibilities of connection. These are at fixed locations in each module – from 0,3m to 2,1m, from 2,1m to 3,9m and from 3,9m to 5,7m, see Figure 3.2. 0,3m were left surrounding the square module which represent the possibility of a wall, exterior or interior. For the present study, this width was not considered relevant. Figure 3.2 presents a representation of a 6x6m^2 module which clarifies these concepts.

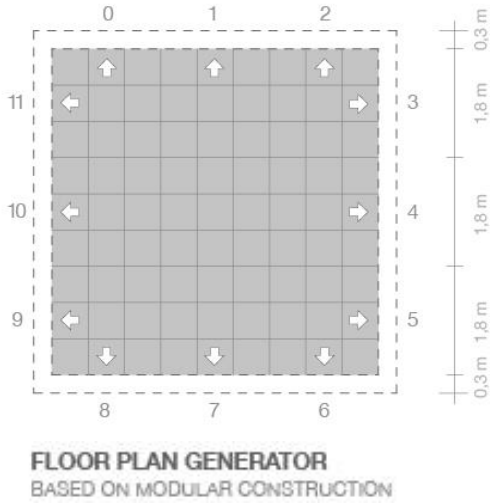


Figure 3.2 - General 6x6m^2 area, with a grid of 0,6m per unit.

These openings correspond to three units of the grid – 3x0,6m. Since a door has a standard dimension of 0,80m on average, it is considered that, if one of the three spaces is occupied within the module and the remaining two are still free, a door can still be allocated on that space. Figure 3.3 clarifies this concept.

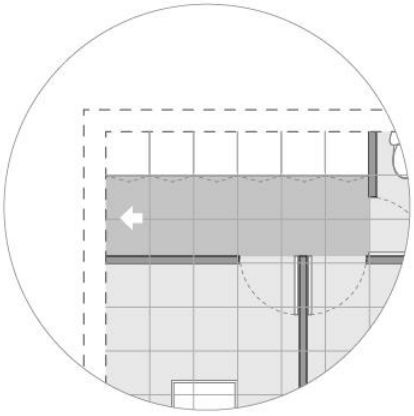


Figure 3.3 – Example of a door with only two units available.

It is important to mention that each module may be allocated in any position. This means that the same modules presented above may undergo 90º clockwise rotations, and may be reflected and rotated again. Therefore, each module presents 8 possible transformations. Figure 3.4 presents an example of a room transformation. Every rotation is clockwise.

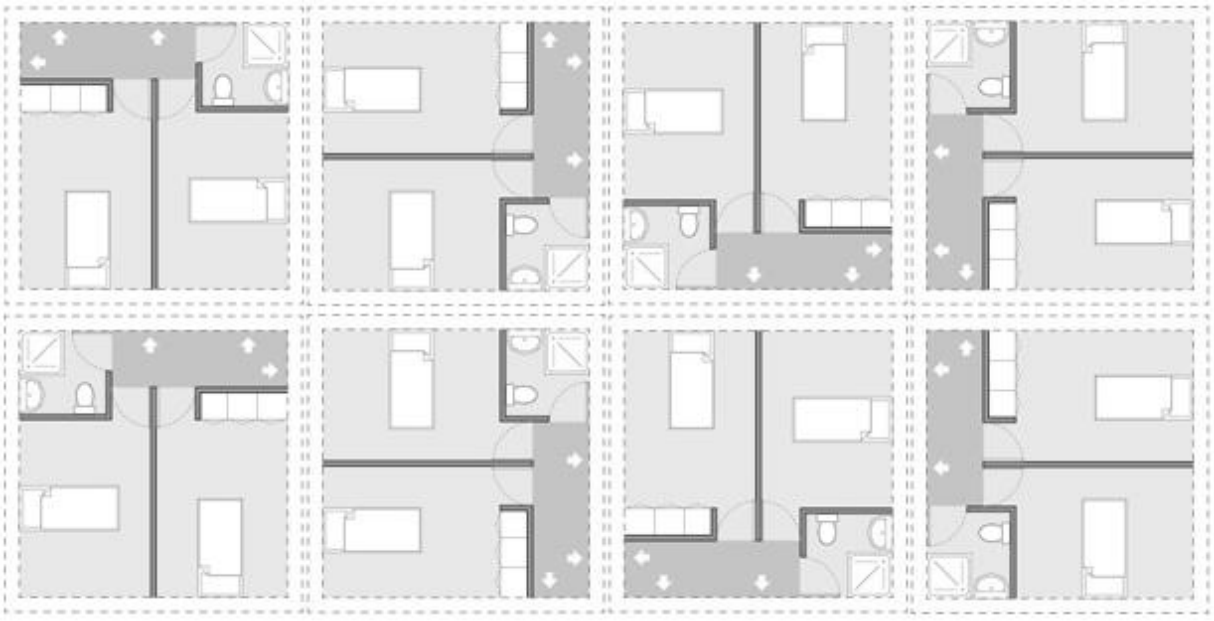


Figure 3.4 – Example of the 8 possible transformations of a module (R1_R0).

At the time of the computer implementation, modules are processed into variables and matrix structures and are later imported in the form of JPG files (computer images) on the output process. Therefore, the algorithm will allocate those variables by testing and inserting them within a predefined space of the allocation area – right to, left to, above of, or below of each modules, according to these possibilities of connection described above.

Therefore the main concern for this algorithm was to agglomerate those pre-defined modules. For this purpose, it is now essential to enumerate and justify what constraints were implemented within the algorithm.

• *Only T2 habitations are considered* – This option was made bearing in mind Martins' conclusions. For the internal arrangement of the modules she suggested, the most adequate space to result from their allocation would be a T2. For other typologies, new modules' arrangements should be made. This implies that only three modules are to be combined for the final solution.

• *Every habitation has one kitchen, one living room, and two rooms* – As already mentioned above, Martins has already studied this part where she separated these different areas within modules. Therefore three modules total are to be allocated, on which both rooms are within the same module.

• *There is at least one possible communication path between each pair of contiguous modules* – This restriction guarantees that each two modules, when connecting with each other, have at least one compatible connection in common since each module has a fixed number of possible connections, resulting in possible and impossible connections. The latter ones are not to be accepted by the algorithm. Figure 3.5 shows examples of these two cases.

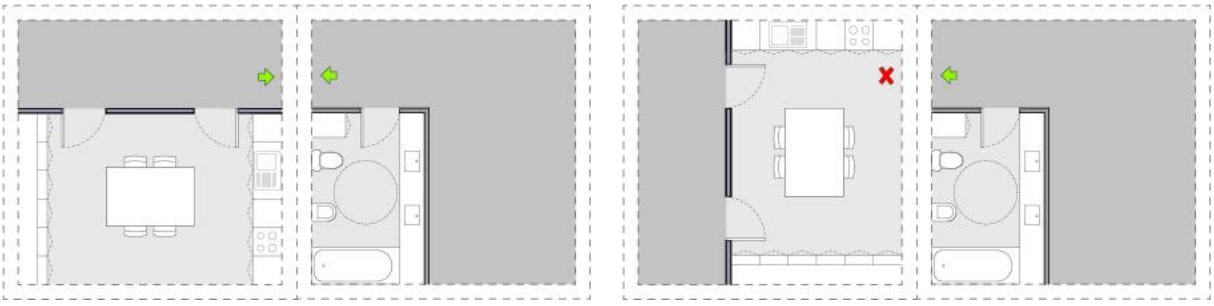


Figure 3.5 – A possible and an impossible communication path between contiguous modules.

• *The habitation shape can be in a L-shape or in line* – see Figure 3.6 which shows these two possibilities, and also see Figure 3.7 which shows discarded configurations for the present work.

Figure 3.6 – L-shape and Line shape configurations of the habitation.

Figure 3.7 – T-Shape and Z-Shape configurations of the habitation: for other typologies.

• *Kitchens cannot be directly connected to bedrooms* – Portuguese regulations do not impose this restriction. Nonetheless it is unwise and unusual to connect such areas directly (Portas, 1996). This is justifiable since residential habitations are usually projected bearing in mind that the house must have two perceptibly separate spaces – a social space and a private space. It is easily perceptible that it is desirable to have an ante-camera separating both areas, otherwise the smells and fumes of that kitchen could easily pass onto the bedrooms. Figure 3.8 clarifies this concept.

Figure 3.8 – A correct solution and an impossible one whereas the rooms are at the middle of the social space.

• *There should be, at least, a communication from the living room or from the ante-camera of the kitchen to the exterior* – Even though the used sample (the ten modules chosen for the implementation and validation of the algorithm) always guarantees an exit from one of these modules, it is fundamental for the algorithm to have this restriction implemented, otherwise, in case the sample was changed, an exit would not be guaranteed within the architecture.

• *Toilets must always be placed alongside with exterior walls; Living rooms must be placed near to, at least, one service toilet; A toilet cannot be placed inside the kitchen* – These three constraints are satisfied even without any additional restriction within the algorithm. This is a consequence of the interior arrangement of the modules.

Last but not least, from the above constraints one may conclude that the correct order for the arrangement of the three modules on the allocation area cannot have rooms, at any point, in the middle, since private space and social space would be intersected.

It is now important to give examples of what should be the expected result from this kind of algorithm. Accordingly, Figures 3.9 and 3.10 present examples of two final T2 habitations: a possible one and an impossible one.

Figure 3.9 - A possible result by the algorithm.

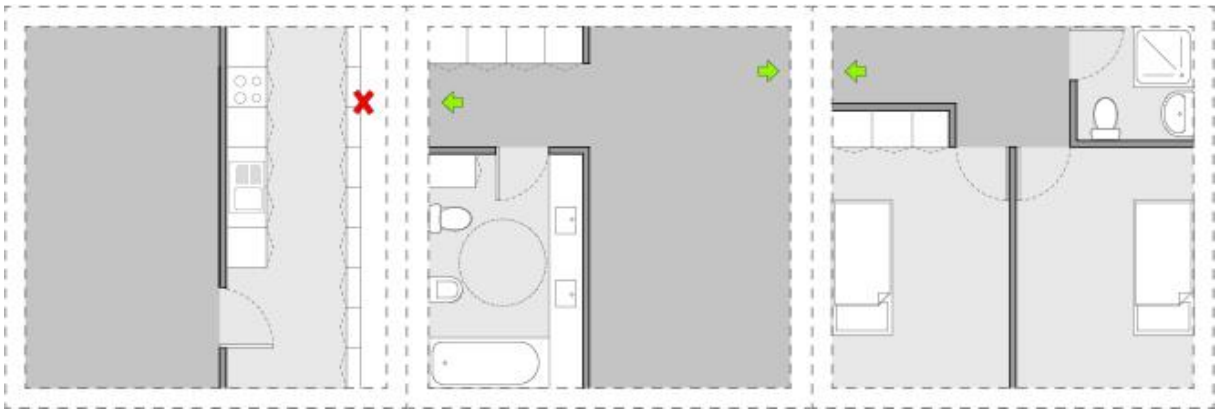


Figure 3.10 - An impossible result by the algorithm.

The purpose of this algorithm is to present every possible combinatorial solution. Hence, every presented result within the program is followed by its solution number, and the total number of solutions, as well as the name of the utilized modules.

3.3. Inquiry and program interface

As suggested by several architectural guidelines, technical drawing lessons or even according to Neufert (2010), one of the most utilized handbooks on the architecture field: before the design process begins there should be an inquiry. Ideally, there should be questions about terrain location, sewers network, compartment necessities – areas, inter-relations, orientation –, available capital, type of construction, not to mention the client's special requirements, neighbourhood buildings' aspects and designs, among other (Neufert, 2010). The list is enormous and depends on the professionals' previous work experience and adopted methods. For this reason, the proposed program starts with an interactive inquiry in order to import some variables that are going to confine final solutions. Moreover, the questions mentioned above

may be translated into mathematical restrictions and added to the inquiry afterwards, at another stage of the development of the algorithm and its respective implementation. Consequently, the program's general scheme is presented on Figure 3.11.

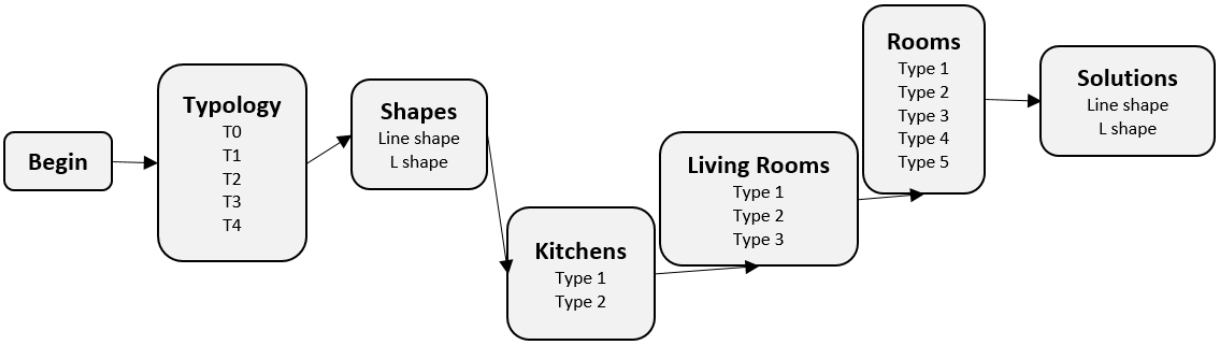


Figure 3.11 – Interface's general scheme.

