Minimum Cost Polygon Overlay with Rectangular Shape Stock Panels

by

Wilson Strethes Siringoringo

This thesis is submitted in partial fulfillment of the requirements of Auckland University of Technology for the degree of Master of Computer & Information Sciences.

School of Computer & Information Sciences
Auckland University of Technology

August 2006

Table of Contents

Acknowledgements

First and foremost, I would like to thank my thesis supervisor Dr. Andrew Connor, whose contributions to the research effort are simply too many to mention here. I would also like to thank Krassie Petrova, MSc., the program leader of School of Computer and Information Sciences, for her continuous support during my Master's study at AUT. For a very different reason, I must thank Professor Stephen MacDonell for his invaluable moral support. It was a small remark he made in one of his lectures some time ago that convinced me that becoming a researcher is not beyond my means.

I owe many thanks to the directors of BISCo, Ltd.: Nick Clements, MBA and Nick Alexander, MSc. for hosting my research. They have also been generous in providing technical knowledge on the house building subject which is the practical application of this study.

This research has been supported by Technology New Zealand through the Technology for Industry Fellowships scheme under grant number BISCO502 and this support is gratefully acknowledged.

Finally, I would like to take this opportunity to thank all of my family and friends whose support throughout the project have made hard times bearable and good times most enjoyable. Most of all, I would like to thank my wife Carol whose love and kindness never cease to amaze me.

Abstract

Minimum Cost Polygon Overlay (MCPO) is a unique two-dimensional optimization problem that involves the task of covering a polygon shaped area with a series of rectangular shaped panels. The challenges in solving MCPO problems are related to the interdependencies that exist among the parameters and constraints that may be applied to the solution.

This thesis examines the MCPO problem to construct a model that captures essential parameters to be solved using optimization algorithms. The purpose of the model is to make it possible that a solution for an MCPO problem can be generated automatically. A software application has been developed to provide a framework for validating the model.

The development of the software has uncovered a host of geometric operations that are required to enable optimization to take place. Many of these operations are non-trivial, demanding novel, well-constructed algorithms based on careful appreciation of the nature of the problem.

For the actual optimization task, three algorithms have been implemented: a greedy search, a Monte Carlo method, and a Genetic Algorithm. The behavior of the completed software is observed through its application on a series of test data. The results are presented to show the effectiveness of the software under various settings. This is followed by critical analysis of various findings of the research.

Conclusions are drawn to summarize lessons learned from the research. Important issues about which no satisfactory explanation exists are given as material to be studied by future research.

List of Abbreviations

2BP	Two-Dimensional Bin Packing
2SP	Two-Dimensional Strip Packing
4GT	Fourth Generation Technique
AAT	Append at Tail
BISCo	Building Integration Software Company
C & P	Cutting and Packing
CAD	Computer Aided Design
CoM	Centre of Mass
CSP	Cutting Stock Problem
DFD	Data Flow Diagram
EP	Evolutionary Programming
ES	Evolutionary Strategies
GA	Genetic Algorithm
MC	Monte Carlo
MCPO	Minimum Cost Polygon Overlay
OOP	Object Oriented Programming
RAD	Rapid Application Development
RFB	Redistribute from Beginning
RFP	Rectangular Floor Plan
SA	Simulated Annealing
SDRM	System Development Research Methodology
UI	User Interface
XML	Extensible Markup Language

1. Introduction

1.1. Project Overview

Optimizing the utilization of valuable resources has always been a premise for any successful undertaking. In manufacturing industries, the optimization of material plays an important part in minimizing the production cost, which in turn contributes to attaining a competitive edge. The importance of material optimization is especially evident in manufacturing goods consisting of large numbers of two-dimensional material components such as sheet metal or fabric material.

The causal relationship between optimized use of raw material and low production cost similarly applies to the civil construction industry as well. Various components of a building are covered with rigid sheets cut from stock material, the waste of which is either impossible or uneconomical to recycle. In many cases, the effort involved in cutting the material also contributes significantly to the cost of the resulting building.

Planning the sheet layout for a section of a building is a tedious process where exact manual calculation is either impractical or uneconomical, particularly when relatively inexpensive material is used. As a result, builders often allocate material based on loose guidelines only, incurring more cost in acquiring the material as well as consuming more manpower resources for material handling.

This purpose of this thesis is to investigate the plausibility of automating the sheet optimization process for flat sections of a building. The goal of such an automated process is to construct a solution that allows the sections to be completely covered with the optimum layout. This can be defined as the smallest possible amount of stock material, which is cut with minimum amount of effort. It is also important for these optimum solutions to be found in a reasonable amount of time. This constraint will allow the approach to be useable by the industry partner, BISCo, Ltd.

1.2. Research Objectives and Methodology

The primary objective of this research is to demonstrate the viability of automated optimization of MCPO problems. A software application is to be developed and evaluated to facilitate the investigation. In more specific terms, the objective of the research can be refined into four distinct goals:

- To develop a model of the MCPO problem into a series of parameters that can be optimized by numerical algorithms

- To develop a software application which implements MCPO automated optimization processes using general-purpose programming tools
- To demonstrate that optimization algorithms can be utilized to solve the MCPO problem effectively
- To observe the relative performance of alternative optimization algorithms

The research follows the System Development Research Methodology (SDRM) proposed by Nunamaker & Chen (1991). In this methodology, the research process takes place in progressive stages and it has been used extensively in research in information systems development. Figure 1.1 shows the stages involved in SDRM (Nunamaker & Chen, 1991).

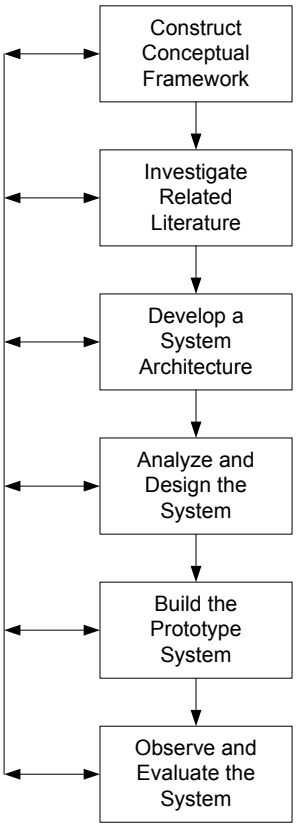


Figure 1.1: Research Process of SDRM (Nunamaker & Chen, 1991)

Each phase consists of a series of activities undertaken to achieve specific goals. Table 1.1 defines activities involved and the set goals for the phases. Because the research is explorative in nature, the completion of the phases is not expected to be perfectly sequential. While in-depth knowledge is gained through the process, repetition is anticipated as the diagram in Figure 1.1 indicates.

Phase	Description
Construct Conceptual Framework	Explore general characteristics of the MCPO problem. The purpose of this phase is to identify defining properties of the MCPO problem and subsequent research objectives.
Investigate Related Literature	In-depth review of materials belonging to the same domain as MCPO. The objective of this phase is to identify key issues, ideas, and techniques that will serve as a basis for the development of MCPO solution software.
Develop a System Architecture	Conduct in-depth analysis aimed for well-justified selection process for methodology and architecture. Design architecture of the system that exhibits favorable characteristics in terms of modularity, extendibility, and control structure. The objective is to define the system as a conceptual collection of functional modules and interrelationships that exist among them that can be realized with available tools and resources.
Analyze and Design the System	Extend the result of previous phase by elaborating on the design of modules and data structures used within the system. The product of this phase is a technical-level design that serves as a framework for the actual coding.
Build the Prototype System	Gather in-depth knowledge regarding the concepts, framework, and design issues through system building process. The objective is to gain insight about the nature of the MCPO problem, critical sub-processes involved, and the complexity of the system as a whole.
Observe and Evaluate the System	Evaluate the system by laboratory experiment. The objective is to evaluate the value of the system for resolving MCPO problems and to draw lessons from experiences learned throughout the project.

Table 1.1: SDRM Methodology

This approach defines the methodology for the underlying research. However the key phases of architecture development through to building the prototype are sufficiently flexible to accommodate various philosophies of software development which will be discussed in later chapters.

The iteration and repetition that is supported in these key phases allows early exploration of the research questions to inform selection of an appropriate software development methodology. Subsequent iterations through these phases are more development orientated and explore the specific information and functionality needs of the software.

1.3. Thesis Structure

This thesis is divided into eight chapters to describe the entire study in a logical manner. Key aspects of the research are problem modeling, software design and implementation, experiments with actual problems, and the interpretation of the results. Although those activities were often repetitive and overlapping with each other, the core idea that drives them evolved in linear pattern and chapters in this thesis are organized as such.

Chapter One provides an overview of the MCPO problem and the outline of this thesis. The purpose of this chapter is to describe the problem in general terms and provide a brief description of the forthcoming chapters. Chapter Two contains detailed analysis of the MCPO problem and the formulation of its proposed solution. Chapter Three summarizes the literature material that serve as the basis for various models and decisions made throughout the development of the solution. The theoretical aspect of the MCPO problem and the design of its solution are covered in these initial three chapters.

The succeeding two chapters deal with the software development aspect of the project. Chapter Four describes the selection of software development model based on choices available judged against the particular requirements of the project. The selection of the development model integrates the research methodology, practical software engineering considerations, and the needs of the industry partner. This chapter also contains the design of the actual software application in the form of the decomposition of the application working mechanism into its key sub-tasks as well as the identified data structures involved. Chapter Five contains technical discussions regarding the actual implementation of the software. A large number of technical issues of various levels of

implementation were encountered and resolved, the most significant of which are described in this chapter.

The remaining chapters of the thesis summarize and interpret the findings made during the execution of the software. Chapter Six reports the behavior of optimization algorithms in an isolated test environment, followed by the optimization results for the actual MCPO problems with those algorithms playing their part. Chapter Seven discusses the most significant findings that have been made during the development of the software application, along with the analysis and interpretation of the results given in Chapter Six. Finally, Chapter Eight provides the conclusion drawn from such findings and interpretation, as well as the identification of key areas where further investigation is deemed worthwhile. Appendices are provided to accommodate various supporting material that otherwise disrupts the logical flow of the thesis' main text.

2. Project Background

Optimum two-dimensional layout is a class of problems encountered in many industries. The problems are characterized with the need to pack non-overlapping shapes in an enclosed plane with the aim of minimizing the area outside the boundaries of the shapes, therefore maximizing the utilization of the material in the base sheet.

A simple presentation of the problem is shown in Figure 2.1. In this example, four shapes are arranged within the boundaries of a slanted pentagon. The objective of an optimum two-dimensional layout is to minimize the shaded area without making the shapes inside overlap. The optimization is done either by shrinking the enclosure, adding more shapes into it or repositioning the existing shapes.

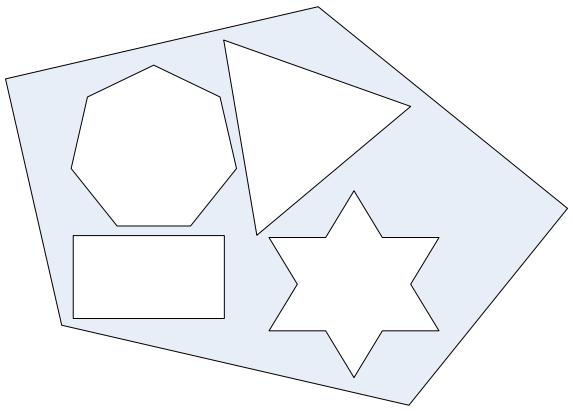


Figure 2.1: Simplified 2D Layout Problem

The actual optimum two-dimensional layout problem exists in several variants. Among them are the *sheet layout* problem, *bin packing* and *strip packing* problems, *optimum floor plan* problem, and *cutting stock* problem. These problems will be discussed in more detail in the succeeding chapter.

The optimum two-dimension layout problem has applications in a wide range of industries. Industries such as textile, timber, glass, and steelworks regularly encounter the problem of cutting the material most efficiently so as to minimize waste. In a different context, *very large scale integration* (VLSI) design requires arranging a large number of transistors and other modules in a rectangular silicon chip. The computer based solution of such problems falls under the blanket of technology referred to as Computer Aided Design and Computer Aided Manufacturing (CAD/CAM).

Although the problems found in the specific industries belong to the same class, a multitude of algorithms have been developed over the past few decades. There are two main reasons behind such a response.

- **Computing power:** the early CAD/CAM applications typically employed simple algorithms (e.g. branch-and-bound) since they had to operate under meager computing resources in terms of CPU speed and memory size. As more powerful computing platforms became available, more sophisticated algorithms were introduced and used.
- **Context-specific constraints:** on homogenous materials such as metal sheet or plain glass panel, the shapes to be contained can be mirrored and rotated to any direction. In many other cases such freedom of orientation is restricted. The material may have a face side, which makes mirroring illegal. It can also have patterns and internal fibers which limit the ranges of potential rotations.

This research is essentially an attempt to address the two points above, mainly in the identification of algorithms and constraints associated to a specific domain. Subsequently a solution is to be developed with respect to the constraints and available computing power.

2.1. Polygon Overlay with Fixed Sized Rectangles

A rather unique variant of the optimum two-dimensional layout problem is found in the construction industry. A polygon shaped area such as wall or ceiling is to be tiled with covering sheet material such as cardboard or plywood. With such tiling, it is essential that the entire surface is covered with no gaps or overlaps. The panels are obtained from the supplier in fixed size rectangles. Typically the individual panel is much smaller than the area to be covered. It is also anticipated that the enclosing area may have an irregular outline.

The problem is demonstrated in Figure 2.2. To keep the construction expenses under control, the builder must arrange the panels in a way that keeps the cost variables low. Such parameters include the number of panels allocated, the amount of discarded off cuts, and the amount of effort required for cutting the panels.

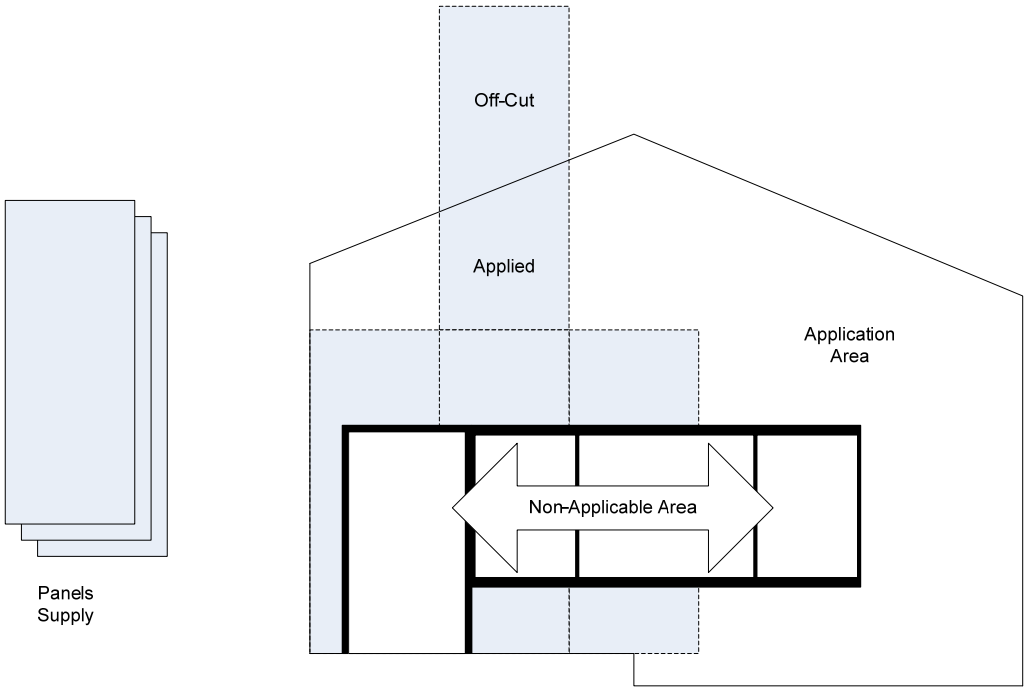


Figure 2.2: Wall Overlay with Fixed Size Panels

A similar problem has been encountered in the shipbuilding industry, particularly in cutting steel sheets to cover various parts of the ship. Adamowicz and Albano defined the problem for the operator (Adamowicz & Albano, 1976):

- A set of standard rectangular sheets of steel is provided
- An order is given to produce various types of shapes which include rectangular and irregular shapes
- It is required that no two shapes may overlap
- Waste is minimized

When the panel is homogenous, such as with sheet metal, it is desirable to reuse the off cuts to cover irregular regions at other places, as this has the potential to reduce the total number of sheets required. A particular example was made by Sibley-Punnett and Bossomaier (2001) regarding the reuse of off cuts from corrugated iron roofs. The justification for such effort is provided by the high cost of delivering the roofing material.

The diversity of materials used for constructing a building provides no guarantee that such homogeneity exists for materials used for a particular area. The implication is that the constraints for a particular section of the building cannot be predetermined. In

response, a computer program used to resolve such problem must be capable of finding the solution under a varying set of constraints to allow it to be used for any specific instance of the general problem.

2.2. Two-Stage Layout Problem

Closer examination reveals that the polygon overlay problem is composed of two sub-problems which must be resolved sequentially, although each sub-problem still belongs to the same two-dimensional layout optimization. For a given enclosed area and given dimensions of rectangular panels, the requirement is twofold:

- Find the optimum arrangement of whole panels in which the covered area within the enclosure is maximized. The by-product of this process is a set of irregular shapes which represent the remaining exposed areas.
- Resolve how such irregular shapes can be nested within the minimum number of panels. Shapes that are bigger than the panel itself are cut at angles parallel with the rectangle's axes to allow such nesting.

This decomposition into two sub-problems can potentially mask the complexity of the task of finding the optimum solution. It is important to recognize that in the construction industry, the actual size of the panels is in itself a design parameter. In some applications, the panel size will remain fixed for the two sub-problems whilst for other applications the panel size could potentially be varied. With this in mind, it becomes apparent that the problem is complex with potentially many locally optimum solutions.

At the end of the calculation process, the desired output consists of numerical and graphical information:

- The total number of panels, consisting of panels to be fitted whole and the remainder to be cut to produce the irregular shapes
- The nesting plan with which irregular shapes are cut from whole panels
- The area overlay plan with which whole panels and irregular cuts are fitted to the enclosed area

It is important to note that although the two sub-problems are similar, they are resolved with mutually unrelated and potentially conflicting objectives. As an example, the lowest cost solution for first sub-problem may be to cover as much area as possible with the least number of panels. However, the optimum solution for the second sub-problem

may be the least amount of cutting (the panel may actually be a marble or granite slab, for instance). Hence a cheap solution in the first phase may lead to expensive penalties in the second.

2.3. Motivation

Apart from reducing the waste and reducing the associated costs, automating the panel placement design also greatly assists the builder in calculating the required material. When the calculation is done by hand, the common practice is to have a human expert work on the layout and to estimate the number of panels needed to cover a particular part of the building. A few extra panels must then be provided to anticipate the error in the calculation.

As the solution only applies to a particular part of the building, the work must be repeated for all other parts as well. The process becomes more tedious when different sizes of the panels are available to choose from. Exploring more than a few different configurations by hand is therefore an impractical proposition.

Another inherent problem in MCPO problems is the lack of guarantee that an optimum solution in the first phase will lead to an optimum solution of the entire problem. Coupled with the absence of *a-priori* knowledge about the cost of the subsequent phase, exploring the less-than-optimum first-phase solutions becomes a necessity. Seen in this light, making the process automatic offers the potential of discovering better solutions than those obtained by hand calculations.

When computers are used, more possible solutions can be explored both for individual parts of the building as well as the sum of all those parts. The desired effect is that by providing the raw information to the software, in the form of a CAD model of the entire building, the builder obtains a detailed and accurate plan about the number of panels required and how they should be cut and placed for the whole structure.

2.4. Building Integration Software Company

Building Integration Software Company (BISCo) Ltd is a business enterprise whose main product is information management software for residential house construction industry. The organization has been founded by people who had identified the need for such centralized information management from their own extensive experience in the civil construction industry.

The premise of such a need is the fact that a typical house building project involves a number of different parties such as an architect, builders, city council, and others, who work in different ways and run their organizations for different goals. Although many have already adopted computer-based information systems, there is no automated means for communication with other parties. Only verbal and paper-based forms of communication are hitherto available to those parties to exchange information. It is not surprising that substantial amounts of money and effort are wasted during a project due to the lack of reliable and efficient communication system.

The core business of BISCo is developing software to accommodate such communication needs. At the time of the writing, the prototype of the software is drawing close to finish, after which it will undergo a series of live tests before it is finalized for release to the market. The software has been given a commercial name *Blue Sky*™.

Blue Sky™ manages a range of information that is very diverse in terms of representation and usage. A typical Blue Sky™ database for a house building project will include pictures and text, Computer Aided Design (CAD) drawings, letters, invoices, and a host of other documents. Such documents take various physical forms such as computer files, paper, and e-mail correspondence. Figure 2.3 shows the different parties interacting through Blue Sky™ software.

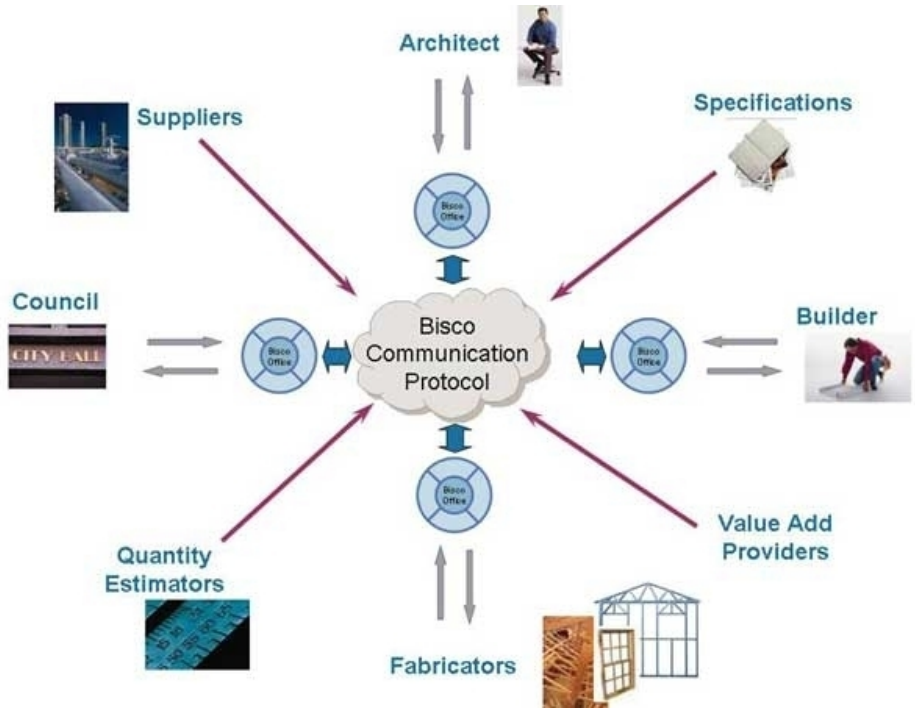


Figure 2.3: Information Exchange using Blue Sky™ Software

The geometric data stored in the CAD drawings assumes overriding importance in the house building project's web of information. Many important documents created during the lifetime of the project, such as cost estimates, specification documentation, or project plans, are actually spawned from the CAD models. Consequently, the software engineering aspect of Blue Sky™ at the current stage revolves primarily on extracting and making use of the CAD data.