This Figure shows the several steps which require user's selection. Only after this input will the program run the algorithm which determines the alternative admissible solutions and outputs the correspondent graphical representations and other data. The program initializes with a panel which is the first approach to the interface, see Figure 3.12. By selecting *Begin* button, the program proceeds to the second panel.

Figure 3.12 - Interface's first panel: begin panel.

The second panel begins the interactive inquiry with what is one of the most imperative questions: *"Which habitation type would you like?"*. There is a drop-down list next to the question which offers the possibility of choosing a typology from T0 to T4. It is relevant to mention that the algorithm was developed only for the T2 typology, because this suffices to

illustrate its capabilities. The other options are left for future development of this program. Figure 3.13 presents the second panel and its response in case another typology is chosen besides T2.

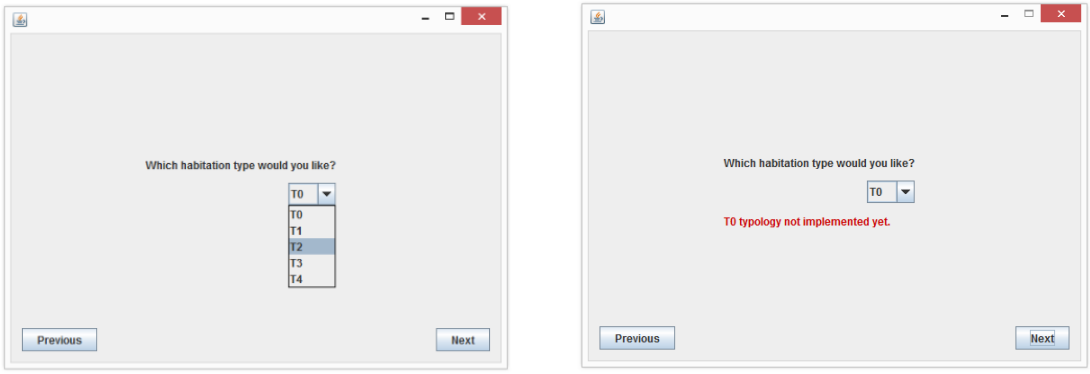


Figure 3.13 - Interface's second panel: typologies option.

After the T2 option is chosen, a third panel shows up, which requires a new parameter: *"Which configuration should the habitation have here?"*. The user can select the final solution of modules in line, L-shape, or both these types. In the case of other typologies, which are left for future developments, the offered options must be changed, according to the number of modules – for instance, if four modules were to be allocated, they could form a perfect square, a L-shape, a T-shape, or even a Z-shape. This variety requires not only the improvement of the proposed algorithm, but also to have new kinds of modules available to combine and allocate. Figure 3.14 shows third the panel.

Figure 3.14 - Interface's third panel: allocation options – L-shape and Line shape.

The following three panels refer to the selection of the eligible modules. They allow the user to choose from a list of available modules the ones that he/she wishes to allocate. Ultimately, the idea is that a company can add as many modules as it desires, and similarly to a catalogue, users can choose any solution. For the present illustration of the program a sample of two kitchens, three living rooms, and five rooms hypothesis were selected, which are specified in Figure 3.1. These panels may be seen on Figure 3.15.

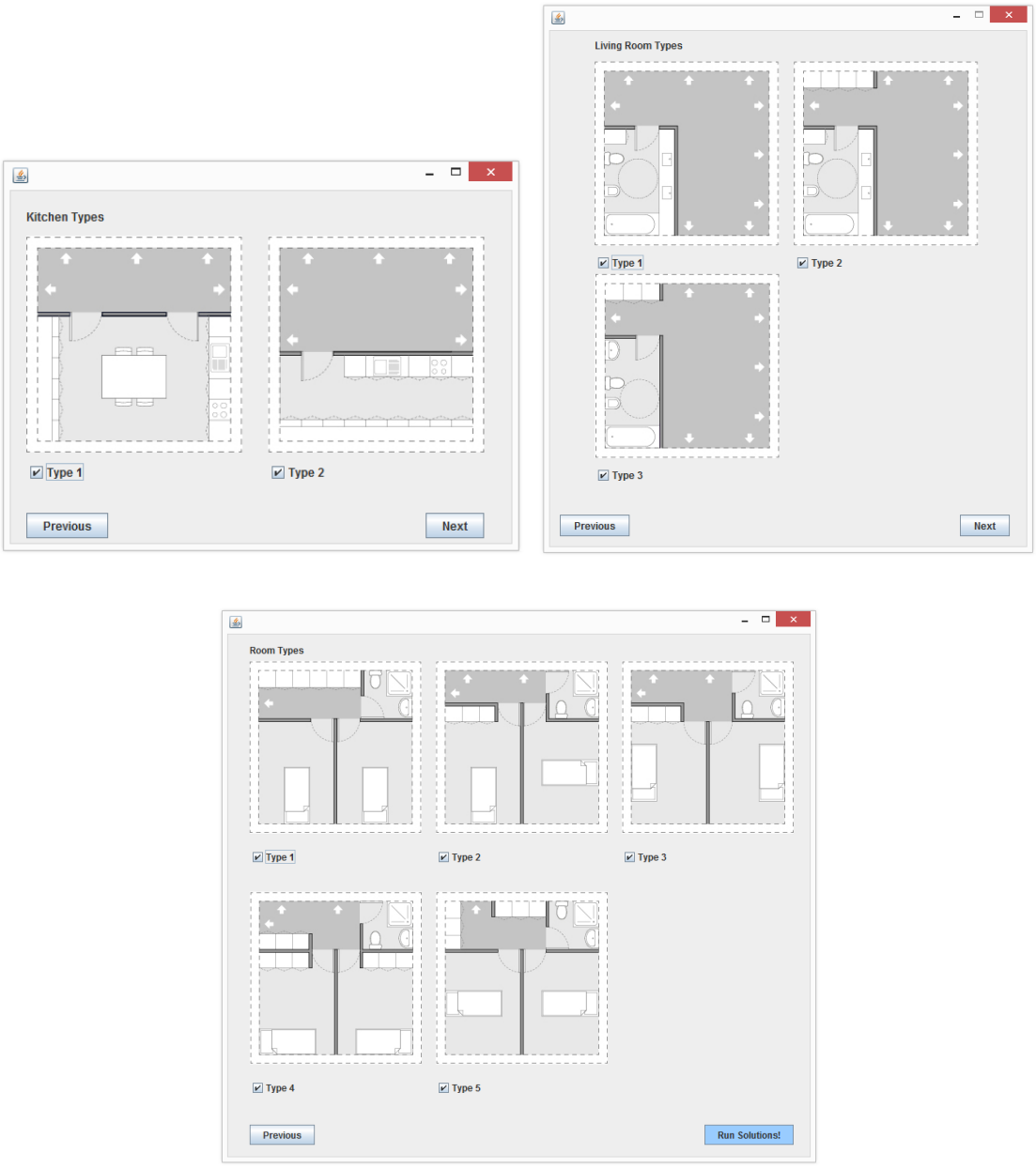


Figure 3.15 - Interface's modules' panel: user's selection.

After inputting the data, the combinatorial algorithm is executed. After the conclusion of this execution, the found optimum alternative will be presented in the following panel(s). This

solutions' panel presents the total number of solutions, and for each of these, its "tag" (reference number), which modules (kitchen, living room, and room) were used, and their relative location. Also, it presents three buttons: *Previous* and *Next* buttons, that offer the possibility of showing the previous/next solution, and *Menu* button, that will lead the user back to the selection interface where he/she is able to redefine his/her module selections and observe new solutions. Figures 3.16 and 3.17 present the layout of the mentioned panels. As it is possible to observe, this allocation will arrange a T2 typology with a kitchen, a living room with an accessible bathroom, and two rooms with a service bathroom.

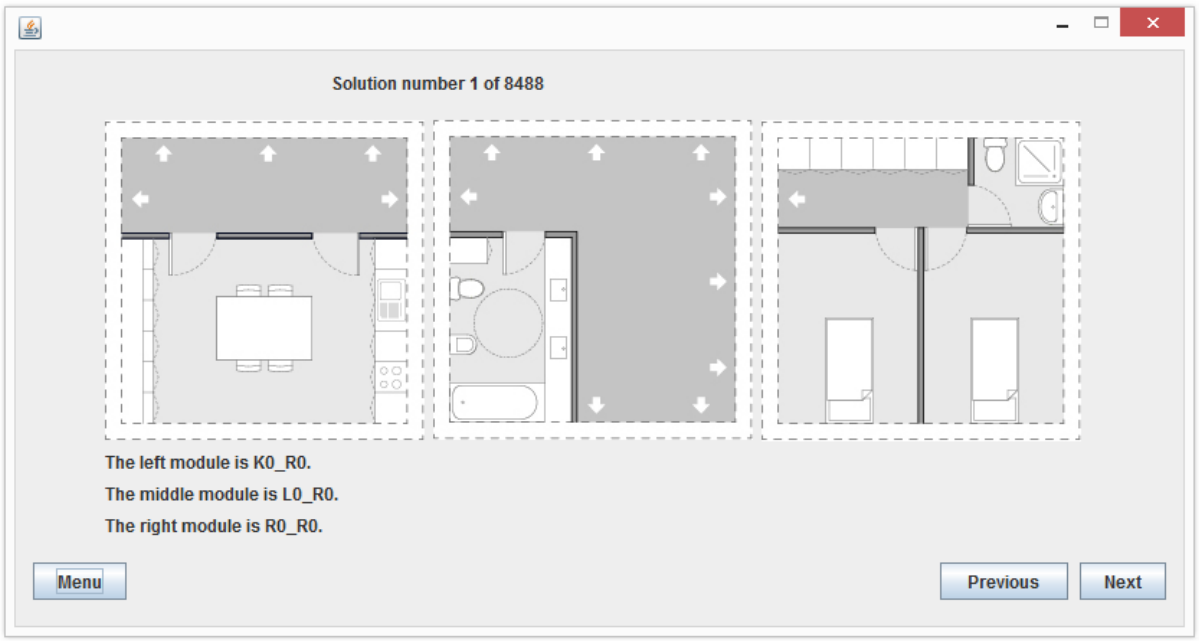


Figure 3.16 - Interface's results' panels – Line shape.

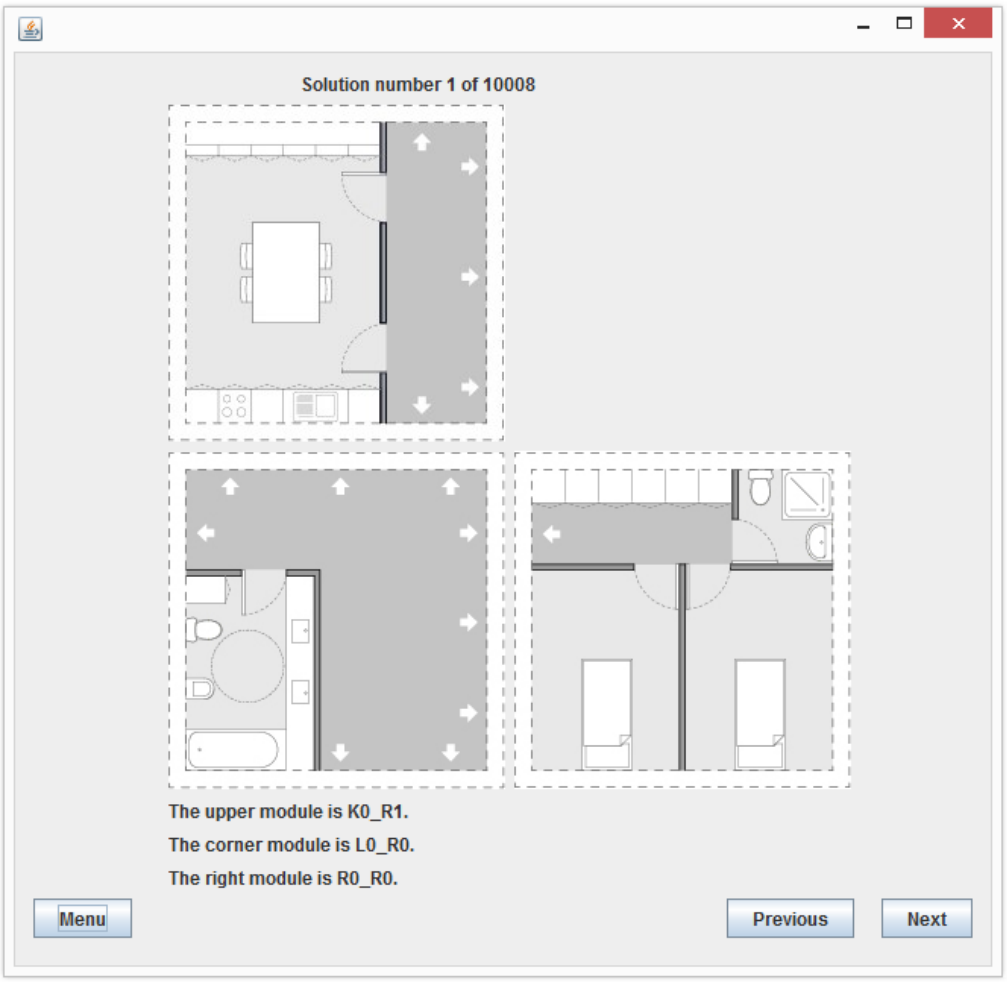


Figure 3.17 - Interface's results' panels – L-shape.