For a number of reasons, accurate cost estimation has long been regarded as problematic to practice in house building projects. Firstly, it is hard for a human estimator to accurately calculate the amount of each material required to build every part of the house just from the 3D and 2D models available in the CAD drawing. Secondly, for any given building material, it is difficult to select the ideal standard dimensions from the numerous choices available in the market even when the estimated amount required is accurate. Thirdly, builders often allocate spare materials because of the difficulty in calculating the number of pieces of certain building material is needed. The spare material not only adds to the shipping and storage costs, but it may also be damaged or deteriorate on the building site and be discarded. Fourthly, inaccuracies in the design are often discovered during the physical assembly of different parts of the house. Finally when the house owner changes his mind about how certain parts of the house should be made, the alteration in many cases can only done by removing the originally-designed part when it has already been constructed – wasting even more material.

Aside from the accurate cost estimation, material optimization is another significant way to augment the enterprise's control on overall building costs. Optimization can be achieved by reconfiguring the construction of the parts of the house to minimize the wasted material. It is widely accepted within the civil construction community that up to 2% nominal worth of a project is normally wasted as scrap material. Although such percentage may look trivial at a glance, the high value of residential houses in New Zealand and other developed countries means reducing the waste by just a half will invariably save the owner a sum of money well worth the effort.

Parts of the house made of flat panels such as roof tile or drywall are prime candidates for optimization. From a financial point of view, the optimization effort is quite justified by the substantial amount of such materials needed every time a house is built. From a software development perspective, such homogenous materials are simple and readily represented as two dimensional shapes. Such characteristics make it possible to develop

and use a generic solution for varying parts of the house, with the provision that different constraints can be applied to reflect the actual materials in use.

The above analysis ultimately leads to the identification of the need to develop software capable of performing two-dimensional optimization. The solution of layout optimization problems, which is the focus of this thesis project, therefore will have an immediate commercial application at BISCo and will assist in maintaining a competitive edge through the ongoing innovation of their products. As a result, all the important requirements of the software written for this project have been identified at BISCo.

2.5. Technical Requirements

As previously discussed, the MCPO problem consists of two variants of the sheet nesting problem. Hence MCPO can be decomposed into two sub-problems, the solution of which should seek to optimize the cost involved in both.

- Design a layout of a set of stock rectangular panels which covers the container region
- Design a set of layouts where irregular remaining shapes of the original container can be fitted back into the minimum number of additional stock rectangular panels

After analysis of these requirements, it is clear that the first sub-problem is well represented by the cutting stock problem whereas the second sub-problem is the “pure” sheet nesting similar to garment sheet layout design. These will be discussed further in the next chapter.

The objective of this project is therefore to construct a computer program capable of resolving the two sub-tasks in order to solve MCPO problems. Table 2.1 lists the input and output parameters of such a program.

For technical and aesthetic reasons, some of the flat building materials, such as drywall and roofing tiles, may have directional grains or patterns. The existence of such patterns limits the possible directions of which irregular pieces can be cut from the stock panel.

Input	Output
1. The outline of the container 2. The areas within the container that should not be covered, i.e. "illegal areas" 3. The dimensions of the stock rectangular panels 4. Rules on how the irregular pieces can be oriented in the cutting template	1. The number of rectangle panels required to cover the area 2. A list of irregular shapes of where the whole panels cannot cover the container 3. A list of nesting layouts where the irregular shapes are contained by the stock panels 4. A nesting layout where whole and irregular-cut panels are fitted inside the original container

Table 2.1: Input and Output Parameters of MCPO Solution

Failure to conform to such restrictions will result in an invalid cutting solution. To accommodate this constraint, a set of rules about the possible orientation of the irregular pieces is added as a program input. This particular input will have an impact on the software design as discussed in Chapter 4.

2.6. Programming Environment

The MCPO software is to be written in Delphi/Pascal code to run on Microsoft Windows™ operating system. The Pascal-based programming language is a natural choice given the fact that other commercial products of BISCo are developed using the language. Although the executable is currently developed exclusively on a Microsoft Windows™ platform, the code is written with special provisions to allow porting to Linux operating system should the need arise.

Although there are plenty of software development tools and compilers in the market, none of them are cheap. The only development suite which offers features suitable to build the layout optimization software reliably within the allotted time is Borland Delphi™ Version 7. This particular version is used because of its availability to the author. However, since only basic features of Delphi are used, the source code should compile on earlier versions of Delphi with little or no change at all.

Apart from the availability issue, the Delphi compiler has been selected because of its full support of object oriented programming. All the important features of OOP used extensively in the development, i.e. the encapsulation, inheritance, and polymorphism, are fully supported in Delphi.

Additionally, Delphi comes with the Integrated Development Environment (IDE), which allows Rapid Application Development (discussed in Chapter 4) to be practiced

to the full extent. The concept of an IDE is not unique to Borland products however. Microsoft Visual Studio™, for instance, uses a programming user interface very similar to IDE. Microsoft Visual Studio™ cannot be used however, since it does not have a Pascal compatible compiler. The author is not aware of any more software development packages that offer a feature set comparable to that in Delphi.

3. Literature Review

The layout optimization problem consists of two main parts: the *problem definition* and its *solution algorithms*. This chapter has been written to address both issues but without specific commentary regarding the research methodologies used in each part. Invariably, research in this area utilizes a constructive methodology, even if it is not formalized, such as those proposed by Nunamaker & Chen (1991), Hevner, March, Park, & Ram (2004) or Peffers et al. (2006). Section 3.1 deals with the unique problem space which sheet layout optimization problem belongs to. The rest of the chapter discusses various search and optimization algorithms in general terms as well as with specific reference to published literature.

3.1. Sheet Layout

Considerable research has been done in various fields of two-dimensional layout optimization problems due to the practical needs of industry. Dyckhoff (1990) makes an attempt to provide a systematic classification of such optimization problems. He uses the term *cutting and packing* (C&P) as a generic name for the problem and all its variants. He further postulates the four properties of each problem which determine to which class it belongs.

Dyckhoff also asserts that there are 96 classes of C&P problems that result from the combination of the four characteristics. For the purpose of this study however, only the most important variants are considered. The significance of such variants is evident by the amount of research done and the publications that follow. The majority of such problems can be modeled in one of the four main variants: the *sheet layout*, *bin packing* and *strip packing*, *rectangular floor planning* and *cutting stock problems*. More detailed discussion about the four follows below.

Because most of the research efforts are driven by the need to solve real-life problems, they tend to focus on specific instance of the C&P class of problems. Consequently, the solutions are often very closely linked with the actual problems, leading to exotic algorithms that are potentially difficult to adopt anywhere else.

3.1.1. Basic Sheet Layout Problem

The sheet layout problem is the most generic and unrestricted form of two-dimensional layout optimization. The problem is also commonly known as *sheet nesting* and *polygon containment*. A simple definition of the problem is defined by Lamousin and Waggenspack:

> [A technique for] allocation or 'nesting' of irregular parts into arbitrary shaped resources. Placements are generated by matching complementary shapes between the unplaced parts and the remaining areas of the stock material (Lamousin & Waggenspack Jr., 1997).

Essentially, the sheet layout problem calls for cramming as many polygon-shaped pieces within a polygon-shaped container without any restrictions apart from the basic requirement that the pieces should never overlap. The pieces are allowed to rotate, translate, and to flip about any axis.

Although the generic definition allows the use of an arbitrary shaped container, in practice most problems are characterized by regular-shaped containers such as rectangular sheets (e.g. metal plates) and fixed width with infinite-length source (e.g. fabric or paper). The shape or pattern of the pieces to nest on the container may be singular or multiple. Two sample applications are discussed below.

3.1.1.1. Metal Stamping Blank Layout

Stamping is a very important technique in metal work. Pieces are engraved or cut from stock metal sheet using die blocks. The majority of everyday objects such as kitchen utensils, motor vehicles, electronic equipment, etc. contain a large number of components made by this process.

Figure 3.1 shows an example of metal stamping layout where only one blank pattern is involved. The major cost involved in the stamping process is incurred in providing the material. Therefore minimizing waste is a major goal in stamping die design.

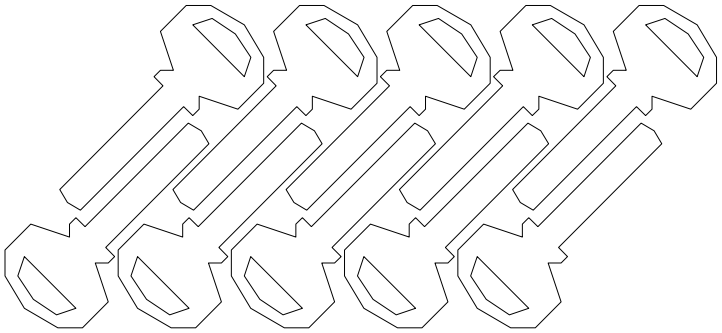


Figure 3.1: Metal Stamping Layout for a Single Pattern

Prasad (1994) describes the procedure, requirements, and constraints of metal stamping in considerable detail. In substance, blank layout design is characterized with limited

variation of shapes that are to be produced in large numbers from stock sheet. Because the die block is highly reusable, the design happens only occasionally.

An algorithm called computer aided sheet nesting system (CASNS) was also developed (Prasad & Somasundaram, 1991). The CASNS algorithm performs the search for the optimum objective value by performing incremental rotation on the pieces. Later Prasad proposes three variants of the algorithm, single-product-single-row (SPSR), single-product-multiple-row (SPMR), and multi-product-single-row (MPSR), to address the different possible requirements (Prasad, 1994).

Nye (2001) proposes a different approach that applies only to identical blanks, which he refers to as an *exact algorithm*. First he defines the objective value as a function of rotation. The algorithm calls for the rotation of the polygon and the objective value abruptly changes whenever the vertices reach certain orientations. By identifying the points where the changes take place, he effectively turns the problem into a discrete search. Linear programming is then used to find the optimum orientation (Nye, 2001).

3.1.1.2. Garment Shape Nesting Layout

In the textile industry, apparel pieces are cut from a strip of fabric which has fixed width and indefinite length. The task of a human marker is to arrange the placement of the pieces in such a way that waste is minimized. Automating the process becomes desirable as a human marker needs considerable training to acquire the necessary skills.

An example of such layout is given in Figure 3.2 below. Unique requirements on garment shape nesting are that flipping is not allowed and the fabric may have patterns which only allow rotation in a very limited range (typically up to 3º). To cope with the varying constraints, Bounsaythip propose the use of evolutionary search. Various heuristic algorithms are used to implement the solution (Bounsaythip & Maouche, 1997; Bounsaythip, Maouche, & Neus, 1995).