4. ALGORITHM, IMPLEMENTATION AND VALIDATION

4.1. Introduction

The presented dissertation was developed and written by a Civil Engineering student whose algorithmic and programming skills were basic, not to say null, at the very beginning of the semester, and had to be developed exponentially. The first step was to produce a bibliographic review and inquire several IT and computer engineer students, professors and professionals about which programming language should be used for the task of codifying the algorithm. The general choice was Java®, a completely new programming language for the writer – an object-oriented one. This decision was made not only because the majority of the developers use it – Eugénio Rodrigues on EPSAP for example (Rodrigues, 2014) –, but also because it is a leading language in object oriented technologies nowadays, which allows the algorithm to be easily further developed on future studies, if necessary.

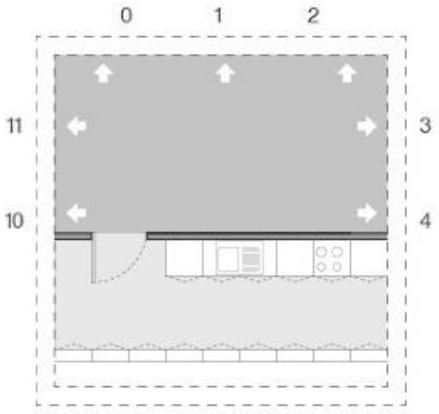
The object oriented approach is a programming paradigm, in terms of the vision that the programmer has about the structure and execution of its program. On object oriented technologies, such as Java, the algorithm may be divided into a collection of objects, with its own attributes and capabilities which interact with each other. Before this paradigm came out, traditional methodologies consisted on programs or algorithms as a list of functions to be followed, almost like a cake recipe. With object oriented technology, each object is capable of receiving messages, process information, to send messages to other objects, or even to evoke them when required. It is even possible to evoke the same object within itself, this being called *recursiveness*, one of the processes utilized in the developed algorithm and program. To provide a helpful visual cue, think of object oriented languages like playing with plasticine. Whenever needed, the programmer can choose from the range of his/her forms, which are his/her classes and methods, and apply them on his/her plasticine.

Moreover, with more than 9 million developers worldwide, Java is the global standard for developing several kinds of applications, games, Web-based content, and enterprise software. An enormous amount of applications and services are currently developed in Java, since it is useful to program high-performance applications for the widest range of computing platforms possible. This versatility and widespread use across heterogeneous environments boosts end-user productivity, communication, and collaboration, as well as dramatically reduces the cost of ownership of both enterprise and consumer applications (Java@, 2015), making it the perfect choice for the implementation of this algorithm.

This chapter will progress by explaining the developed algorithm. However, since there is a huge step from theory to practice, which is without comparison way more complex, it was decided to implement this algorithm and also to present the developed code of the program, which will be used to validate the algorithm.

4.2. Algorithm

The first challenge was to transform the modules' characteristics into mathematical constraints. As previously explained, twelve possibilities of door locations exist for each module. Therefore, a mathematical shortcut was used which consists in a matrix of ones and zeros (Boolean matrix) to be filled. As an illustration of the contents of this matrix, the second type of kitchen (*L1_R0*) is presented on Figure 4.1, together with its corresponding *doors* matrix, in order to perceive this concept.



[1 1 1 | 1 1 0 | 0 0 0 | 0 1 1]

Figure 4.1 – L1_R0 module (2nd Kitchen, Rotation zero).

It can be seen that: (i) the matrix is subdivided into four ordered sub-matrices, corresponding to the north, east, south and west sides; (ii) for each side of the module three possibilities for door locations exist, showing a *"1"* if there is an opening and a *"0"* otherwise.

Then, because every module may be rotated and reflected (and on its turn rotated again), the problem was solved by finding a pattern on each rotation and reflection. Thus, and bearing in mind that all rotations were considered clockwise, in any 90º rotation one may conclude that every three columns progress three places to the right. The matrix of the same kitchen rotated once is now presented, as well as the respective kitchen, on Figure 4.2.

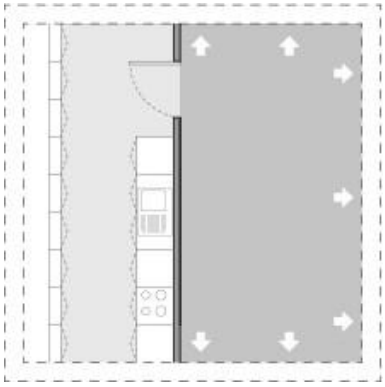
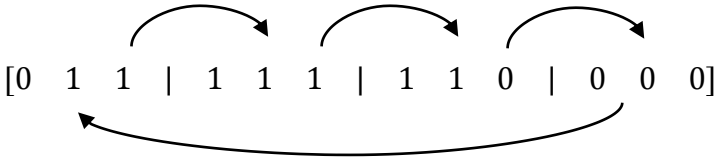


Figure 4.2 - L1_R1 module (2nd Kitchen, Rotation one).



For reflecting the module around a vertical axis, the above module appears as illustrated on Figure 4.3, along with its *doors* matrix.

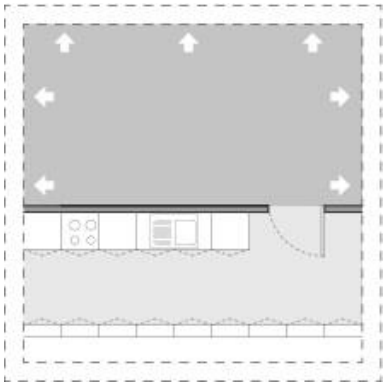
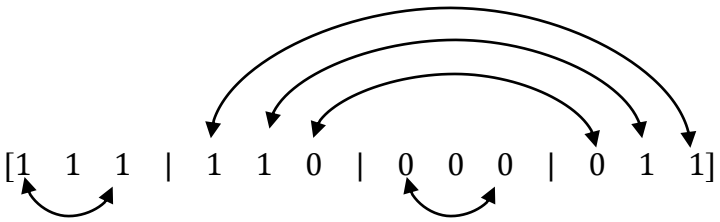


Figure 4.3 - L1_S0 module (2nd Kitchen, Symmetry zero).



In this case: (i) the order of the first terms inside the sub-matrix is inverted, and similarly for the third one; (ii) the order of the other two sub-matrices is also inverted, but these two sub-matrices also swap their position.

In conclusion, it is now possible to define the *doors* matrix, which for the sample of 10 modules and for the 8 types of rotation and reflection will have 80 lines (and 12 columns). Adding a first column to the matrix with the description of the module and the geometric transformation, the result will be the following, partially represented, matrix.

$$\begin{bmatrix} K0\_R0\_ & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ K0\_R1\_ & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ K0\_R2\_ & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ R4\_S3\_ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The first letter of the description may be *K*, *L* or *R*, standing for kitchen, living room, or room. The digit after this letter sub-specifies the module type; for instance *L0, L1, L2* represent three living room types. The second letter may be *R* or *S*, standing for rotation or symmetry. This is followed by an integer ranging from 0 to 3, standing for the number of 90º rotations. It is now possible to deal with modules mathematically, because their interactivity is fully described by this matrix.

As every important piece of code will be presented, described and analysed in detail in the next section, note however that the program was implemented alongside with the development of the algorithm. Therefore, even though they cannot be seen as independent one from the other, a brief description of the algorithm will now be presented. First, there are two distinct subroutines within this work. The first imports the above *doors* matrix from a *txt* file and turns it into variables. The second deals with the general combinatorial problem and deals with *brute forcing* and *recursiveness*. The latter consists on the basic core structure of the work.

The first subroutine uses each line of the text file and separates it into several arrays of strings. Then, it saves the description of the module ("*K0_R0_*", for instance) on a string variable, the first character of the description ("*K*", for instance) on a char variable, and transforms the strings of the bits into integers and saves them on an integer list. Then it uses another method named *place()* in order to place all these elements on their correct place. Figure 4.4 consists on a small diagram which explains this concept for the kitchens arraylist, which solely has two different types.

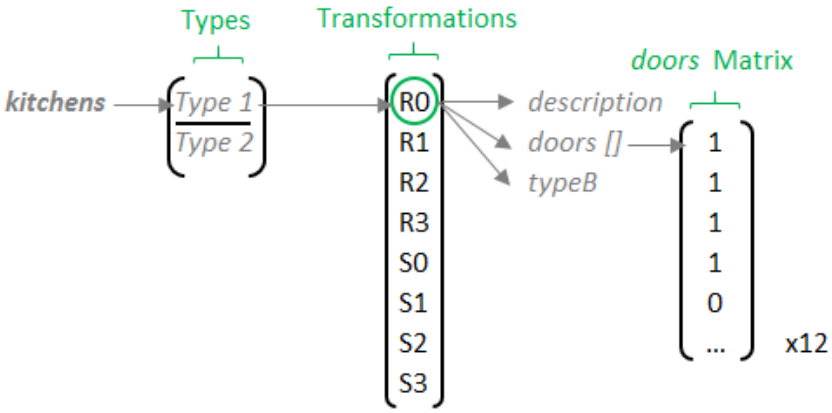


Figure 4.4 - Kitchens arraylist of block objects.

The second subroutine is significantly more complex. It begins by selecting living rooms to be the first allocated modules; this first module always occupies the middle (second of the three possible) position. Therefore, this procedure will be repeated for every living room and for

every kitchen, but never for rooms. Let us suppose that the first module is a living room: for every living room, if selected by the user, for every transformation, the algorithm will allocate one living room. Then the next part of this subroutine will allocate a second module in one of the two still empty possible locations. This means that the programmer may choose both the second module to be allocated and the second space to be filled, which leads to two other structures of code to fill all options: with the same places in the same order to be filled, and changing the order of the modules to be filled. Again as an example, and thinking about the L-shape solution on the first algorithmic structure (first A block on the Figure 4.5), firstly it will allocate a kitchen above the living room, and a room to the right of the living room. Each of these processes uses the same method. Therefore, for each case the code goes into another method, which now allocates the second module, on the second place. Hence, and continuing with the same example, for every kitchen, if chosen by the user, and for every transformation, it allocates a kitchen, if there is at least one connection in common between both modules, which is tested using a new method. Then, if the spaces to be allocated are not totally filled yet – both the third space and module are yet to be filled and allocated respectively, the process is repeated. In other words, the method evokes itself and repeats the procedure, this time for the room modules, making the same tests with respect to the living room previously mentioned.

At the end there is one solution. However, it is only saved on the list of solutions if, for the four sides of the kitchen and of the living room, it verifies that there is at least one exit to the exterior, since no solution can be valid without at least one. This is tested using another method that, for each of the four sides that should have an exit, runs all three possible openings from that side and verifies that there is at least one door. These procedures are repeated as many times as required. Figure 4.5 presents a diagram which intends to make the complete procedure described perfectly clear.

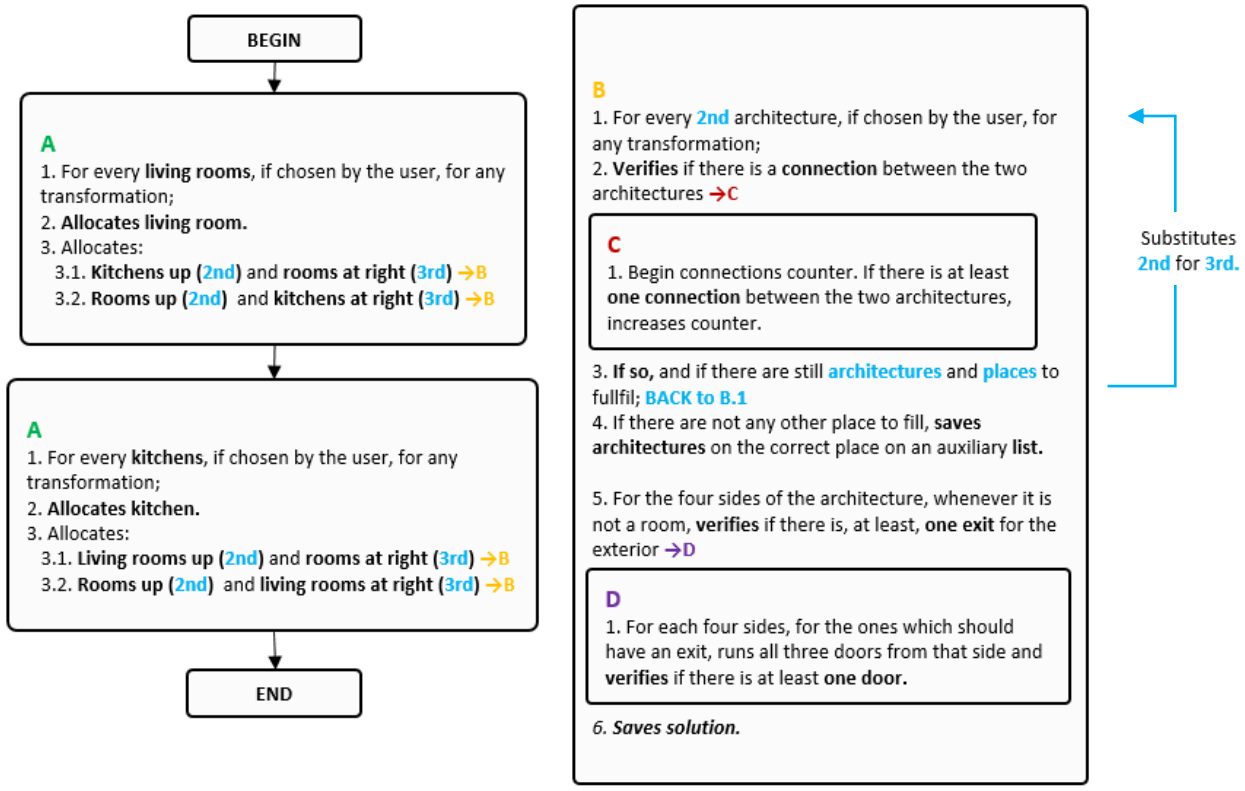


Figure 4.5 – Algorithm's diagram for L-shape solutions.

4.3. Program

Java is an object-oriented language, therefore several classes were used for the code's development: a main class, a class per each panel presented in the program and already described on section 3.3, a *Manager* class which manages all panels, and a class containing the most important algorithmic core within the program, called *Block* class, constituting a total of eleven classes. This section will fully explain the program piece by piece, skipping non-relevant parts such as the automatic code generated by the IDE used. The entire code is fully commented and is included in the appendix at the end of this document, except for the code lines referent to the interface, which were excluded.

To begin with, as in any program, the *main* class initializes the program. The present main declares and initializes (instances) a variable called *manager* from the *Manager* class and sets a *Welcome* panel visible. It is called *TeseMestrado* class.

```
public class TeseMestrado {

    public static void main(String[] args) {

        Manager manager = new Manager();
        manager.Wwelcome.setVisible(true);
```

Next, the *Manager* class is described. This class declares all panels that exist (once again, each panel has its own class), and two Booleans which receive the user's choice of allocation, i.e. in Line or in L-shape. Next it starts two auxiliary variables, *br* and *line*, that will read the *config.txt* file which contains the *doors* matrix, as mentioned on section 3.2. *br* equals the content of config.txt file, and while each line from *config.txt* is different from null, *createBlock* method is invoked for each line as a parameter.

```
public class Manager {

    public Welcome Wwelcome;
    public Habitation Whabitation;
    public Shape Wshape;
    public KitchenT Wkitchen;
    public LivingRT Wliving;
    public RoomT Wroom;
    public SolutionL WsolL;
    public SolutionLine WsolLine;

    public boolean shapeLine;
    public boolean shapeL;

    public Manager(){

        BufferedReader br = null;
        String line = null;

        try {
            br = new BufferedReader(new FileReader("config.txt"));
            while((line = br.readLine()) != null)
                Block.createBlock(line);
        } catch (FileNotFoundException ex) {
            System.out.println("Text file containing doors matrix not found:
config.txt");
            System.exit(0);

        } catch (IOException ex) {
            ex.printStackTrace();
        }

        this.Wwelcome= new Welcome(this);
        this.Whabitation = new Habitation(this);
        this.Wshape = new Shape(this);
        this.Wkitchen = new KitchenT(this);
        this.Wliving = new LivingRT(this);
        this.Wroom = new RoomT(this);
```

From each line the *createBlock* method receives as a parameter it declares and initializes an auxiliary Block *b* with the constructor which receives a string as a parameter (the second constructor in the class). Basically, a *description* string, a *typeB* character and a *doors* matrix

are created. Afterwards these variables are placed within the correct arraylist with the *place* method. *createBlock* method is presented below.

```
public static boolean createBlock(String config)
{
    Block b = new Block(config);
    return b.place();
}
```

The *Block* constructor which receives a string, starts by creating two arrays of string type. The first string consists on splitting the line it receives every time a "_" appears, resulting in a matrix with three columns where the third contains a list of integers (in the form of characters) separated by tab ("\t"). For instance, the first line will result in: *[RO] [S0] [0 0 0 …(bits)]*. Then the second string consists on splitting the third column whenever a tab ("\t") appears, resulting in a matrix with twelve columns. Afterwards, for each column of the third string, it fills the *doors* matrix by transforming the string into an integer, resulting in a Boolean array named *doors[]*. *The description* string is also filled, saving the module name, as well as the *typeB* character, which saves the first character: *K*, *L* or *R*, standing for *Kitchens*, *Living rooms*, or *Rooms*. Pointers are matched to null.

```
public Block(String config) {

    this();
    String[] split = config.split("_");
    String[] bits = split[2].split("\t");

    int i  = 0;
    for(int j=0;j<bits.length;j++)
    {
      doors[i++] = (Integer.parseInt(bits[j])==1);
    }
    description = split[0].concat("_").concat(split[1]);
    typeB=split[0].charAt(0);
    up = left = right = down = null;
}
```

Place method consists on allocating *description*, *typeB* and *doors[]* on the correct position within *kitchens*, *livingRooms* or *rooms* arraylists. These variables are within a block variable, which is one of the possible transformations of a module, that is also within another block variable that stands for the type of the module, and these types are within a final arraylist with all types of modules. Figure 4.4 explains the concept. The *Place* method is presented below.

```
public boolean place()
{
    String split[] = description.split("_");
    char tipo = split[0].charAt(0);
    char transformation = split[1].charAt(0);
    int number = Character.getNumericValue(split[0].charAt(1));
    int pos = Character.getNumericValue(split[1].charAt(1));
```

```
        if(transformation == 'S')
        {
            pos += 4;
        }
        switch (tipo)
        {
            case 'K':
                if(kitchens.size() == number)
                {
                    kitchens.add(new Block[8]);
                }
                kitchens.get(number)[pos] = this;
                return true;

            case 'R':
                if(rooms.size() == number)
                {
                    rooms.add(new Block[8]);
                }
                rooms.get(number)[pos] = this;
                return true;

            case 'L':
                if(livingRooms.size() == number)
                {
                    livingRooms.add(new Block[8]);
                }
                livingRooms.get(number)[pos] = this;
                return true;
        }
        return false;
    }
```

After filling *kitchens, livingRooms* and *rooms* arraylists, one must go back to *Manager* class, which leads us to *Welcome* class. This class is not worth further explaining, since its only objective is to show the first panel. As explained on section 3.3, a begin button leads us now to *Habitation* class. This class allows the user to move on to other panels when he/she selects "*T2 typology*". When this option is chosen and *Next* button is selected, the *Shape* class is evoked. This class' unique purpose is to save two Booleans, *shapeLine* and *shapeL*, which indicate if the user pretends solutions in line or L-shape (or both).

Next, three panels appear whose purpose is to save on a Boolean array (per panel) the user's selection of the types to module he/she chooses to combine for the final solution alternatives. The program does not allow to progress without selecting at least an option of each (a kitchen, a living room and a room). All these classes described are attached to this document, since even though they are not algorithmically relevant, they are all imperative. The arrays described are presented below.

```
    public static boolean[] kitchensAvailability = new boolean[2];
```

```
    public static boolean[] livingRoomsAvailability = new boolean[3];
    public static boolean[] roomsAvailability = new boolean[5];
```

Rooms panel leads to the alternative solutions panels, depending on if the Booleans *shapeLine* and *shapeL* are or are not true. These latter panels present solutions, in line or in a L-shape. Only the solution in L will be described since both algorithms are similar except for the input parameters. Therefore, what *SolutionsL* panel does is to call *solveL* method from *Block* class. This method consists on the implementation of the algorithm developed within this dissertation and hence on its most important part. Let us start by presenting all the variables of *Block* class.

```
    public static ArrayList<Block[]> kitchens = new ArrayList<>();
    public static ArrayList<Block[]> livingRooms = new ArrayList<>();
    public static ArrayList<Block[]> rooms = new ArrayList<>();
    public static boolean[] kitchensAvailability = new boolean[2];
    public static boolean[] livingRoomsAvailability = new boolean[3];
    public static boolean[] roomsAvailability = new boolean[5];
    public static boolean kitchensAvailable, roomsAvailable, livingRoomsAvailable;
    public static ArrayList<Block[]> solutionsLine = new ArrayList<>();
    public static ArrayList<Block[]> solutionsL = new ArrayList<>();

    public boolean[] doors;
    private String description;
    public char typeB;
    public Block up, left, right, down;
```

kitchensAvailable, roomsAvailable, livingRoomsAvailable are indicators of which modules are being allocated. Hence, this method begins by setting the three variables as true. An integer j is declared. *solutionsL* arraylist is also declared and initialized within the method in order to, if the user modifies his/her selection, clean up the previous arraylist.

```
    public static void SolveL()
    {
        solutionsL = new ArrayList<>();
        kitchensAvailable = true;
        roomsAvailable = true;
        livingRoomsAvailable = true;
        int j;
```

This subroutine utilizes a *brute force* search. A brute force method consists on an exhaustive search which looks at every single possible solution for a problem and returns the ideal (optimal) answer, and, basically, on the present case, it consists on saving the ones which fill certain restrictions. The living rooms modules are the first to be allocated and are forced to be at the middle. Then, for every type of living room, if it was chosen by the user, and for each of the 8 transformations a module may have (rotations and symmetries), every four pointers are set to null with the *removeNeighbours* method. Then *checkAreaList* method is applied for each of these modules with certain input parameters, and, finally, *removeNeighbours* method is once again applied. This part of the code is presented below, as well as the *removeNeighbours* method.

```
        livingRoomsAvailable = false;
        j = 0;
        for (Block[] livingRoom : Block.livingRooms) {
            if(livingRoomsAvailability[j++]){
                for (int i = 0; i< 8; i++){
                livingRoom[i].removeNeighbours();
                livingRoom[i].checkAreaList(Block.kitchens, kitchensAvailability,
"up", "right", false);
                livingRoom[i].removeNeighbours();
                }
```

```
    public void removeNeighbours()
    {
       up = left = right = down = null;
    }
```

checkAreaList is crucial for the program and is now going to be described. It receives five parameters: an arraylist of block variables (*blocklist*), a Boolean array (*areaAvailability*), two strings (*direction* and *nextDirection*) and a Boolean (*isLine*). Depending on the *blocklist* it receives, it will turn its availability to false. For instance, if the first arraylist received is *kitchens*, it will set *kitchensAvailable* to false and will start working on kitchen blocks.

```
    public void checkAreaList(ArrayList<Block[]> blockList,boolean[]
areaAvailability, String direction, String nextDiretion, boolean isLine){

      if(blockList == Block.kitchens)
            kitchensAvailable = false;
      if(blockList == Block.rooms)
            roomsAvailable = false;
      if(blockList == Block.livingRooms)
            livingRoomsAvailable = false;
```

Next, a similar process to the one described above occurs: for every type of *blocklist* – which at the time being are the *kitchens* –, if it was chosen by the user, and for each of the 8 transformations a module may have (rotations and symmetries), an auxiliary *block* variable is created. Then, the *living room* block is tested with the *kitchen* block and the *numConnections* method is applied.

```
      Block b;
      int j = 0;
      for (Block[] area : blockList) {
            if(areaAvailability[j++])
                  for (int i = 0; i< 8; i++){
                        b = area[i];
                        if(numConnections(b, direction) > 0){
```

The *numConnections* method basically tests two different modules and returns an integer if there is any possible communication between both modules, and fills pointers. Even though this method seems basic, it is also imperative, and consists on the most important restriction implemented. A switch case is utilized, but just one of the four cases will be presented below, in order to not extend this explanation with repetitions. The other cases are presented in the appendix.

```
public int numConnections(Block b, String positionb)
{
	int connections = 0;
	switch (positionb.toLowerCase()) {

	case "left":
		//   0 1 2     0 1 2
		// 11      3 11      3
		// 10   b  4 10 this 4
		// 9       5 9       5
		//   8 7 6     8 7 6

		if(doors[11] && b.doors[3])
			connections++;
		if(doors[10] && b.doors[4])
			connections++;
		if(doors[9] && b.doors[5])
			connections++;
		if(connections > 0)
		{
			this.left = b;
			b.right = this;
		}
		break;
	}
//(...)
return connections;
}
```

Now back to *checkAreaList* method. After *numConnections* verification the code verifies if *nextDirection* is null. In case it is not, which will happen on the first iteration of this method, the method is called again within itself. This is the called recursiveness – a method which calls itself directly or not. Each recursive method must have a basic case, which must be calculated without using its recursive process, and a recursive step, which should call itself to solve the problem. Usually, the recursive step call is a simplification of the state of the problem. Normal errors may be the incorrect definition of the state of the problem, divergence, or even recursive calls which overlap themselves and lead to exponential complexity (Paquete & Araújo, 2008). Hence, and back to the subroutine, c*heckAreaList* is called again, which will deal with the module that was not combined so far – this being the rooms at this iteration –, and setting next iteration as null – thus on the next iteration it does not go into this process again.

```
//(...)
if(numConnections(b, direction) > 0){
	if(nextDiretion != null){
		if(kitchensAvailable){
			checkAreaList(Block.kitchens, kitchensAvailability,
nextDiretion, null, isLine);
		}
		else if(roomsAvailable){
			checkAreaList(Block.rooms, roomsAvailability, nextDiretion,
null, isLine);
		}
		else if(livingRoomsAvailable){
			checkAreaList(Block.livingRooms, livingRoomsAvailability,
nextDiretion, null, isLine);
		}
	}
```

At this time a solution with a kitchen, a living room and a room which have communications between each other was already found. Therefore, it only remains to verify if there is at least a communication from the kitchen or from the living room to the exterior. Firstly, the solution is allocated on a temporary array of Block variables named *sol*. Then, depending on the fifth parameter of *checkAreaLis*t method – *isLine* Boolean –, the program goes deeper in another *if* structure. Only the *isLine = false* case, which will lead for in a L-shape, is going to be analysed. In this case, four new variables – integers – are declared and initialized: *exitL*, and the arrays *upL*, *cornerL*, and *rightL*. These three arrays contain only ones and zeros (Boolean). One stands for the sides of the module which need to be tested for an exit, i.e. the sides which are not connected to other sides. The solution is then allocated to the temporary array *sol*. Then, for each of the modules found within the solution, and if they are not a room – which does not need an exit door –, *exitL* sums up if there is an exit and immediately breaks. This is tested using the *checkExit* method.

```
else{
	int exitL = 0;
	int upL[]={1,1,0,1};
	int cornerL[]={0,0,1,1};
	int rightL[]={1,1,1,0};

	sol[0] = this.up;
	sol[1] = this;
	sol[2] = this.right;
	for(int g=0;g<3;g++){
		if(sol[g].typeB != 'R'){
			switch(g){
			case(0):exitL += Block.checkExit(upL, sol[0]);break;
			case(1):exitL += Block.checkExit(cornerL, sol[1]);break;
			case(2):exitL += Block.checkExit(rightL, sol[2]);break;
			default:break;
			}
```

```
            }
      }
```