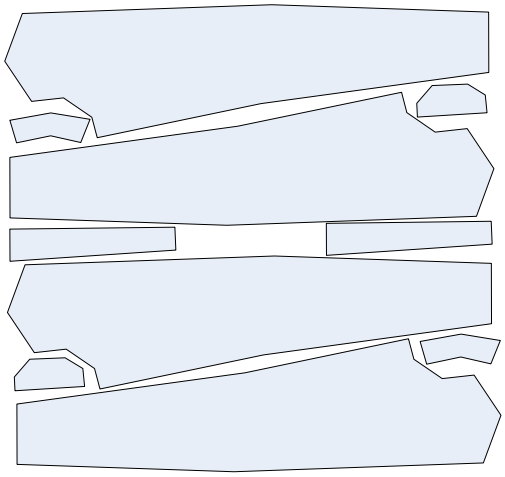


Figure 3.2: Garment Shape Layout

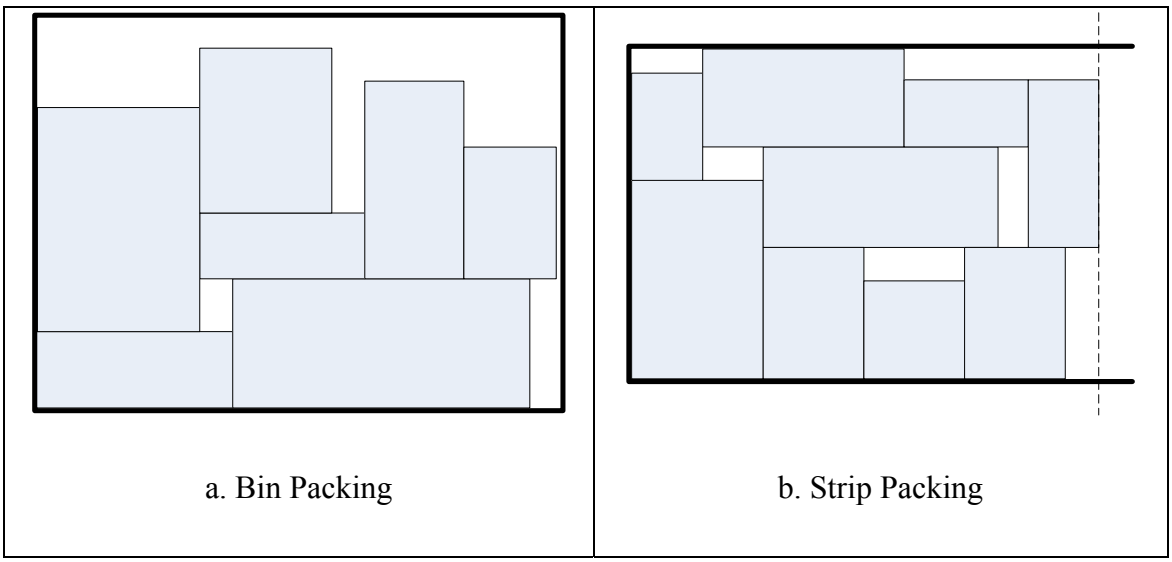
3.1.2. Bin Packing and Strip Packing

Unlike the general sheet layout problem, the objective of strip packing (SP) and bin packing (BP) is limited to placing rectangular items within a fixed width container. Furthermore, rotation is allowed only at 90º increments whereas mirroring is irrelevant because of the rectangle's symmetry. The subject of SP and BP covers problems of various dimensions. However, two-dimensional BP and SP problems can be considered a subset of the sheet layout problem class.

Lodi, A., Martello, S., & Monaci, M. (2002) define the SP and BP problems respectively :

- **Two-Dimensional Strip Packing (2SP):** for a given set of rectangles, a single bin with fixed width and unlimited height (called strip) is provided. The objective is to allocate all the items to the strip by minimizing the height of the strip used.
- **Two-Dimensional Bin Packing (2BP):** for a given set of rectangles, an unlimited number of identical rectangular bins of fixed height and width are provided. The objective is to allocate all the items to the minimum number of bins.

They further report their observation that algorithms used to solve 2SP and 2BP problems fall into three classes. They are *approximation algorithms*, *lower bounds algorithms*, and *exact algorithms*.



a. Bin Packing

b. Strip Packing

Figure 3.3: BP and SP Layouts

An example of an approximation algorithm is provided by Vassidialis (2005) who creates a model of the 2BP problem using a binary tree data structure and local search optimization methods. He argues that the tree-representation in his design is capable of capturing any configuration and translations of the problem efficiently and offers a strong base for the optimization algorithms that follow. He further specifies simulated annealing (SA) and threshold accepting (TA) to implement the local search.

Another example of a 2BP approximation algorithm is provided by Shigehiro et al (2001), which is based on tabu search. In their algorithm, various close permutations of the rectangles formation are explored to find local optima while maintaining the list of previously known optimum solutions .

An exact algorithm solution for strip packing is proposed by Hifi (1997). The 2SP problem is decomposed into a series of two-dimensional constrained cutting stock problems and a branch-and-bound procedure is used to compute the final result.

3.1.3. Rectangular Floor Plans

The rectangular floor plan (RFP) problem is a finer subset of the sheet layout problem. With RFP, the problem is limited to arranging rectangle shaped objects within a fixed size, rectangle shaped container. Therefore the RFP can be regarded as a special case of 2BP, where the objective is to put as many non-overlapping objects as possible inside a single bin.

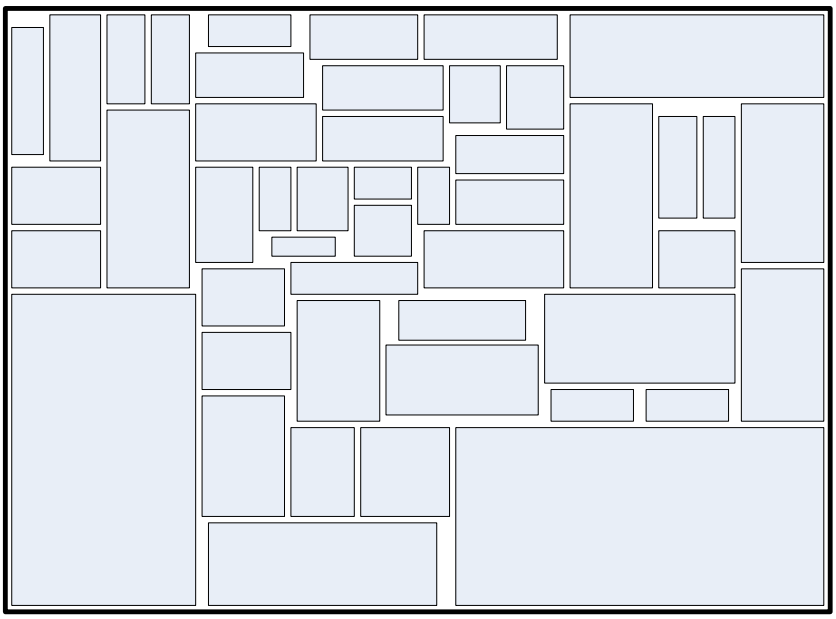


Figure 3.4: Rectangular Floor Plan

In the past, RFP has a range of applications such as in metal fabrication and publication layout (Imahori, Yagiura, & Ibaraki, 2005). However RFP later found an application in the design of very large scale integrated circuit (VLSI) chips (Hakimi, 1988; Hsu & Kubitz, 1988; Kiyota & Fujiyoshi, 2000; Murata, Fujiyoshi, Nakatake, & Kajitani, 1995). Hence despite its being a very small subset of the sheet layout problem, RFP has become an extremely important subject of research in recent years.

3.1.4. Cutting Stock Problem

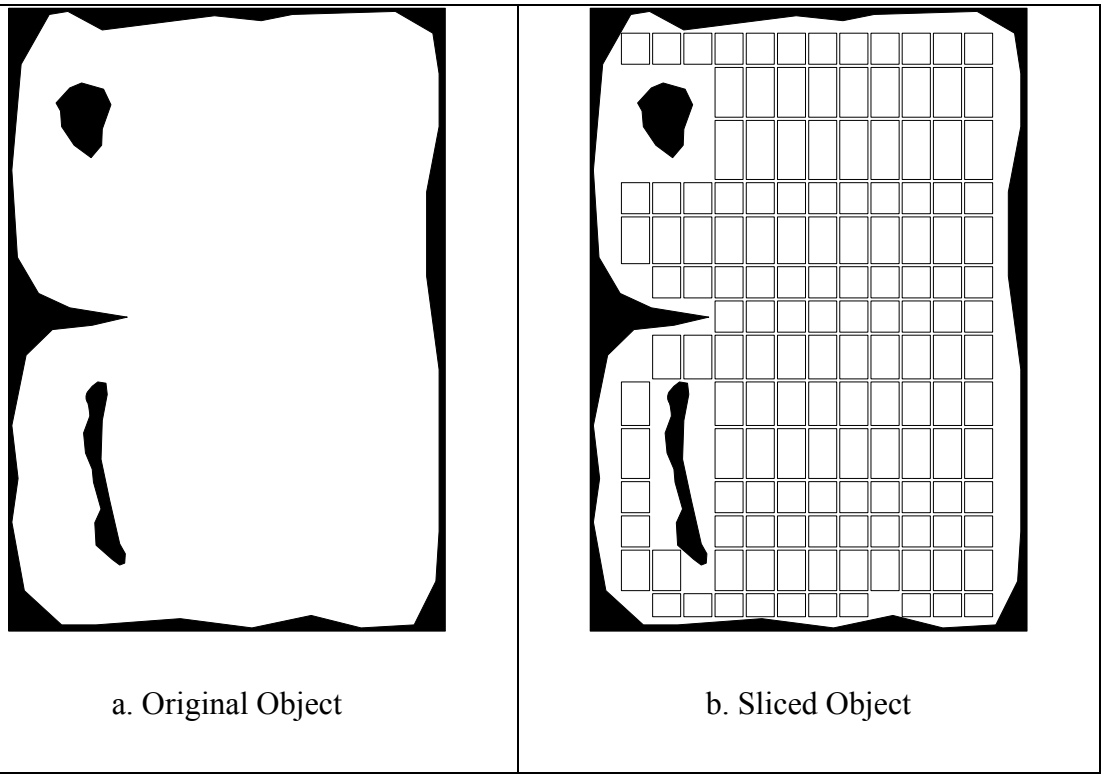
Another variant of the sheet nesting problem is the cutting stock problem (CSP). In CSP, a single stock sheet is to be cut into a series of rectangular pieces of predetermined sizes. The sizes are usually associated to values, from which the objective function of the optimization is constructed.

The CSP has a major application in the iron and steel industries. Tokuyama and Uneo (1985) define that such an application is characterized by :

- Varying criteria such as maximizing yield or increasing efficiency of production lines
- The cutting stock problem is accompanied by an optimal stock selection problem.

Similar to BP and SP, typically the stock material in CSP has a rectangular shape. In some cases however, the material can have an irregular outline as well as defective

spots in the internal area. Georgis et al (2000) define the generalized CSP and propose a solution to such problem based on the simulated annealing technique. An example of CSP on irregular shaped stock material is given in Figure 3.5.



a. Original Object

b. Sliced Object

Figure 3.5: Cutting Stock Layout

3.1.5. Summary

In the previous sections, the four main classes of layout optimization problem have been discussed. Whilst awareness of the types of problems solved to date have informed this research, the constraints on the wall layout planning problem as described in Section 2.2 are sufficiently different to make it unique. This view is supported by the limited amount of literature relating to building services applications.

The problem being solved at the first-stage is most similar to the CSP class of problem, but there is an additional constraint that coverage of the area *must* be 100%, as the wall or floor cannot be allowed to have gaps or holes. The second-stage problem is similar to the 2BP class of problems in the sense that the objective of the optimization is minimizing the number of containers used. The second-stage problem however is more complicated because it involves irregular shapes instead of the exclusively rectangular objects dealt with in 2BP. The basic sheet layout problem fits the profile better in this regard.

3.2. Layout Optimization Approaches

All classes of sheet layout problems share the characteristic requirement of putting together multiple pieces in every container. Therefore the designer of the solution is presented with the choice of whether to fit the pieces sequentially or simultaneously. The following discussion explores both options in more detail.

3.2.1. Placement Strategies

Because of the limitation of computing resources, the early solutions to sheet nesting problems are based on sequential placement of the pieces. Adamowicz and Albano implemented their algorithm in 1974 using FORTRAN on an IBM 360/67, which had 50 kB RAM (Adamowicz & Albano, 1976). About twenty years later, Daniels and Milenkovic (1995) used a SPARC computer with a CPU speed of 28 MHz. The sequential search algorithms are typically simpler to construct but easily become trapped in locally optimum solutions.

At present the typical personal computer has up to 1 GB memory and operates at 2 GHz CPU clock, providing computing power significantly greater than that in 1994. The availability of more powerful computers has made possible the approach of simultaneous placement of the nested pieces. The simultaneous placement approach allows for wider exploration within the search space, which increases the chances of finding better solutions than that obtained from sequential placement.

3.2.1.1. Sequential Placement

Sequential placement algorithms are characterized by populating the container with one piece after another. When a piece is placed on the container, an irregularly shaped smaller container is created in effect. The algorithms greedily conserve the size of the newly created restricted area when it picks subsequent pieces. The process is repeated until either the pieces are exhausted or the container is unable to accommodate more pieces.

Cheng and Atkinson list three techniques used in determining the allocation sequence of the parts in sheet layout problem. These techniques are the *Monte Carlo* technique, *random evolution*, and *heuristic sequencing* (Cheng & Atkinson, 1994).

- **Monte Carlo technique:** The entire allocation sequence is determined beforehand with a random generator. The optimum coordinates and orientation of individual pieces are then determined during the placement. At the end of the

placement process, the utilization efficiency is calculated. To obtain good results, the entire procedure is repeated with each result discarded unless it improves on the utilization efficiency of the best solution found to date.

- **Random Evolution:** This approach is generally similar to that of the Monte Carlo technique. However, only the placement sequence of the first iteration is generated as random. In the subsequent iterations, only two pieces in the sequence are interchanged while retaining the rest. The selection of pieces to interchange is also done randomly.
- **Heuristic Approach:** Instead of selecting pieces at random, with the heuristic approach the pieces are sorted according to the fitness value. Cheng and Atkinson specify that the irregular shaped pieces are first approximated with an enclosing rectangle. Such rectangles are sorted according to their sizes in descending order. The algorithm then proceeds to place the rectangles one by one. There are no subsequent iterations of the procedure. The authors claim that the technique is very efficient in terms of computation time and ease of programming, although the utilization efficiency as compared to other techniques is not mentioned (Cheng & Atkinson, 1994).

An equally important aspect of sequential placement algorithms is the optimization of coordinates and orientation of the pieces. Linear programming is perhaps the most popular approach found in the literature. With linear programming, possible coordinates and orientations are limited to discrete values only. The configuration that yields the optimum value for the subsequent objective function is then selected. Laurent and Iyengar (1982) provide an example of linear programming in use for solving nesting problem with rectangular objects.

3.2.1.2. Simultaneous Placement

With simultaneous placement, pieces are selected and placed in the container without using any sequence allocation list. Instead, other data structures such as trees and graphs are used to represent the nesting and the position of each piece relative to one another (Bounsaythip & Maouche, 1997; Bounsaythip, Maouche, & Neus, 1995). The optimization task is accomplished by finding the configuration of such structures which provides the best value for the objective function.

Typically there is a very large number of possible configurations for a given nesting problem, which makes an exhaustive search unfeasible. Several researchers argue that

the sheet nesting problem generally falls into an *NP-hard* computational complexity category. Faina (1999) concludes that the implication of being an NP-hard problem is finding the absolute optimum is not feasible when the number of items is large.

To appreciate the complexity of NP-hard problems, it is important to first understand the notion of the NP (Non-deterministic Polynomial time) problem. NP is the set of decision problems solvable in polynomial time on a non-deterministic Turing machine. The machine used to solve NP problems needs to be non-deterministic because the alternative – the deterministic machine – would attempt to find the solution through exhaustive search, which is clearly impossible in most cases since the number of evaluations increases in exponential proportion to the number of parameters. Fortnow and Homer (2002) provide historical reflection on how researchers concluded that the NP-hard class contained problems that are at least as hard as any decision problem in NP.

The computational complexity of the NP problem class is evident in the traveling salesman problem (TSP), where solving a problem with 100 nodes or more using exhaustive search requires computation time well exceeding human life time given the computing power of current technology hardware. Clearly exhaustive search cannot be suitable to NP-hard problems such as sheet layout optimization when simultaneous placement strategy is used.

Meta-heuristic algorithms offer the means of finding good solutions to such problems, although they do not guarantee the discovery of a global optimum. *Evolutionary algorithms* (EA) and *genetic algorithms* (GA) are especially popular as found in the literature (Bounsaythip & Maouche, 1997; Crispin, Clay, Taylor, Bayes, & Reedman, 2005; Horn, 2005). Other researchers prefer *simulated annealing* (SA) and *tabu search* instead (Bennell & Dowsland, 1999; Shigehiro, Koshiyama, & Masuda, 2001; Yuping, Shouwei, & Chunli, 2005). Newer meta-heuristic algorithms such as swarm intelligence (SI) and ant colony optimization (ACO) are also beginning to gain popularity (Hsieh, Lin, & Sun, 2005; Jiang, Xing, Yang, & Liang, 2004; Sun & Teng, 2002).

Given the scope of this project, it is not possible to implement and evaluate more than a small number of optimization algorithms in the MCPO software application. Three algorithms viewed to be representative for the range of available algorithms have been selected for implementation. They are the *greedy algorithm*, the *Monte Carlo* technique and the *Genetic Algorithm*. With this selection, it is anticipated that a performance

comparison can be made in terms of placement strategy (sequential in greedy algorithm against simultaneous in MC and GA), parameter manipulation approach (direct parameter handling in greedy algorithm method against indirect handling in MC and GA), and the impact of guidance (random walk in MC against guided search in GA).

3.2.2. Greedy Algorithm

Solving optimization problems typically involves the process of going through a series of steps, making a decision from a set of possible choices at each step. If the information about payoff for each choice is available, such optimization problems can be solved using relatively unsophisticated methods such as a *greedy algorithm*.

At any point, the greedy algorithm always picks a choice that gives the best reward at the moment. No consideration is given for the lesser immediate payoff alternatives, despite the potential of greater long-term reward. Because of this characteristic, greedy algorithms are simple in concept and easy to implement.

There is a weakness to this strategy of being unable to escape local optima traps. In the classic hill climbing problem, the algorithm makes its ascent by successively selecting the highest neighboring node until the peak is reached and no more climbing is possible. Obviously, this approach is prone to premature convergence if the search space happens to contain multiple local optima.

Greedy algorithms seldom find the globally optimum solution, yet in many cases they are capable of finding reasonable solutions quickly (Cormen, Leiserson, Rivest, & Stein, 2003). Because of the simplicity and speed of execution, greedy methods are quite powerful and well suited for a range of problems. Greedy methods are used in a number of important algorithms such as *minimum-spanning-tree* algorithms, *Dijkstra's single-source-shortest-path*, and for data compression using *Huffman codes* (Cormen et al 2003).

In the optimization domain, greedy algorithms may not be the best solution because they cannot reliably find better-than-average results. Nevertheless, a greedy algorithm implementation is important for this research for a number of reasons. Firstly, it provides an easy to construct platform to verify the correctness of the problem modeling. More importantly, however, it serves as the baseline solution against which the performances of more sophisticated algorithms can be measured. For industrial use, it is important to trade off solution quality and speed of convergence, and it may be that

for the building services industry that the baseline greedy algorithm may provide sufficiently good solutions in an acceptable timeframe.

3.2.3. Monte Carlo Technique

A *heuristic* approach is commonly used in optimization problem when the search space is too large for exhaustive exploration. In a heuristic algorithm, rules and methods are applied to narrow the search just to the most promising areas in the search space (Dean, Allen, & Aloimonos, 1995).

Heuristic techniques are a major subject in the field of Artificial Intelligence (AI) as AI problems are typically represented as a large search space, from which the solution is to be discovered. Heuristic techniques exist in many forms and are the key ingredient for many successful and robust AI algorithms. In the GA discussed below for example, the guidance takes the form of the three genetic operators of selection, crossover and mutation.

Despite being a good practice in general, heuristic techniques do not guarantee success in every case. They do however, offer better chances of a good result most of the time than deterministic methods. In some cases, particularly when the objective function has a discontinuous or random pattern, a blind guess may give an equal or better result than the guided search (Dean et al., 1995).

The discrete-time stochastic process called the *Markov chain* is a prime example of such a case. In a Markov chain, the past state of the system no has influence on its next state. The future state is only dependent on the current state and the *transition probability*, which is constant. Therefore even though the system is statistically stable, its exact state for a given point is completely unpredictable (Oloffson, 2005).

If unbiased dice are tossed a number of times, the resulting Markov chain will have such an erratic pattern that applying guided search will serve no purpose. In cases like this, random guessing stands an equal chance of giving a good result without any of the overhead required in heuristic decision making. Similarly, in a search space with many local optima, unguided search can come across good optimum point entirely by chance.

A *Monte Carlo method* is a blanket term used to describe any method characterized by the use of a random number generator and the complete disregard of dynamics involved in reaching the results. Weisstein (1999) defines a Monte Carlo technique in general as:

> [Monte Carlo technique is] any method which solves a problem by generating suitable random numbers and observing that fraction of the numbers obeying some property or properties. The method is useful for obtaining numerical solutions to problems which are too complicated to solve analytically.

Apart from the transition probability, which is constant, decisions at any stage are made without any restriction in a Monte Carlo method. The original Monte Carlo method was first used to create models in statistics. Later it found its use in various optimization problems.

In its most basic form, a memory-less *random walk* is all that is involved in implementing a Monte Carlo optimization method. With such unrestricted search, completely lacking in decision making rules and record keeping makes Monte Carlo optimization much simpler to implement than the heuristic algorithms.

3.2.4. Genetic Algorithm

The natural world has long been regarded the ultimate source of inspiration for design and optimization. Many sophisticated structures such as the shape of bird wings or the branching of blood vessels can be commonly found in nature, of which no man-made equivalents of comparable efficiency exist. Despite the continuing controversy about how such designs emerged in the natural world, the explanation coming from Charles Darwin's theory of evolution has been accepted in the scientific domain and firmly established itself as the foundation of modern science of biology.

Evolutionary computation and optimization were born when researchers proposed the idea of developing powerful optimization algorithms based on simulation of evolutionary process. The efforts spawned a number of algorithms, of which Bäck & Schwefel have identified three mainstream methods: the *genetic algorithm* (GA), *evolutionary programming* (EP), and *evolution strategies* (ES) (Bäck & Schwefel, 1996).

These algorithms use the concept of a population of individuals which are subject to a series of probabilistic operators such as *mutation*, *selection* and *recombination*. Each individual represents a potential solution to a given optimization problem. During the computation process, the population will undergo a draconian process in which stronger individuals will thrive while the weaker ones perish.