The *checkExit* method basically checks if there is a communication on each of the sides where it is possible to have one. For the four sides, if it can be tested, the method goes to the possible communications of that side, and tests. When it finds one, it immediately returns one. The code may be consulted below.

```
public static int checkExit(int [] exits, Block current){
      for(int i = 0 ; i<4; i++){
            if(exits[i] == 1){
                  for(int j = i*3; j<(i+1)*3; j++){
                        if(current.doors[j] == true){
                              return 1;
                        }
                  }
            }
      }
      return 0;
}
```

Therefore, when it finds a solution, it goes back to *checkAreaList* method and adds the solution to the arraylist of solutions.

```
if(exitL > 0 ){
      solutionsL.add(sol);
```

Concluding, the *checkAreaList* method, the only thing left to do is to set all modules back to true, for the new iteration.

```
if(blockList == Block.kitchens)
      kitchensAvailable = true;
if(blockList == Block.rooms)
      roomsAvailable = true;
if(blockList == Block.livingRooms)
      livingRoomsAvailable = true;
```

At this time, after concluding this process (*checkAreaList*), the program has tried every solution for the rooms on the right and kitchens above the living room, saving on an arraylist every possible solution. Then it leads back to the *removeNeighbours* method where it puts pointers back to null, and repeats this process for every of the eight possible transformations that each type of living rooms has. Then it repeats the process to allocate rooms above and kitchens right, and just after that the program changes for another type of living room. After this, it repeats this group of processes for kitchens at the middle of the solution. This structure is now presented below, without any interruption, for a better understanding of it.

```
kitchensAvailable = false;
j = 0;
for (Block[] kitchen : Block.kitchens) {
    if(kitchensAvailability[j++]){
        for (int i = 0; i< 8; i++){
        kitchen[i].removeNeighbours();
        kitchen[i].checkAreaList(Block.livingRooms, livingRoomsAvailability, "up",
"right", false);
        kitchen[i].removeNeighbours();
        }
        for (int i = 0; i< 8; i++){
        kitchen[i].removeNeighbours();
        kitchen[i].checkAreaList(Block.livingRooms, livingRoomsAvailability,
"right", "up", false);
        kitchen[i].removeNeighbours();
        }
    }
}
kitchensAvailable = true;
```

The *solveL* method finishes by setting all pointers back to null by utilizing the *removeNeighbours* method again.

```
for (Block[] kitchen : Block.kitchens) {
    for (int i = 0; i< 8; i++)
        kitchen[i].removeNeighbours();
}
for (Block[] livingRoom : Block.livingRooms) {
    for (int i = 0; i< 8; i++)
        livingRoom[i].removeNeighbours();
}
for (Block[] room : Block.rooms) {
    for (int i = 0; i< 8; i++)
        room[i].removeNeighbours();
```

This basically concludes the algorithm implementation. From this point on, the code goes back to *solutionsL* panel. This panel (and *solutionsLine* as well) basically imports the solutions list from *Block* class and applies a method called *updateImg*, which essentially imports images within the package of the code that have the exact same names as string the *description* plus "_", from the solutions list. This method is presented below.

```
public void updateImg(){
      if (num < Block.solutionsL.size())
      {
            jLabel4.setText("Solution number " + (num +1) + " of " +
(Block.solutionsL.size()));
            jLabel1.setText("The upper module is " +
Block.solutionsL.get(num)[0].getDescription());
```

```
            jLabel2.setText("The corner module is " +
Block.solutionsL.get(num)[1].getDescription());
            jLabel3.setText("The right module is " +
Block.solutionsL.get(num)[2].getDescription());
            Image1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsL.get(num)[0].getDescription() + "_.jpg")));
            Image2.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsL.get(num)[1].getDescription() + "_.jpg")));
            Image3.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsL.get(num)[2].getDescription() + "_.jpg")));
        }
}
```

At this point, if the user selects the *previous* button, the program goes to the previous admissible solution, while if *next* button is selected, it goes to the next one.

The most relevant parts of the code were described within this section. The complete code is in the appendix at the end of this document, except for the interface's code lines.

4.4. Results

After the complete description of the program, it is now relevant to present some *case studies* and their results. Therefore, and firstly, one must bear in mind that the user is always forced to choose one module of each type, which means that for any input the user selects, there will always be solutions. Several cases were tested but for the section's purpose only seven are presented. Table 4.1 presents the cases and their respective results, both for Line and L-shape solutions, and sums both.

Table 4.1 - Case studies' results.

	Kitchen Types	*Living room Types*	*Room Types*	*Line shape*	*L-shape*	*Total*
Case 1	1	1	1	160	200	***360***
Case 2	2	1	1	184	220	***404***
Case 3	1	2	1	128	160	***288***
Case 4	1; 2	1	1	344	420	***764***
Case 5	1	1; 2; 3	1	416	520	***936***
Case 6	1	1	1; 2; 3; 4; 5	1352	1660	***3012***
Case 7	1; 2	1; 2; 3	1; 2; 3; 4; 5	8488	10008	***18496***

Case 1, *2* and *3* stand for choosing one type of every modules; *case 1* selects all the first types whereas *case 2* selects a different type of kitchen and *case 3* a different type of living room. By

comparing the three one may conclude that: (i) Results coincide on the same range of values; (ii) Results always present pair values and L-shape provides bigger solutions than Line shape; (iii) By summing the cases one may perceive the results for combining different types of modules as seen on *case 4*, which combines the two types of kitchen and thus results on the sum of *case 1* and *2* (despite being an expected and obvious conclusion, one must state it); (iv) Even though the first cases are solely combining one module of each, the results are already excessive to be submitted to a thorough analysis by the user.

Lastly, and further exploring the program's capabilities and aiming for the worst case scenario on which every modules are selected and combined – 2 kitchens, 3 living rooms and 5 rooms – , one concludes once again the fourth point supra-mentioned, but on a larger scale. For this particular case – *case 7* –, results will increase to 18496 solutions, being 8488 for Line shape and 10008 for L-shape. It is certain that no user could accurately analyse this amount of solutions.

This results are easily explained since the problem is combinatorial on its basis, and this kind of problem's approach can easily and quickly explode into massive results, which leads not only to memory capacities' problems but also to an unnecessary amount of data to be analysed. Given the case where there were not any restrictions and the purpose was solely to combine the 2 kitchens, the 3 living rooms and the 5 rooms on a line and taking into account the 8 possible transformations, equation (1) presents the following result.

$$(2x8)x(3x8)x(5x8)x(3x2x1) = 92160 \quad (1)$$

As it is possible to conclude the amount of solution is exorbitant even though the sample being fairly small. Hence the question is: how to reduce the number of results to an analysable output?

A more heuristic approach should be implemented. This means that more restrictions should be applied such as minimal areas parameter, or the automatic adjustment of the interior areas, and, for instance, sieve solutions by groups. Another idea would be to use a genetic approach as described on section 2.3.1, and that Rodrigues applied on section 2.3.2. One other suggestion would be to provide random solutions from which the user would select the ones best preferred. From a sample of some of this user's selection, the program would provide a new generation of random solutions based on the information received, resulting on an iterative process.

Concluding, this results are yet on the field of the T2 typology, on a spatial organization perspective, and not on an architectonic formalization. A first-hand approach to the problem is already presented within this work which is prepared for further development on a near future.

4.5. Conclusions

Even though the title of the present dissertation is "*Development of an algorithm which generates functional floor plans based on modular construction"*, the final result is in fact a program. On a first analysis this program may be perceived as a non-essential part of this dissertation. However, it is the author's belief that an algorithm has no use if not validated, and also because the author could not progress in any algorithmic structure without any programming or algorithmic knowledge in advance. Therefore, the algorithm and its implementation and validation should be seen as a unique piece of work, with a beginning, middle, and an end. One only exists if the other does as well.

The code and algorithm implemented were developed from scratch. This means no pre-existing algorithms or codes were found that could be used for the final objective of this work: generation of functional floor plans based on modular construction. This was a long process which required heavy assistance in programming, and a large amount of time spent studying tutorials and reading books on Java. Hence, one must say that the only way to learn programming is programming.

5. CONCLUSIONS AND FURTHER WORKS

5.1. Conclusions

The algorithm was implemented in Java® 1.8.0_40, executing the timing trials on an Asus with an Intel® Core™ I7-3537U CPU @ 2.0GHz with 4GB of RAM and a 64–bit Operating System. The algorithm was started to be implemented on eclipse© but was completed on NetBeans© IDE, due to its auxiliary graphic component. The algorithm exhibits run time suitable for real-time applications, and for the implemented sample it is capable of generating 18496 different architectures houses in few seconds. It is a brute force search algorithm which obeys to some imposed restrictions, using a recursive method, and that generates plausible residential floor plans for modular construction. Even though this approach does not yet offer a very high degree of automation, which might be considered for this kind of problem, it already enables the user to select some variables which were implemented as rule sets. Nonetheless, this algorithm is novel because it consists on a new approach to automatically combine and generate floor plans based on modular construction. Like any other calculus software, it is believed that these procedural methods may dramatically reduce the need for human labour in the field of engineering, design and architecture, as well as substantially reduce time consumption.

This work represents the writer's first approach to the algorithmic and programming areas, which are, now more than ever, essential for the labour of any engineer, architect, or even designer. Hence, and even though it seems out of context in the structural mechanics field, it is in fact a fundamental work for any person who is interested in this multidisciplinary subject. In today's world every engineer needs to be versatile in his/her labour, hence engineers should not focus entirely on one subject but, in fact, they must have a wide array of multidisciplinary skills in different fields. In conclusion, the writer tries to highlight this importance by incorporating programming into the structural mechanics field, showing the two can, and in fact should, work hand in hand harmoniously in order to allow greater progress on the engineering field.

This dissertation greatly contributed for the writer's development on the Architecture and Informatics field.

5.2. Further works

The developed algorithm aims to design a T2 floor plan architecture through modular construction by combining three square modules on a 2D plane bearing in mind some particular constraints. These modules are of three different types: a kitchen, a living room and a room –

which allocates two separate rooms within itself. Obviously this is yet a really incipient work. Architecture is an enormous, complex and dense world. Creativity is also inherent to the subject, as well as a number of other factors which were not taken into account for this dissertation. As already explained, the objective of this study is to take a step up into this area, and to encourage and promote further development on the matter. Undoubtedly, if a program comparable to AutoCAD© was intended, a big group of developers, from IT and software engineers to civil engineers, architects, designers, managers and modular construction companies, must collaborate to reach what this kind of program can and should be: a fully developed algorithm which tackles all kind of typologies, adds garages, several floors, solar orientation, neighbour buildings, a completely new and vast library of modules to allocate, with different measures, with windows openings on the drawings, and even gardens or open space modules, and to fulfill a wider range of accessibilities restrictions, or to even design more than residential buildings such as hospitals, supermarkets or office buildings, all based on modular construction. In the writer's opinion, the further development of this algorithm with these suggestions would be, without question, a crucial tool for optimizing the time consumption of modular buildings' projects in the future.

The present work is a small step further into this field.

REFERENCES

"RGEU – Regulamento Geral das Edificações Urbanas" (2009). Porto Editora, Porto, Portugal.

coolhaven@ (2015). Cool Haven (official webpage), Coimbra. http://www.coolhaven.pt/conceito/cool-haven-concept.php [viewed at 03/03/2015]

Coyne, R. F. and Flemming, U. (1990). "Planning in Design Synthesis: Abstraction Based LOOS". J. Gero, editor, Springer, Artificial Intelligence in Engineering V, Vol. 1: Design (Proc. Fifth International Conference, Boston, MA), New York, pp. 91-111.

du Plessis, C. (2002). "Agenda 21 for Sustainable Construction in Developing Countries". CIB & UNEP-IETC, Pretoria, South Africa.

Duarte, J. P. (2001). "Customizing Mass Housing: A Discursive Grammar for Siza's Malagueira Houses". Doctoral Thesis, Department of Architecture, Massachusetts Institute of Technology, Massachusetts.

Flack, R. W. J. (2011). "Evolution of Architectural Floor Plans". Master Dissertation, Faculty of Computer Science, Brock University St. Catharines, Ontario.

Flemming, U. (1989). "More on the representation and generation of loosely packed arrangements of rectangles". Environment and Planning B: Planning and Design Vol. 16 Issue 3, pp. 327–359.

Flemming, U. and Woodbury, R. (1995). "Software Environment to Support Early Phases in Building Design (SEED): Overview". Journal of Architectural Engineering, Vol. 1, pp. 147-152.

Galle, P. (1981). "An algorithm for exhaustive generation of building floor plans". Communications of the ACM, Vol. 24, Issue 12, pp. 813–825.

Gero, J.S. & Kazakov, et al. (1996). "Learning and re-using information in space layout planning problems using genetic engineering". Artificial Intelligence in Engineering Vol. 11, pp. 329–334.