Genetic algorithms, which were first developed by John Holland and his colleagues at the University of Michigan (Holland, 1975), exhibit all the three main characteristics of evolutionary computation (Bäck & Schwefel, 1996). Their research goals were to rigorously explain the adaptive processes of natural systems and to design an algorithm that faithfully replicates the important mechanisms of natural systems.

In a GA, an individual is represented as a string of genes, or *chromosome*. Unlike its natural counterpart however, the genes do not manifest themselves in the physical traits of the organism. The algorithm is only interested in the gene string itself as the potential solution of the optimization problem. No mapping to physical characteristic is necessary or desired beyond that which is required to evaluate the fitness of the candidate solution.

From the optimization point of view, the chromosome serves as the representation of the coded parameters of the optimization problem. To determine how 'good' an individual is as a solution, its chromosome is decoded to retrieve the actual values, which are then fed to the objective function of the original optimization problem. The routine that decodes the gene string and calculates its objective function is called the *fitness function*, and the result of the examination is called the *fitness value*. Gene strings with better fitness values represent the stronger individuals within the population. Such individuals are favored by the system and more likely to survive and reproduce.

A genetic algorithm starts with an initial population, which will be successively replaced by newer generations until the algorithm terminates either when a sufficiently good individual is found or the number of generations has exceeded the limit set by the user. Many variants of GAs exist, but they are generally easy to recognize as they are constructed using the same following outline. If $P(t)$ denotes a population of μ individuals at generation t, and Q is a special set of individuals to be considered for selection, then the GA can be summarized as follows:

Outline of Genetic Algorithm:

```
1.  set t = 0
2.  initialize P(t)
3.  evaluate P(t)
4.    set P'(t) = recombination of P(t)
5.    set P''(t) = mutation of P'(t)
6.    evaluate P''(t)
7.    set P(t+1) = selection of (P''(t) U Q)
8.    increment t
9.  repeat steps 3 to 8 until termination condition is met
```

Goldberg (1989) asserts that GAs are more robust than many other optimization techniques, particularly when the search space contains many local optima. He further attributes the robustness of GAs to four special characteristics of the algorithm:

1. Instead of working directly with the optimization parameters, GA works with a coded set of the parameters
2. The optimization result is obtained from a population of points instead of a single point
3. GAs directly use the objective function to calculate the payoff information instead of derivatives or other auxiliary information
4. Probabilistic transition rules are used in GAs instead of deterministic rules

GAs have been applied to a wide range of problems that have been considered intractable to other approaches. The diversity of applications can be appreciated from a sample of the recent literature. A brief review of 2006 publications indicates that GAs have been applied to a huge range of problems including logic tree decision modelling (Mak, Blanning & Ho, 2006), database partitioning (Du, Alhajj & Barker, 2006), design of composite laminates (Pai et al, 2006), reliability engineering (Levitin, 2006), the design of water distribution networks (Reca & Martinez, 2006) and the classification of software failures (Watkins et al, 2006). A comprehensive review of applications of GAs is not required to discover the interest in applying this method to solving complex problems and there is considerable interest in the approach.

In many GA implementations in the literature the chromosome is commonly implemented as a finite-length binary vector. A binary vector provides the maximum flexibility for parameter coding and interpretation in much the same way as basic data types such as numerical or symbolic values are internally represented in the computer memory. Non-binary strings are also used however, in specific cases such as when

representing nodes in Traveling Salesman Problem (TSP), where a binary equivalent is impractical or inefficient (Ansari & Hou, 1997).

Because of its very flexibility, coding the optimization parameters into a gene string can be a daunting task. For any given optimization problem, there are typically a number of possible ways to code the parameters into the gene string, some are better than others. There is surprisingly little available literature providing a general guideline for coding GA parameters. Coding guidelines for specific domains do exist however, such as those proposed by Nagao for optimization of numerical parameters (Nagao, 1996).

The three basic operators in evolutionary computing, *mutation*, *selection* and *recombination*, are used in the implementation of the genetic algorithm. Specifically in the context of GAs, the operators are referred to respectively as *mutation*, *reproduction*, and *crossover* (Ansari & Hou, 1997).

3.2.4.1. Reproduction

Reproduction is the way a GA recreates new individuals in the population when the generation changes. Candidates for reproduction are selected randomly from the old population. Similar to the notion of survival of the fittest commonly observed in the natural world however, the selection of individuals is biased in favor of the stronger ones. The concept is expanded further in the form of *elitism* in some GA implementations, where chromosomes with the best fitness values in the population are favored for reproduction or even directly reintroduced to the succeeding generation (Connor, 1996). Whilst a range of elitism strategies have been discovered in the literature (Ahn & Ramakrishna, 2003; Bellomo, Naso & Turchiano, 2002; Djurisic, 1998) the use of elitism strategy at this stage has been discounted in order to investigate the performance of a simple implementation.

This mechanism allows chromosomes yielding better fitness values to stand greater chances to reproduce, in the hope of passing their good quality genes to the next generation. Less favored individuals are still kept as legal candidates despite their lower fitness values and reduced chance of being selected, in effect retaining the diversity of the chromosomes population and the search direction. Seminal work by De Jong (1975) demonstrates the effects of adjustments in GA parameters and modifications from the basic algorithm in great detail.

3.2.4.2. Crossover

The crossover operator is applied to a pair of chromosomes that have been selected for reproduction. Mating between two individuals mixes the gene strings to create a new pair of strings representing new candidate solutions. Since the two chromosomes selected for crossover are likely to have good traits, the resultant gene strings may have better features due to the recombination.

A very simple demonstration of a crossover operation is given in Figure 3.6 below. In this example, an arbitrary point, or *crossover site*, has been selected to split the parent chromosomes into left and right segments.

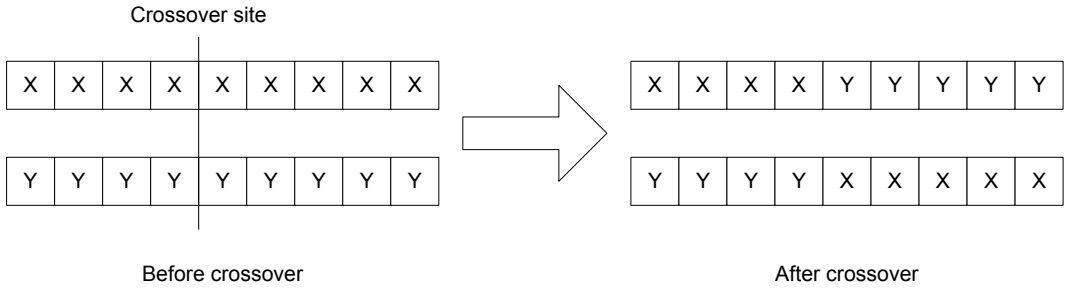


Figure 3.6: Simple Crossover Operation

In practice, any part of the chromosomes can be exchanged during the crossover. To complete the crossover operation in the above example, the rightmost segments are swapped to construct a pair of new chromosomes.

The crossover operation does not need to be limited to a single site as in the above example. More advanced versions of GAs sometimes use multiple crossover sites, as exemplified by Chang (2006) and Yoon & Moon (2002). In the interest of measuring baseline performance however, only single point crossover will be used in the GA implementation for the MCPO problem.

3.2.4.3. Mutation

The use of crossover on its own makes for a rather brittle genetic algorithm. If the parent chromosomes are identical, no new patterns will emerge in the resultant chromosomes. Similarly, no new strings will be generated when the entire population has only one type of string. The mutation operator provides a remedy to this situation.

A mutation test is applied to all genes from the chromosome of a candidate solution, normally with a very low probability of occurrence. For a positive test, the mutation

operator is applied and changes the value of the gene under consideration. This process simulates the spontaneous genetic alteration that is one of the cornerstones of the theory of evolution. In software design terms, mutation introduces variability into the population, and serves as an escape mechanism from local optima traps (Ansari & Hou, 1997).

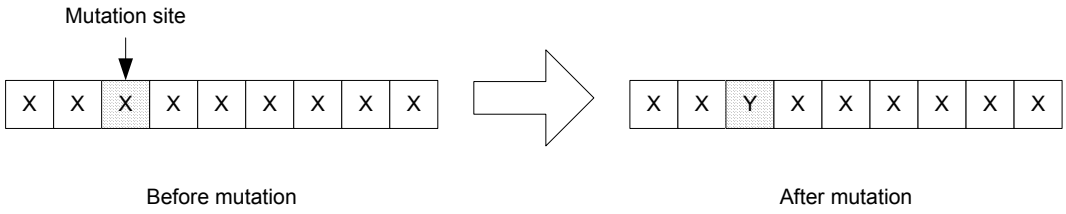


Figure 3.7: Mutation Operation

Liberal use of the mutation operator can be potentially disruptive to the search. This view is supported by De Jong who asserts that in formal terms:

> *With too high mutation rate, the performance is degraded by the sub-optimal allocation of trials to competing hyper-planes* (De Jong, 1975)

In relatively stable populations, mutation should occur only occasionally. Although there is no rule about how often mutation should be allowed to take place, successful GA implementations tend to keep its control parameter, the *mutation probability*, at low values such as 0.001 or less (De Jong, 1975).

3.2.4.4. Schemata

It has been observed that patterns of genes at certain positions have significant contribution to the fitness value of the individual. Such fixed position gene patterns are called *schemata*, which is quite an important concept in GAs (Bolc & Cytowski, 1992).

A schema is defined as a similarity template that describes a subset of chromosomes with certain similarities at certain genes. Schemata provide a basic means for analyzing the net effect of genetic operators on individuals within the population (Goldberg, 1989). Although the use of schemata is a powerful tool for creating and fine tuning sophisticated GA solutions, it is not required in a basic GA implementation.

3.2.4.5. Parameter Coding

As previously mentioned, parameter coding for GA has a major contribution towards the effectiveness of the optimization engine. A set of chromosomes containing wrong

sets of parameters or poorly mapped parameter values will ruin an otherwise good GA implementation. Similarly a good representation of the parameters will make it possible for the GA implementation to realize its full potential.

Parameter coding is especially problematic in MCPO problems, especially for the second-stage optimization, because interdependencies exist among the parameters. Referring to Section 3.1.5, the second stage optimization appears to be best modeled on the 2BP problem. Although the use of a GA in solving 2BP can be found in a number of publications such as those by Chan, Au, & Chan (2005), Falkenauer & Delchambre (1992), Lewis, Ragade, Kumar, & Biles, (2005), and Liu & Teng (1999), none provides the technical description about the actual parameter coding. Perhaps the most technical detail can be found in the work of Shian-Miin, Cheng-Yan, & Jorng-Tzong (1994) where complex tree structures are used to represent the nested objects.

In the absence of an exact description regarding the parameter coding of 2BP optimization, a novel solution for parameter coding has been devised. The complete discussion about the parameter coding is provided in Section 5.5.2. Critical analysis is provided in subsequent discussions regarding the impact of such a solution to the effectiveness of the search algorithm.

3.2.4.6. Extension of the Basic Genetic Algorithm

The basic GA can be improved in many ways. Goldberg (1989) describes a number of advanced techniques applicable to the GA. Some of the techniques are adopted from the natural world, such as *diploidy and dominance*, *elitism*, and *segregation*. Others, such as *inversion*, *translocation*, and *duplication and deletion*, are based more on mathematical reasoning.