Gervásio et al. [Notes from] "Estruturas Metálicas (2014/15)". Department of Civil Engineering of the University of Coimbra, FCTUC.

Groover, M. & Zimmers, E. (1983). "CAD/CAM: Computer-Aided Design and Manufacturing". Pearson Education.

Jackson, H. (2002). "Chapter 11 – Toward a symbiotic coevolutionary approach to architecture". Creative Evolutionary Systems, A volume in The Morgan Kaufmann Series in Artificial Intelligence, pp. 299-313.

java@ (2015). Java (official webpage). https://www.java.com/en/about/ [viewed at 05/08/2015]

Jo, J.H., & Gero, J.S. (1998). "Space layout planning using an evolutionary approach". Artificial Intelligence in Engineering Vol. 12, Issue 3, pp. 149–162.

Kalay, Y.E. (2004). "Architecture's New Media: Principles, Theories and Methods of Computer-Aided Design". Cambridge, Massachusetts, The MIT Press.

Lawson et al, (2014). "Design in Modular Construction". CRC Press, New York.

Liggett, R. S. (2000). "Automated facilities layout: past, present and future". Automation in Construction Vol. 9, Issue 2, pp. 197–215.

lineshapespace@ (2014). Line//Shape//Space (official webpage), Denver. "http://lineshapespace.com/the-benefits-of-modular-construction/" [viewed at 20/08/2015]

Martins, T. (2011). "Desenvolvimento tipológico de habitação unifamiliar a partir da solução base "Coolhaven"". Master Dissertation, Department of Civil Engineering and Architecture, University of Beira Interior, Covilhã.

Martins, T., Murtinho, V., Correia, A., Ferreira, H., Silva, L. (2011). "Avaliação de versatilidade tipológica em módulos de base quadrada com estrutura em aço enformado a frio". VIII Congresso de Construção Metálica e Mista, LNEC, Lisboa. pp. 1–13.

Mitchell, M. (1996). "An Introduction to Genetic Algorithms". Cambridge, MA: MIT Press.

Mitchell,W.J. (1998). "The Logic of Architecture: Design, Computation, and Cognition". 6th ed. Cambridge, MA: MIT Press.

Narayan, K. Lalit (2008). "Computer Aided Design and Manufacturing". New Delhi: Prentice Hall of India. p. 3.

Neufert et al, (2010). "Arte de projetar em arquitetura". Gustavo Gili, Barcelona.

Portas, N. (1996). "Funções e Exigências de Áreas da Habitação". LNEC, Lisboa.

Paquete, L. & Araújo, F. [Notes from] "Laboratório de Programação Avançada (2008/09)". Department of Informatics Engineering of the University of Coimbra, FCTUC.

Rodrigues, E. (2014). "Automated Floor Plan Design – Generation, Simulation, and Optimization". Doctoral Thesis, Department of Mechanical Engineering, Faculty of Sciences and Technologies, Coimbra.

Rosenman, M.A.,& Gero, J.S. (1999). "Evolving designs by generating useful complex gene structures". Bentley, P.J., Ed., Evolutionary Design by Computers, San Francisco, CA: Morgan Kaufmann.

Schnier, T., & Gero, J.S. (1996). "Learning genetic representations as alternative to hand-coded shape grammars". Gero, J.S., & Sudweeks, Eds., Artificial Intelligence in Design, Dordrecht, Kluwer Academic Publ. pp. 39–57.

Serag, A., Ono, S., & Nakayama, S. (2008). "Using interactive evolutionary computation to generate creative building designs". Artificial Life and Robotics, Vol. 13, Issue 1, pp. 246-250.

Weisberg, D. E. (2008). "The Engineering Design Revolution – The People, Companies and Computer Systems That Changed Forever the Practice of Engineering". S. Jamaica Court, Englewood.

Wong, S. S. Y., & Chan, K. C. C. (2009). "EvoArch: An evolutionary algorithm for architectural layout design". Computer-Aided Design, Vol. 41, Issue 9, pp. 649-667.

APPENDIX

A.1. *TeseMestrado* class

```
public class TeseMestrado {

	//code's main. initializes the code
	public static void main(String[] args) {

		//declaration and initialization of a variable (Manager type) by its constructor (go to Manager class)
		Manager manager = new Manager();
		//sets visible 'Welcome' panel which is initialized on Manager
		manager.Wwelcome.setVisible(true);
	}
}
```

A.2. *Welcome* class

```
public class Welcome extends javax.swing.JFrame {

	Manager manager;

	//constructor
	public Welcome(Manager manager) {
		this.manager = manager;
		initComponents();
	}

	//'graphics' automatic code
	/*********DELETED*********/

	//begin button. leads to Habitation class
	private void BeginBTActionPerformed(java.awt.event.ActionEvent evt) {

		manager.Whabitation.setVisible(true);
		this.setVisible(false);

	}


	public static void main(String args[]) {
	//'graphics' automatic code
	/*********DELETED*********/



	// Variables declaration
```

```
        private javax.swing.JButton BeginBT;
        private javax.swing.JFrame jFrame1;
        private javax.swing.JLabel jLabel1;
        private javax.swing.JLabel jLabel2;
        private javax.swing.JLabel jLabel3;
        private javax.swing.JLabel jLabel4;
        private javax.swing.JLabel jLabel5;
        private javax.swing.JLabel jLabel7;
        // End of variables declaration
}
```

A.3. *Manager* class

```
import java.io.*;

public class Manager {

    //declaration of interface's variables. each variable has its own class
    public Welcome Wwelcome;
    public Habitation Whabitation;
    public Shape Wshape;
    public KitchenT Wkitchen;
    public LivingRT Wliving;
    public RoomT Wroom;
    public SolutionL WsolL;
    public SolutionLine WsolLine;

    //Boolean accepts true or false. these variables 'save' user's decision
about a solution in line or L
    public boolean shapeLine;
    public boolean shapeL;

    public Manager() {

        //variables which will read txt file
        BufferedReader br = null;
        String line = null;
        //this debug simply shows in shell the content of ArrayLists<Block[]>
kitchens, livingRooms and rooms
        boolean DEBUG = false;

        try {
            //br equals the content of config.txt file
            br = new BufferedReader(new FileReader("config.txt"));
            //while each line from config.txt is different from null
            while ((line = br.readLine()) != null)
            {
                //createBlock method (from Block class) is used with line as
a parameter (go to Block class, createBlock method)
                Block.createBlock(line);
            }

            if (DEBUG) {
                for (Block[] kitchen : Block.kitchens) {
```

```
                    for (int i = 0; i < 8; i++) {
                        System.out.println(kitchen[i]);
                    }
                }
                for (Block[] kitchen : Block.livingRooms) {
                    for (int i = 0; i < 8; i++) {
                        System.out.println(kitchen[i]);
                    }
                }
                for (Block[] kitchen : Block.rooms) {
                    for (int i = 0; i < 8; i++) {
                        System.out.println(kitchen[i]);
                    }
                }
            }

            //case config file doesn't exist
        } catch (FileNotFoundException ex) {
            System.out.println("Text file containing doors matrix not found:
config.txt");
            //abnormal termination
            System.exit(0);
            //exception handling
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        //creates and initializes panels
        this.Wwelcome = new Welcome(this);//go to Welcome class
        this.Whabitation = new Habitation(this);//go to Habitation class
        this.Wshape = new Shape(this);//go to Shape class
        this.Wkitchen = new KitchenT(this);//go to KitchenT class
        this.Wliving = new LivingRT(this);//go to LivingRT class
        this.Wroom = new RoomT(this);//go to RoomT class
    }
}
```

A.4. *Habitation* class

```
public class Habitation extends javax.swing.JFrame {

      Manager manager;

      //constructor
      public Habitation(Manager manager) {
            this.manager = manager;
            initComponents();
      }



      //'graphics' automatic code
      /*********DELETED*********/

      //next button
```

```
      private void NextBTActionPerformed(java.awt.event.ActionEvent
evt) {
            String op = ListCB.getSelectedItem().toString();
            //if chosen option differs from T2, a text will appear
            if (!op.equals("T2"))

                  ErrorL.setText(op + " typology not implemented
yet.");
            else{
                  //if not, leads to Shape class
                  ErrorL.setText("");
                  manager.Wshape.setVisible(true);
                  this.setVisible(false);
            }
      }

      private void ListCBActionPerformed(java.awt.event.ActionEvent
evt) {
      }

      //previous button. leads to Welcome class
      private void PrevBTActionPerformed(java.awt.event.ActionEvent
evt) {
            this.setVisible(false);
            manager.Wwelcome.setVisible(true);
      }

      public static void main(String args[]) {
      //'graphics' automatic code
      /*********DELETED*********/

      // Variables declaration
      private javax.swing.JLabel ErrorL;
      private javax.swing.JComboBox ListCB;
      private javax.swing.JButton NextBT;
      private javax.swing.JButton PrevBT;
      private javax.swing.JLabel jLabel1;
      // End of variables declaration
}
```

A.5. *Shape* class

```
public class Shape extends javax.swing.JFrame {

      Manager manager;

      //constructor
      public Shape(Manager manager) {
            this.manager = manager;
            initComponents();
            LCB.setSelected(true);
            LineCB.setSelected(true);
      }

      //'graphics' automatic code
```

```
      /*********DELETED*********/

      private void LCBActionPerformed(java.awt.event.ActionEvent evt)
{
      }

      //previous button. leads to Habitation class
      private void PrevBTActionPerformed(java.awt.event.ActionEvent
evt) {
            this.setVisible(false);
            manager.Whabitation.setVisible(true);
            manager.shapeL = LCB.isSelected();
            manager.shapeLine = LineCB.isSelected();
      }

      //next button. leads to KitchenT class if at least one of the
shapes is selected
      private void NextBTActionPerformed(java.awt.event.ActionEvent
evt) {
            manager.shapeL = LCB.isSelected();
            manager.shapeLine = LineCB.isSelected();
            if (manager.shapeL || manager.shapeLine){
                  manager.Wkitchen.setVisible(true);
                  this.setVisible(false);
                  ErrorL.setText("");
            }
            else{
                  ErrorL.setText("Choose at least one of the
options.");
            }
      }

      private void LineCBActionPerformed(java.awt.event.ActionEvent
evt) {
      }

      public static void main(String args[]) {
      //'graphics' automatic code
      /*********DELETED*********/

      // Variables declaration
      private javax.swing.JLabel ErrorL;
      private javax.swing.JCheckBox LCB;
      private javax.swing.JLabel LI;
      private javax.swing.JCheckBox LineCB;
      private javax.swing.JLabel LineI;
      private javax.swing.JButton NextBT;
      private javax.swing.JButton PrevBT;
      private javax.swing.JLabel jLabel1;
      private javax.swing.JLabel jLabel2;
      // End of variables declaration
}
```

A.6. *KitchenT* class

```
public class KitchenT extends javax.swing.JFrame {
```

```
      Manager manager;

      //constructor
      public KitchenT(Manager manager) {
            initComponents();
            this.manager = manager;
            Type1CB.setSelected(true);
            Type2CB.setSelected(true);
      }

      //'graphics' automatic code
      /*********DELETED*********/

      //previous button. leads to Shape class
      private void PrevBTActionPerformed(java.awt.event.ActionEvent
evt) {
            this.setVisible(false);
            manager.Wshape.setVisible(true);
      }

      private void Type1CBActionPerformed(java.awt.event.ActionEvent
evt) {
      }

      //next button
      private void NextBTActionPerformed(java.awt.event.ActionEvent
evt) {
            //fills kitchensAvailability Boolean array
            Block.kitchensAvailability[0] = Type1CB.isSelected();
            Block.kitchensAvailability[1] = Type2CB.isSelected();

            //if at least one of the kitchens is selected
            if (Block.kitchensAvailability[0] ||
Block.kitchensAvailability[1]){
                  //leads to LivingRT class
                  manager.Wliving.setVisible(true);
                  this.setVisible(false);
                  ErrorL.setText("");
            }
            //if no kitchen is selected
            else{
                  ErrorL.setText("Choose at least one of the
options.");
            }
      }
      public static void main(String args[]) {
      //'graphics' automatic code
      /*********DELETED*********/

      // Variables declaration
      private javax.swing.JLabel ErrorL;
      private javax.swing.JLabel Kitchen1I;
      private javax.swing.JLabel Kitchen2I;
      private javax.swing.JButton NextBT;
      private javax.swing.JButton PrevBT;
      private javax.swing.JCheckBox Type1CB;
```

```
      private javax.swing.JCheckBox Type2CB;
      private javax.swing.JLabel jLabel1;
      // End of variables declaration
}
```

A.7. *LivingRT* class

```
public class LivingRT extends javax.swing.JFrame {

    Manager manager;

    //constructor
    public LivingRT(Manager manager) {
        initComponents();
        this.manager = manager;
        Type1CB.setSelected(true);
        Type2CB.setSelected(true);
        Type3CB.setSelected(true);
    }

      //'graphics' automatic code
      /*********DELETED*********/

    private void Type2CBActionPerformed(java.awt.event.ActionEvent
evt) {
    }

    //previous button. leads to KitchenT class
    private void PrevBTActionPerformed(java.awt.event.ActionEvent evt)
{
        this.setVisible(false);
        manager.Wkitchen.setVisible(true);
    }

    //next button
    private void NextBTActionPerformed(java.awt.event.ActionEvent evt)
{
        //fills livingRoomsAvailability Boolean array
        Block.livingRoomsAvailability[0] = Type1CB.isSelected();
        Block.livingRoomsAvailability[1] = Type2CB.isSelected();
        Block.livingRoomsAvailability[2] = Type3CB.isSelected();

        //if at least one of the living rooms is selected
        if (Block.livingRoomsAvailability[0] ||
Block.livingRoomsAvailability[1] || Block.livingRoomsAvailability[2]){
            //leads to RoomT class
            manager.Wroom.setVisible(true);
            this.setVisible(false);
            ErrorL.setText("");
        }
        //if no living room is selected
        else{
            ErrorL.setText("Choose at least one of the options.");
        }
    }
```

```
    public static void main(String args[]) {
      //'graphics' automatic code
      /*********DELETED*********/

    // Variables declaration
    private javax.swing.JLabel ErrorL;
    private javax.swing.JLabel LR1I;
    private javax.swing.JLabel LR2I;
    private javax.swing.JLabel LR3I;
    private javax.swing.JButton NextBT;
    private javax.swing.JButton PrevBT;
    private javax.swing.JCheckBox Type1CB;
    private javax.swing.JCheckBox Type2CB;
    private javax.swing.JCheckBox Type3CB;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration
}
```

A.8. *RoomT* class

```
public class RoomT extends javax.swing.JFrame {

      Manager manager;

      //constructor
      public RoomT(Manager manager) {
            this.manager = manager;
            initComponents();
            Type1CB.setSelected(true);
            Type2CB.setSelected(true);
            Type3CB.setSelected(true);
            Type4CB.setSelected(true);
            Type5CB.setSelected(true);
      }

      //'graphics' automatic code
      /*********DELETED*********/

      //previous button. leads to LivingRT class
      private void PrevBTActionPerformed(java.awt.event.ActionEvent evt) {
            this.setVisible(false);
            manager.Wliving.setVisible(true);
      }

      //next button
      private void SolBTActionPerformed(java.awt.event.ActionEvent evt) {
            //fills roomsAvailability Boolean array
            Block.roomsAvailability[0] = Type1CB.isSelected();
            Block.roomsAvailability[1] = Type2CB.isSelected();
            Block.roomsAvailability[2] = Type3CB.isSelected();
            Block.roomsAvailability[3] = Type4CB.isSelected();
            Block.roomsAvailability[4] = Type5CB.isSelected();
```

```
            //if at least one of the rooms is selected
            if (Block.roomsAvailability[0] || Block.roomsAvailability[1] ||
Block.roomsAvailability[2] || Block.roomsAvailability[3] ||
Block.roomsAvailability[4]){
                  this.setVisible(false);
                  ErrorL.setText("");
                  //if L shape was selected (manager.shapeL is a Boolean)
                  if (manager.shapeL)
                  {
                        //leads to SolutionL class and sets it visible
                        manager.WsolL = new SolutionL(manager);
                        manager.WsolL.setVisible(true);
                  }
                  //if line shape was selected (manager.shapeLine is a
boolean)
                  if (manager.shapeLine)
                  {
                        //leads to SolutionLine class and sets it visible
                        manager.WsolLine = new SolutionLine (manager);
                        manager.WsolLine.setVisible(true);
                  }

            }

            else{
                  ErrorL.setText("Choose at least one of the options.");
            }

      }


      public static void main(String args[]) {
      //'graphics' automatic code
      /*********DELETED*********/

      // Variables declaration
      private javax.swing.JLabel ErrorL;
      private javax.swing.JButton PrevBT;
      private javax.swing.JButton SolBT;
      private javax.swing.JCheckBox Type1CB;
      private javax.swing.JCheckBox Type2CB;
      private javax.swing.JCheckBox Type3CB;
      private javax.swing.JCheckBox Type4CB;
      private javax.swing.JCheckBox Type5CB;
      private javax.swing.JLabel jLabel1;
      private javax.swing.JLabel jLabel2;
      private javax.swing.JLabel jLabel3;
      private javax.swing.JLabel jLabel4;
      private javax.swing.JLabel jLabel5;
      private javax.swing.JLabel jLabel6;
      // End of variables declaration
}
```

A.9. *SolutionsL* class

```
public class SolutionL extends javax.swing.JFrame {

    Manager manager;
    int num;

    //constructor
    public SolutionL(Manager manager) {
        initComponents();
        this.manager = manager;
        //SolveL() method from Block class
        Block.SolveL();
        //begins counter
        num = 0;
        //updateImg() method from this class
        updateImg();
    }

       //'graphics' automatic code
       /*********DELETED*********/

    private void PreviousBTActionPerformed(java.awt.event.ActionEvent evt) {

        //if counter is bigger than 0 (in case num=0, nothing happens)
        if (num > 0) {
            num--;
        }
        updateImg();
    }

    private void MenuBTActionPerformed(java.awt.event.ActionEvent evt) {
        manager.Wroom.setVisible(true);
        this.setVisible(false);
    }

    private void NextBTActionPerformed(java.awt.event.ActionEvent evt) {
        //if counter is lower than the total number of solutions (in case num
equals the total number of solutions, nothing happens)
        if (num < Block.solutionsL.size() - 1) {
            num++;
        }
        updateImg();
    }
    //updates the interfaces: the nr of solutions, the descriptions and the
images
    public void updateImg() {
        if (num < Block.solutionsL.size()) {
            jLabel4.setText("Solution number " + (num + 1) + " of " +
(Block.solutionsL.size()));
            jLabel1.setText("The upper module is " +
Block.solutionsL.get(num)[0].getDescription());
```

```
            jLabel2.setText("The corner module is " +
Block.solutionsL.get(num)[1].getDescription());
            jLabel3.setText("The right module is " +
Block.solutionsL.get(num)[2].getDescription());
            Image1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsL.get(num)[0].getDescription() + "_.jpg")));
            Image2.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsL.get(num)[1].getDescription() + "_.jpg")));
            Image3.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsL.get(num)[2].getDescription() + "_.jpg")));
        }
    }

    public static void main(String args[]) {
       //'graphics' automatic code
       /*********DELETED*********/

    // Variables declaration
    private javax.swing.JLabel Image1;
    private javax.swing.JLabel Image2;
    private javax.swing.JLabel Image3;
    private javax.swing.JButton MenuBT;
    private javax.swing.JButton NextBT;
    private javax.swing.JButton PreviousBT;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    // End of variables declaration
}
```

A.10. *SolutionsLine* class

```
//this class is similar to SolutionL class, but for lines solutions
public class SolutionLine extends javax.swing.JFrame {

      Manager manager;
      int num;

      public SolutionLine(Manager manager) {
            initComponents();
            this.manager = manager;
            Block.SolveLine();
            num = 0;
            updateImg();
      }

      //'graphics' automatic code
      /*********DELETED*********/
```

```
      private void MenuBTActionPerformed(java.awt.event.ActionEvent
evt) {
            manager.Wroom.setVisible(true);
            this.setVisible(false);
      }

      private void
PreviousBTActionPerformed(java.awt.event.ActionEvent evt) {
            if(num>0)
            {
                  num--;
            }
            updateImg();
      }

      private void NextBTActionPerformed(java.awt.event.ActionEvent
evt) {
            if(num<Block.solutionsLine.size()-1)
            {
                  num++;
            }
            updateImg();
      }

      public void updateImg()
      {
            if (num < Block.solutionsLine.size())
            {
                  jLabel1.setText("Solution number " + (num +1) + " of
" + (Block.solutionsLine.size()));
                  jLabel2.setText("The left module is " +
Block.solutionsLine.get(num)[0].getDescription() + ".");
                  jLabel3.setText("The middle module is " +
Block.solutionsLine.get(num)[1].getDescription() + ".");
                  jLabel4.setText("The right module is " +
Block.solutionsLine.get(num)[2].getDescription() + ".");
                  Image1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsLine.get(num)[0].getDescription() + "_.jpg")));
                  Image2.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsLine.get(num)[1].getDescription() + "_.jpg")));
                  Image3.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/Imagens/" +
Block.solutionsLine.get(num)[2].getDescription() + "_.jpg")));
            }
      }

      public static void main(String args[]) {
      //'graphics' automatic code
      /*********DELETED*********/

      // Variables declaration
      private javax.swing.JLabel Image1;
      private javax.swing.JLabel Image2;
      private javax.swing.JLabel Image3;
      private javax.swing.JButton MenuBT;
      private javax.swing.JButton NextBT;
```

```
        private javax.swing.JButton PreviousBT;
        private javax.swing.JLabel jLabel1;
        private javax.swing.JLabel jLabel2;
        private javax.swing.JLabel jLabel3;
        private javax.swing.JLabel jLabel4;
        // End of variables declaration
}
```