The use of advanced techniques allows the basic GA to be either developed to deal specific problems or to be improved in efficiency (Connor, 1996). A wide range of advanced techniques have been investigated, such as the use of parallel populations (Lis, 1996), shuffling individuals between such populations (Ndiritu & Daniell, 1999), the use of "introns", which are uncoded chromosome segments (Levenick, 1991), variable population sizes (Shi et al, 2003) and the use of a hybrid method (Hwang & He, 2006) to name but a few. The improved GA in turn can be used to solve MCPO problems more effectively. Further investigation on these techniques, however, is beyond the scope of this research, and the basic GA will be used in the software application to provide results against which future enhancements can be benchmarked.

4. Software Design

As discussed in Chapter 1, this research utilizes the System Development Research Methodology (SDRM) proposed by Nunamaker & Chen (1991). Within the phases of this methodology, as outlined in Table 1.1, the selection of an appropriate design methodology is an explicit activity required to conform to the research methodology.

4.1. Design Process

4.1.1. Development Process Models

A number of software development models have emerged since the inception of electronic computers in the 40s. Software had become particularly important when punched cards were introduced in early 50s, replacing the old system where computers were 'programmed' physically by changing its electronic circuits (Tanenbaum, 2001).

Most of the software development models have been shaped by the lessons learned as various related technologies evolved. A few are now obsolete and irrelevant to the demands of today's software. On the other hand, there are still quite a few contemporary models to choose from, each has formed to address certain characteristics of the project. Pressman (2004) provides a list of the most common models currently used in practice. For the purpose of this work, it is important to select a development model that complements the research activities.

Linear Sequential Model: a classic life cycle development model borrowed from general engineering practices. The development model consists of a sequential process progressing through *analysis*, *design*, *coding*, *testing*, and *support*. This is a very sensible approach, which is the oldest and most widely used in software engineering. It is rather inflexible model however, demanding explicit and precise specification of the problem in the initial stage of the project, which is often difficult in practice. A major omission in the requirement specification, if not detected early, can be disastrous to the project.

Prototyping Model: users unaware of the issues involved in software development typically define a set of general objectives for the software they want without providing a detailed specification. Critical information such as the input, output, or human-computer interaction requirements are often left out. The prototyping model solves this problem through an iterative *specification*, *mockup build/revision*, and *customer testing* cycle. Comments given by the user during testing are used to refine the specification,

leading to a revision of the software closer to the actual solution. Although this approach can be very effective when properly used, critics point out its inherent weakness of the tendency to lure the developer to make implementation compromises to quickly get the prototype working. Inefficient algorithms may be used just to demonstrate the overall capability of the software. As the software grows, the inappropriate choices become embedded deeply in the system and become an integral part of it.

Rapid Application Development (RAD) Model: attempts to enable high speed development while maintaining the stability of the linear sequential model. The model still retains the notion of sequential development, with stages similar to that of the linear sequential model. Rapid pace of software development is achieved through extensive deployment of reusable software components. If necessary, several RAD teams can work in parallel to construct different parts of the system, minimizing the total development time for the overall project. If the scope is well defined, a fully functional system can be constructed within very short time periods using the RAD model. There are, however, a few drawbacks of the RAD model. For larger projects, significant human resources are required to allow RAD to have any impact. Further, everyone involved must be committed to the frantic pace of the development activities. Finally, not all projects are suitable for RAD, especially high performance systems whose efficiency will be compromised by the use of large numbers of software components communicating through standardized protocols (which are as a rule slower than proprietary ones).

Evolutionary Models: since the user is subject to competition or business pressure, building comprehensive software in a single development project is often unfeasible. The business and product requirements may also change over time, necessitating major updates in the software. For software that is expected to evolve over longer periods of time, a model that accommodates incremental development with minimum disruption to the overall system is required. An evolutionary model is an iterative paradigm used for the development of large systems, characterized by the emphasis on allowing the engineers to build increasingly more complete software over time while it is used in the live environment. There are a number of software development models that are considered evolutionary: the *incremental model*, the *spiral model*, the *WINWIN model*, and the *concurrent development model* (Pressman, 2004). Because of the nature of the

problem addressed by evolutionary models is unique, their advantages and weaknesses cannot be directly compared to the other development models discussed so far.

Component-Based Development: the advances in the technologies of object-oriented computing have magnified the impact of code reusability further than that in RAD. The component-based development model builds applications from reusable software components, many of which are available from third party vendors. The engineering activities are therefore more focused on mapping the functionalities required from the system with the software components suitable for the tasks. When suitable components cannot be found, custom components are engineered using the same object-oriented methods and added to the library. The *Unified Modeling Language* (UML) has been defined to facilitate efficient component-based development. Apart from the fast progress enabled by code reuse, component-based development has a pronounced advantage of allowing a scenario-based approach in software design, allowing the users to participate closely in defining the system they want. The component-based development model has its disadvantage too. Firstly, it tends to limit the developer's options to what is available in the component library. Secondly, third party components often come in binary form only, making it extremely difficult to track programming errors when they occur.

Formal Methods Model: in some cases, formal mathematical notations are the best way to rigorously specify, develop, and verify computer-based systems. Formal methods provide a way to construct correct code through the application of mathematical analysis instead of the ad-hoc review used in the mainstream models. The formal methods are very powerful and promise software that is completely free of defects. However it requires a formal mathematical ability on the part of the software developer, necessitating extensive training. Similar command in mathematics is also required from the customer if the model is to be effectively communicated. Nonetheless, the formal methods model has a secure niche in the development of safety-critical applications where software errors cannot be tolerated, such as in aviation, military, and medical equipment.

Fourth Generation Techniques: some of the latest software development tools have the capability of generating program code directly from the specification provided by the software designer. Such an approach is called *computer-aided software engineering* (CASE) or *fourth generation techniques* (4GT). With 4GT, certain specification

graphics and languages are used to define the problem, resulting in meta-code that can be translated to the actual program code using the development tools. Apart from the automatically generated code, 4GT differs little from other models discussed above. The distinct advantage in the 4GT approach is the reduced time required for design and analysis, particularly for small application. It also allows the technically inclined users to develop a credible solution of their own directly. The potential productivity boost comes at a price however. The ease with which code is generated may lead the developers into neglecting the importance of a good design, resulting in poor quality software. Another problem inherent in automatic code generation is that the process is unidirectional. As a result, manual modifications on the code will not be incorporated back to the design. Worse yet, regenerating the program will overwrite the modified code, effectively wiping out all the manual changes.

Agile Methods: in the interest of satisfying the customer's demands on the software at a very rapid pace, a relatively new approach called *agile methods* have emerged in the last few years. The agile methods refer to a number of software development practices that put heavy emphasis on progress in the form of working software. Such progress is achieved by developing the system in a series of development mini-periods, each lasting between one to four weeks. At the end of each period, the resulting software is evaluated, establishing the base upon which the user requirement is further refined or possibly expanded. The agile methods rely on intensive, face-to-face communication between customer and developer, resulting in little formal specification and documentation compared to other models. While potentially very effective, agile methods may lead to undisciplined or chaotic development activities, limiting its value to small sized and highly skilled development teams only.

4.1.2. Project Characteristics

All the software development models discussed above offer enough potential to be considered to select one that suits best for this project. With the key characteristics of the models identified, the selection of which one to use in this project is to be decided by analysis of the characteristics of the problem.

The basic premise is that the layout optimization software is first and foremost a research project. The project scope and size is deliberately limited to that required to examine the concept and algorithms involved in the problem space. Therefore the project should be regarded as a short term software development, with limited goals

only. There is a strong prospect for longer term deployment of the commercial version however, which calls for sound design and easy to maintain code.

As evident in Chapter 2, there is no detailed specification provided by the user of the system, which is not surprising given the research oriented nature of the development. What is available is a general description of what the system is going to be used for and what it should be capable of doing. Whilst such information is enough to provide a general direction for the development, much of the technical details must be discovered as part of the research and development activities. The absence of detailed specification favors the prototyping and RAD models whilst ruling out the use of linear sequential model.

There is also a definite constraint in the development timeline. A fully working system is required within no more twelve months from the beginning of the project. While such restriction is quite reasonable for the scope of the project, successful completion requires focus on critical parts of the software that are developed to address specific aspects of the research. Non-critical parts are implemented using rudimentary algorithms since little time is available to explore or implement more sophisticated alternatives. Without the need of building a large system over prolonged periods of time, evolutionary models automatically become irrelevant.

Although quite a number of people are involved in the project, the actual development of the software is done by a single person. Consequently the resultant amount of work is extremely limited compared to what a team of developers can achieve.

Due to the lack of mathematical background on the part of the author, formal methods cannot be used with any degree confidence given the time available. Similarly, there is no suitable 4GL tool at the author's disposal during the project. All these constraints leave the prototyping, RAD, and component-based models to be the only viable choices. The actual development model uses features of all three. Prevailing development effort takes mainly the form of RAD because the requirements are best mapped to this model.

4.1.3. Rapid Application Development Model

As previously discussed, the RAD model dominates the actual development pattern of the layout optimization software. Key characteristics of RAD are modular design, parallel development, and code reuse; all resulting in development of good quality software at a rapid pace. Other deciding factors are the availability of a development

tool and the ease of future integration to the BISCo commercial software. Finally, RAD allows development to begin in the absence of a comprehensive system requirement, making it ideal in a research environment.

The effectiveness of RAD comes from the way it is designed to decompose the problem into its logical components. This allows development of some components of the software to commence whilst research activities are still being conducted to determine the requirements for other aspects of the software. The development process is also divided into several phases. Much like the linear sequential model, the RAD cycle in a commercial project consists of five phases: *business modeling*, *data modeling*, *process modeling*, *application generation*, and *testing and turnover* (Pressman, 2004).

Business modeling: the way information flows within the customer's organization is analyzed and modeled in terms of data architecture, application architecture, and the technology infrastructure. Since the layout optimization problem is to be developed much in isolation, analysis is focused only on data and application architecture. Rigorous analysis on future customer's technology infrastructure will have little impact on the construction of the software.

Data modeling: the information flow is further refined by identifying all the important data objects that the system should maintain. The data objects are defined by the attributes and relationships that exist with other data objects.

Process modeling: another refinement of the information flow is when relevant processes that transform the data objects are defined for implementation. Important functionalities such as data object creation, alteration, calculation and removal are specified in sufficient detail for the *application generation* phase that follows.

Application generation: ideally, the implementation phase uses a 4GT tool to generate the code. Third generation programming languages are used when a suitable 4GL is not available. Existing program components are used whenever possible. New components are created when necessary, with reusability as one of the main design goals.

Testing and turnover: the final product is tested and updated as necessary. The true value of RAD should be apparent at this stage. Since the majority of the reusable components have been proven to work in other applications before, few errors are expected to occur from them. The net result is reduced overall testing time. Care must

be taken, however, to thoroughly test new components to ensure their reliability in future projects.

The remainder of this chapter deals with the business and data modeling, along with the abstract part of process modeling. The RAD cycles are discussed in Chapter 5: Software Implementation.

4.2. System Modeling

4.2.1. Software Scope

Before analyzing the system in more detail, it is a good practice to describe the scope in a brief statement. Such a statement serves as the basis for communication among parties involved in the project, especially between the engineer and the customer. In the case of this layout optimization project, the customer is represented by BISCo. The layout optimization scope statement is as follows:

> Layout Optimization Software searches for the most efficient configuration to cover an area using flat rectangular panels of fixed length and width. The layout optimization problem consists of the layout area(s), the invalid areas that should not be covered, and the list of candidate stock panels.
>
> For each optimization run, the user provides additional information such as the panel legal orientations, search strategy, and optimization algorithm-specific parameters.
>
> The output of the optimization consists of the dimensions of the most efficient stock panel, the cutout plan for individual panels, and the plan to cover the layout area. Efficiency is primarily calculated in terms of the wasted material, although the amount of cutting also determines the quality of the solution. The less efficient solutions are also provided for comparison. The results must be sufficient allow the user to select the most efficient panel, cut the irregular shapes from the stock panel if necessary, and arrange them on the layout area with no manual calculation.