A.11. *Block* class

```
import java.util.ArrayList;

//block's class initialization
public class Block {

        //variables
        //arraylists of block type with areas allocated
        public static ArrayList<Block[]> kitchens = new ArrayList<>();
        public static ArrayList<Block[]> livingRooms = new ArrayList<>();
        public static ArrayList<Block[]> rooms = new ArrayList<>();
        //essentially indicates which areas the user would like to combine
        public static boolean[] kitchensAvailability = new boolean[100];
        public static boolean[] livingRoomsAvailability = new boolean[100];
        public static boolean[] roomsAvailability = new boolean[100];
        //indicates the availability of a space
        public static boolean kitchensAvailable, roomsAvailable,
livingRoomsAvailable;
        //lists of solutions in line and L
        public static ArrayList<Block[]> solutionsLine = new ArrayList<>();
        public static ArrayList<Block[]> solutionsL = new ArrayList<>();

        public boolean[] doors;
        private String description;
        //indicates 'K', 'L' or 'R'
        public char typeB;
        //pointers of Block type
        public Block up, left, right, down;

        //default constructor
        public Block() {
                //creates doors matrix
                doors = new boolean[12];
                //clean pointers
                up = left = right = down = null;
        }

        //constructor which receives a string
        public Block(String config) {

                //represents the physical memory address location of the object
which is being utilized
                this();
                //creates an array of string type by splitting on "_". Ex: [RO]
[S0] [bits...]
```

```
            String[] split = config.split("_");
            //creates an array of string type by splitting on tabs the 3rd
column of split array. Ex: [0] [0] [1] [1] [0] ...
            String[] bits = split[2].split("\t");

            int i = 0;
            //for each door
            for (int j = 0; j < bits.length; j++) {
                  //fills doors matrix
                  doors[i++] = (Integer.parseInt(bits[j]) == 1);
            }
            //returns a string. Ex: "R0_S0"
            description =
split[0].concat("_").concat(split[1]);//concat():Returns a string that is the
result of concatenating two or more string values.
            //returns 'K', 'L' or 'R'
            typeB = split[0].charAt(0);
            //clean pointers
            up = left = right = down = null;
      }

      //place method
      public boolean place() {
            //fills array of string type split[]. Ex: K0_R0 -> [K0][R0]
            String split[] = description.split("_");
            //fills character tipo. Ex: [K0] -> K. For the implemented code,
only K(itchens), L(iving rooms) and R(ooms) are being used.
            char tipo = split[0].charAt(0);
            //fills character transformation. Ex: [R0] -> R. For the
implemented code, only R(otate) and S(ymmetric) are being used.
            char transformation = split[1].charAt(0);
            //fills integer number with the integer correspondent to the
second character at split[0]. Ex: [K0] -> 0.
            int number = Character.getNumericValue(split[0].charAt(1));
            //fills integer pos with the integer correspondent to the second
character at split[1]. Ex: [R0] -> 0.
            int pos = Character.getNumericValue(split[1].charAt(1));
            //when transformation is symmetric, sums 4 to pos. Ex: R->0-3,
S->4-7.
            if (transformation == 'S') {
                  pos += 4;
            }
            switch (tipo) {
            //case tipo = 'K'
            case 'K':
                  //at each cycle it increases ArrayList<Block[]> kitchens
size (allocated space on memory). If it is equal to the present number, if is
used.
                  if (kitchens.size() == number) {
                        //adds an array of 8 Blocks on each space of
kitchen
                        kitchens.add(new Block[8]);
                  }
                  //allocates present variables(this)on the correct array
                  kitchens.get(number)[pos] = this;
```

```
                    //gives a signal that an module has been placed (place
method returns boolean)
                    return true;

                    //case tipo = 'R'; same as above.
              case 'R':
                    if (rooms.size() == number) {
                          rooms.add(new Block[8]);
                    }
                    rooms.get(number)[pos] = this;
                    return true;

                    //case tipo = 'L'; same as above.
              case 'L':
                    if (livingRooms.size() == number) {
                          livingRooms.add(new Block[8]);
                    }
                    livingRooms.get(number)[pos] = this;
                    return true;
              }
              return false;
        }

        //SolveLine method. For more detail read SolveL method.
        public static void SolveLine() {
              solutionsLine = new ArrayList<>();
              kitchensAvailable = true;
              livingRoomsAvailable = true;
              roomsAvailable = true;
              int j;

              livingRoomsAvailable = false;
              j = 0;
              for (Block[] livingRoom : Block.livingRooms) {
                    if (livingRoomsAvailability[j++]) {
                          for (int i = 0; i < 8; i++) {
                                livingRoom[i].removeNeighbours();
                                livingRoom[i].checkAreaList(Block.kitchens,
kitchensAvailability, "left", "right", true);
                                livingRoom[i].removeNeighbours();
                          }
                          for (int i = 0; i < 8; i++) {
                                livingRoom[i].removeNeighbours();
                                livingRoom[i].checkAreaList(Block.kitchens,
kitchensAvailability, "right", "left", true);
                                livingRoom[i].removeNeighbours();
                          }
                    }
              }
              livingRoomsAvailable = true;

              kitchensAvailable = false;
              j = 0;
              for (Block[] livingRoom : Block.kitchens) {
                    if (kitchensAvailability[j++]) {
                          for (int i = 0; i < 8; i++) {
```

```
                                    livingRoom[i].removeNeighbours();

            livingRoom[i].checkAreaList(Block.livingRooms,
livingRoomsAvailability, "left", "right", true);
                                    livingRoom[i].removeNeighbours();
                              }
                              for (int i = 0; i < 8; i++) {
                                    livingRoom[i].removeNeighbours();

            livingRoom[i].checkAreaList(Block.livingRooms,
livingRoomsAvailability, "right", "left", true);
                                    livingRoom[i].removeNeighbours();
                              }
                        }
                  }
                  kitchensAvailable = true;

                  for (Block[] kitchen : Block.kitchens) {
                        for (int i = 0; i < 8; i++) {
                              kitchen[i].removeNeighbours();
                        }
                  }
                  for (Block[] livingRoom : Block.livingRooms) {
                        for (int i = 0; i < 8; i++) {
                              livingRoom[i].removeNeighbours();
                        }
                  }
                  for (Block[] room : Block.rooms) {
                        for (int i = 0; i < 8; i++) {
                              room[i].removeNeighbours();
                        }
                  }
            }

            //SolveL method
            public static void SolveL() {
                  //if user goes back to menu and rearrange selection, a new array
is filled
                  solutionsL = new ArrayList<>();
                  //puts all modules available to allocate
                  kitchensAvailable = true;
                  roomsAvailable = true;
                  livingRoomsAvailable = true;
                  int j;

                  //forcing living rooms to be in the middle; first to be
allocated
                  livingRoomsAvailable = false;
                  j = 0;
                  //for any type of living rooms
                  for (Block[] livingRoom : Block.livingRooms) {
                        //if any living room which was selected by the user
                        if (livingRoomsAvailability[j++]) {
                              //for any of the 8 possibilities a typology has
                              for (int i = 0; i < 8; i++) {
```

```
                                    //removeNeighbours method applied to
livingRoom[i] (please read removeNeighbours method)
                                    livingRoom[i].removeNeighbours();
                                    //checkAreaList method applied to
livingRoom[i] with these input parameters (please read checkAreaList method)
                                    //kitchens will be tested above, and rooms
right to the living rooms
                                    livingRoom[i].checkAreaList(Block.kitchens,
kitchensAvailability, "up", "right", false);
                                    livingRoom[i].removeNeighbours();
                              }
                              //same as above. this time, kitchens will be tested
at right, and rooms above the living rooms
                              for (int i = 0; i < 8; i++) {
                                    livingRoom[i].removeNeighbours();
                                    livingRoom[i].checkAreaList(Block.kitchens,
kitchensAvailability, "right", "up", false);
                                    livingRoom[i].removeNeighbours();
                              }
                        }
                  }
                  livingRoomsAvailable = true;

                  //same thing but now for kitchens at middle
                  kitchensAvailable = false;
                  j = 0;

                  for (Block[] kitchen : Block.kitchens) {
                        if (kitchensAvailability[j++]) {
                              for (int i = 0; i < 8; i++) {
                                    kitchen[i].removeNeighbours();
                                    kitchen[i].checkAreaList(Block.livingRooms,
livingRoomsAvailability, "up", "right", false);
                                    kitchen[i].removeNeighbours();
                              }
                              for (int i = 0; i < 8; i++) {
                                    kitchen[i].removeNeighbours();
                                    kitchen[i].checkAreaList(Block.livingRooms,
livingRoomsAvailability, "right", "up", false);
                                    kitchen[i].removeNeighbours();
                              }
                        }
                  }
                  kitchensAvailable = true;

                  //puts all pointers back to null (security check)
                  for (Block[] kitchen : Block.kitchens) {
                        for (int i = 0; i < 8; i++) {
                              kitchen[i].removeNeighbours();
                        }
                  }
                  for (Block[] livingRoom : Block.livingRooms) {
                        for (int i = 0; i < 8; i++) {
                              livingRoom[i].removeNeighbours();
                        }
                  }
```

```
            for (Block[] room : Block.rooms) {
                  for (int i = 0; i < 8; i++) {
                        room[i].removeNeighbours();
                  }
            }
      }

      //checkAreaList method. it has 5 input parameters, as seen below
      public void checkAreaList(ArrayList<Block[]> blockList, boolean[]
areaAvailability, String direction, String nextDiretion, boolean isLine) {
            //puts the following module (blocklist) to be tested false
            if (blockList == Block.kitchens) {
                  kitchensAvailable = false;
            }
            if (blockList == Block.rooms) {
                  roomsAvailable = false;
            }
            if (blockList == Block.livingRooms) {
                  livingRoomsAvailable = false;
            }

            //auxiliary block b (from default constructor)
            Block b;
            int j = 0;
            for (Block[] area : blockList) {
                  //if an option was selected by the user
                  if (areaAvailability[j++]) //for the 8 types of
transformations
                  {
                        for (int i = 0; i < 8; i++) {
                              //each transformation
                              b = area[i];
                              //if integer from numConnections method is
bigger than 0. At this point, if auxiliary module (b) can connect to the
current module being tested(this)
                              if (numConnections(b, direction) > 0) {
                                    //at this point 2 of 3 blocks are
already defined
                                    if (nextDiretion != null) {
                                          //if there's still an area left
to fill, checkAreaList is called inside its own method
                                          if (kitchensAvailable) {

      checkAreaList(Block.kitchens, kitchensAvailability, nextDiretion,
null, isLine);
                                          } else if (roomsAvailable) {

      checkAreaList(Block.rooms, roomsAvailability, nextDiretion, null,
isLine);
                                          } else if
(livingRoomsAvailable) {

      checkAreaList(Block.livingRooms, livingRoomsAvailability,
nextDiretion, null, isLine);
                                          }
                                    } //if all blocks are defined
```

```
                                        else {
                                              //auxiliary array of blocks sol
                                              Block[] sol = new Block[3];

                                              //if isLine=true (line shape)
                                              if (isLine) {
                                                    int exitLine = 0;
                                                    int leftLine[] = {1, 0,
1, 1};
                                                    int middleLine[] = {1,
0, 1, 0};
                                                    int rightLine[] = {1, 1,
1, 0};


                                                    sol[0] = this.left;
                                                    sol[1] = this;
                                                    sol[2] = this.right;


                                                    for (int g = 0; g < 3;
g++) {
                                                          //if one of the
modules is a Kitchen or a Living Room
                                                          if (sol[g].typeB
!= 'R') {
                                                                switch (g)
{
                                                                //checks if
there is an exit at each position; breaks immediately after finding one
                                                                case (0):

      exitLine += Block.checkExit(leftLine, sol[0]);
                                                                break;
                                                                case (1):

      exitLine += Block.checkExit(middleLine, sol[1]);
                                                                break;
                                                                case (2):

      exitLine += Block.checkExit(rightLine, sol[2]);
                                                                break;
                                                                default:

      break;
                                                                }
                                                          }
                                                    }
                                                    //if there is at least
one exit, adds solution to solutionsLine
                                                    if (exitLine > 0) {

      solutionsLine.add(sol);
                                                    }
                                              } //if isLine=false (L shape)
                                              else {
                                                    int exitL = 0;
```

```
                                                int upL[] = {1, 1, 0,
1};
                                                int cornerL[] = {0, 0,
1, 1};
                                                int rightL[] = {1, 1, 1,
0};


                                                sol[0] = this.up;
                                                sol[1] = this;
                                                sol[2] = this.right;
                                                //for the three modules
of the possible solution
                                                for (int g = 0; g < 3;
g++) {
                                                      //if one of the
modules is a Kitchen or a Living Room
                                                      if (sol[g].typeB
!= 'R') {
                                                            switch (g)
{
                                                            //checks if
there is an exit at each position; breaks immediately after finding one
                                                            case (0):
                                                                  exitL
+= Block.checkExit(upL, sol[0]);
                                                            break;
                                                            case (1):
                                                                  exitL
+= Block.checkExit(cornerL, sol[1]);
                                                            break;
                                                            case (2):
                                                                  exitL
+= Block.checkExit(rightL, sol[2]);
                                                            break;
                                                            default:

      break;
                                                            }
                                                      }
                                                }
                                                //if there is at least
one exit, adds solution to solutionsL
                                                if (exitL > 0) {

      solutionsL.add(sol);
                                                }
                                          }
                                    }
                              }
                        }
                  }
            }
            //sets all Booleans back to true, for the new iteration
            if (blockList == Block.kitchens) {
                  kitchensAvailable = true;
            }
```

```
            if (blockList == Block.rooms) {
                  roomsAvailable = true;
            }
            if (blockList == Block.livingRooms) {
                  livingRoomsAvailable = true;
            }
      }

      //numConnections method
      public int numConnections(Block b, String positionb) {

            int connections = 0;
            //switch for direction string
            switch (positionb.toLowerCase()) {
            // b is left to this Block
            case "left":
                  //   0 1 2     0 1 2
                  // 11      3 11      3
                  // 10   b  4 10 this 4
                  // 9       5 9       5
                  //   8 7 6     8 7 6

                  //if any connection verifies
                  if (doors[11] && b.doors[3]) {
                        connections++;
                  }
                  if (doors[10] && b.doors[4]) {
                        connections++;
                  }
                  if (doors[9] && b.doors[5]) {
                        connections++;
                  }
                  if (connections > 0) {
                        //fills the pointers
                        this.left = b;
                        b.right = this;
                  }
                  break;
                  // b is right to this Block. as described above
            case "right":
                  //   0 1 2     0 1 2
                  // 11      3 11      3
                  // 10 this 4 10   b  4
                  // 9       5 9       5
                  //   8 7 6     8 7 6

                  if (doors[3] && b.doors[11]) {
                        connections++;
                  }
                  if (doors[4] && b.doors[10]) {
                        connections++;
                  }
                  if (doors[5] && b.doors[9]) {
                        connections++;
                  }
                  if (connections > 0) {
```

```
                        this.right = b;
                        b.left = this;
                }
                break;
                // b is below this Block. as described above
        case "down":
                //   0 1 2
                // 11      3
                // 10 this 4
                // 9       5
                //   8 7 6
                //   0 1 2
                // 11      3
                // 10   b  4
                // 9       5
                //   8 7 6
                if (doors[8] && b.doors[0]) {
                        connections++;
                }
                if (doors[7] && b.doors[1]) {
                        connections++;
                }
                if (doors[6] && b.doors[2]) {
                        connections++;
                }
                if (connections > 0) {
                        this.down = b;
                        b.up = this;
                }
                break;
                // b is above this Block. as described above
        case "up":
                //   0 1 2
                // 11      3
                // 10   b  4
                // 9       5
                //   8 7 6
                //   0 1 2
                // 11      3
                // 10 this 4
                // 9       5
                //   8 7 6
                if (doors[0] && b.doors[8]) {
                        connections++;
                }
                if (doors[1] && b.doors[7]) {
                        connections++;
                }
                if (doors[2] && b.doors[6]) {
                        connections++;
                }
                if (connections > 0) {
                        this.up = b;
                        b.down = this;
                }
                break;
```

```
			}
			//as self-explanatory, returns integer connections
			return connections;
		}

		//checkExit method
		public static int checkExit(int[] exits, Block current) {
			//exits[upper side, right side, lower side, left side]
			for (int i = 0; i < 4; i++) {
				//if exist[i] = 1 must verify if that side has an exit
				if (exits[i] == 1) {
					//tests doors from that side
					for (int j = i * 3; j < (i + 1) * 3; j++) {
						if (current.doors[j] == true) {
							//when it finds an exit, returns 1
							return 1;
						}
					}
				}
			}
			return 0;
		}

		//createBlock method
		public static boolean createBlock(String config) {
			//creates an auxiliary Block b and initializes it with the
constructor which receives a string as a parameter(first constructor)
			//for each line, creates a doors matrix and a description
string.
			Block b = new Block(config);
			//calls place() function for each auxiliary Block b(please read
place method)
			return b.place();
		}

		public String getDescription() {
			return description;
		}

		//removeNeighbours() method
		public void removeNeighbours() {
			//all pointers are null
			up = left = right = down = null;
		}

		public String s(boolean b){
			if(b)
				return "1";
			return "0";
		}

		public String line1(){
			String str =  "    "+ s(doors[0]) +" "+ s(doors[1]) +" "+
s(doors[2]) + "  ";
			if (left != null)
				str = str + (left.line1());
```

```
                return str;
        }

        public String line2(){
                String str =  " " + s(doors[11]) + "       " +  s(doors[3]);
                if (left != null)
                        str = str + (left.line2());
                return str;
        }

        public String line3(){
                String str =  " " + s(doors[10]) + " " + description + " " +
s(doors[4]);
                if (left != null)
                        str = str + (left.line3());
                return str;
        }

        public String line4(){
                String str =  " " + s(doors[9]) + "       " +  s(doors[5]);
                if (left != null)
                        str = str + (left.line4());
                return str;
        }

        public String line5(){
                String str =  "   "+ s(doors[8]) +" "+ s(doors[7]) +" "+
s(doors[6]) + "  ";
                if (left != null)
                        str = str + (left.line5());
                return str;
        }

        //when an array is sent to print, it will use this override method
        @Override
        public String toString(){
                String str = "";
                if(down != null)
                {
                        str = down.toString();
                }
                return line1() + "\n" +
                line2() + "\n" +
                line3() + "\n" +
                line4() + "\n" +
                line5() + "\n" + str;
        }
}
```