The purpose of the above statement is to capture all the essential parts of the system. Further technical details are not relevant at this stage and will be defined at the subsequent phases.

4.2.2. Information Flow

In well-functioning software, information is transformed from its raw form into its final, useful representation. To afford credibility for such transformation, it is important to define the components taking part and the various pieces of information that undergo the process. Data computation is seen as *information flow* from input of various forms, which ends in the output forms, through a series of transitional forms.

There are various ways of modeling the information transformation. The *Data Flow Diagram* (DFD) is the prime modeling tool to use when the designer is interested in decomposing the system based on its functional components. *Entity-Relationship* (ER) models can be used instead when relationships within the data are regarded to have overriding importance. Otherwise when the system is defined by its time-dependent behavior, *State Transition* modeling is the most suitable tool (Yourdon, 1989).

In the case of the layout optimization software, the DFD model is used for a number of reasons. First, the system must be built when no formal or standardized representation of the data exists. A key element of this research is the exploration of the information required for the solution of the MCPO problem and how the software interacts with this data. The functionality of the software, namely discovering the solution for a given optimization problem, therefore dominates all other issues. Second, the data does not dictate the behavior of the system at all. Instead, there is a rigid mechanism to which the information is subjected. These two reasons rule out the use of entity-relationship and state transition models. Finally, the DFD model actually serves as an indispensable tool for defining what components are required to build the entire system, as reflected in the discussions that follow.

The ideas similar to DFD have been circulated in the engineering communities since the mid-seventies (Yourdon, 2006). The DFD owes much of its appeal to the simplicity of its notation and representation. Such simplicity makes DFD easy and intuitive to use, making it ideal to communicate design ideas among designers and users alike.

A data flow diagram consists of four components, of which three are static and one dynamic:

> **External Entities:** objects and actors that reside outside the system and interact with it. The most important external entities are the input device, output device, and the user. External entities are represented in the diagram by rectangles.

Processes: the transition in which the input data is turned into output. A process can be directly mapped to a functional module of the software, serving as a black box where only input and output types of information are defined without revealing the actual mechanism of the transformation. A process is only identified by its functionality, and is represented by a circle in the DFD.

Data Store: repository for where non volatile data is kept. Theoretically the storage has no restrictions in terms of size or lifetime. A data store is represented by parallel lines in the diagram.

Data Flow: the dynamic component which describes the movement of packets of information moving from one component of the system to another component. The flow represents the data in motion, as opposed to the data store which represents the data at rest. Data flow is represented in the diagram as an arrow coming from or into a process.

Another major feature of DFD is the freedom for the designer to zoom in on a particular process to reveal its inner working as a mini-system. When a process is analyzed, the resulting DFD consists of the components similar to the DFD of the higher levels, but with more refined processes and data flow. Due to this hierarchy, any part of the system can be analyzed to any level of detail. The decision on how much analysis is required is left to the designers and perhaps the user.

Figure 4.1 shows the information flow at the top level of the layout optimization software. The diagram of this type is often referred to as *DFD Level 0*, for it shows the system components and the information flow at the most abstract level. The DFD shows that nine processes are involved in transforming raw data from the input device to the final form presented to the user.

The nine processes are the key to successful optimization operation from one end of the system to the other. Further decomposition of these processes into their respective sub-processes and information flows can be found in Appendix B. The remainder of this section discusses all the components found in DFD Level 0.

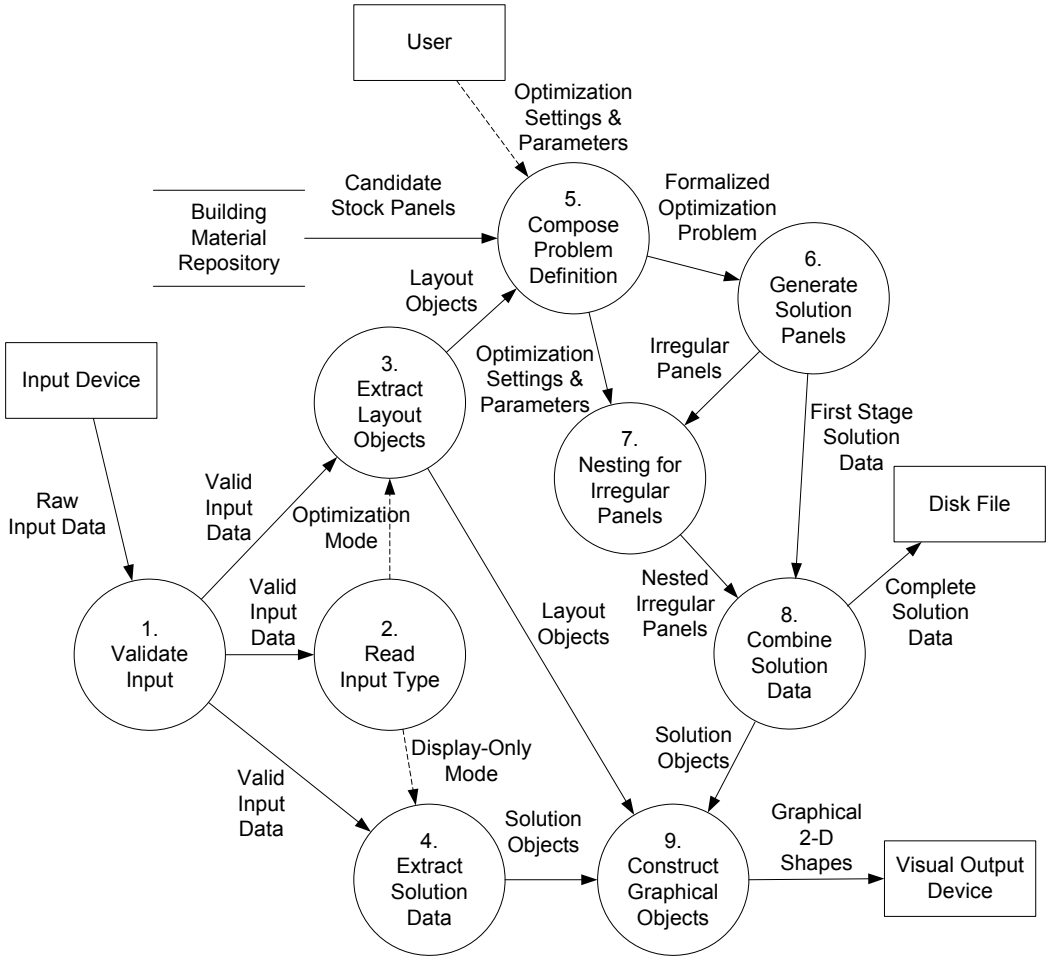


Figure 4.1: Overall System Information Flow

4.2.2.1. External Entities

There are four external entities used in the DFD: the *input device*, *user*, *disk file*, and visual *output device*.

Input Device supplies the problem definition to the software. Disk files are excellent candidates for the input device, although other forms of input such as *remote procedure calls* (RPC) and onboard editors may also be used. RPC may become the primary input device in the future, especially when the software is integrated into the Blue Sky™ system.

User interacts with the software in many ways. The most important role of the user from the design perspective, however, is to determine the actions to be taken for a given optimization problem as well as specifying the various parameters required by the optimization algorithms.

Disk File can be considered a virtual output device, where the results of the optimization can be stored for future use. The disk file is not a true output device however, since the information is only useful within the system's domain.

Visual Output Device presents the solution of the optimization problem in its final form to the user. Because the layout optimization deals with two-dimensional objects, the main use of an output device is for viewing the objects. Thus only graphics capable devices can be used. In most cases the computer screen can satisfactorily serve as the output device, although a printer may be preferred by the user in other cases.

4.2.2.2. Processes

As mentioned earlier, there are nine processes that make up the system at the conceptual level. Although these processes vary in size and complexity, none can be omitted from a fully working system. The data flow diagram for the following can be found in Appendix B, from Figure B.2 to Figure B.10, respectively.

Validate Input ensures that the input data is of correct physical format and is organized in a valid structure. As discussed in section 5.2.2., Extensible Markup Language (XML) has been chosen as the physical format of the input data. The validating measure is very simple: the input stream is checked whether it has an XML document, in which case the root node in the XML hierarchy must have a certain name to be accepted as valid. Illegal input will immediately cause the whole loading process to be terminated.

Read Input Type seeks to find whether the XML input file contains a *problem set* or a *solution set* data by examining a certain attribute of the root node. The program enters the *optimization mode* when a problem set input is encountered, whereas a solution set will cause the program to enter the *display-only mode*.

Extract Layout Objects extricates the layout *containers* and the *obstacles* within them from the input data. These objects are very significant in a layout optimization problem, and they are both present in either type of input. The containers and obstacles are used to define the problem in optimization mode, which makes them critically important.

Extract Solution Data reads the information about the shapes covering the container legal area as well as their nesting plan in the stock panel. Such information is used to present the solution visually to the user. Naturally, this operation is only relevant when the input is of solution set type.

Compose Problem Definition packs the data required for the actual optimization process. Layout optimization data loaded from the input device is combined with the supporting data in the repository to make the complete problem definition. Finally, the algorithm-specific optimization parameters are supplied from the user input.

Generate Solution Panels resolves the first stage of the optimization problem as discussed in Chapter 2. The solution is constructed by cutting the layout container area with horizontal and vertical lines as if it were a large sheet of material. The cut lines are defined by the dimensions of the corresponding stock panel. The resulting shapes are then deducted by the obstacle shapes within the container to find the final set.

Nesting for Irregular Panels resolves the second stage of the optimization problem. The irregular shapes from the solution set of the first stage are mapped to the smallest possible number of stock panels. Three search strategies are available to find the solution: the *greedy algorithm*, the *Monte Carlo search*, and *Genetic Algorithm search*. The result of this process is the irregular shape nesting layout plans, according to which the actual material will be cut.

Combine Solution Data arranges the bits of information obtained from both stages of the optimization into an organized form, ready to either display on the visual output device or save into the disk file. Checks are also made to detect and remove redundancies as well as inconsistencies.

Construct Graphical Objects extracts the visual information from the solution and transforms it to the standard format of the output device. Section 5.3.3 in Chapter 5 discusses the concept of *graphics pipeline* which makes this process necessary. In simplest terms, the optimization problem and its solution use a coordinate system that differs to that in visualization. The process of constructing graphical objects takes care of the transformation between the coordinate systems as well as adding visual properties that helps the user to differentiate between objects in his viewing device.

4.2.2.3. Data Store and Data Flow

There is only one data store object used in the system, that of the Building Material Repository. In contrast, there are thirteen data flows defined in the DFD Level 0. The large number of the data flows very well demonstrates that information does flow in many forms through the system before the final result can be obtained.

Building Material Repository is where static, problem-independent data is stored. In the commercial context, the repository takes the form of the Blue Sky™ database. Referring to section 2.4 in Chapter 2, the database contains a large collection of disparate objects. In the context of layout optimization however, only the stock panel information – which makes for only a small subset of the data – is used. In the research context, the repository is implemented as a disk file of much simpler structure. The

decision to keep the repository separate from the main input remains justified, however, by the static nature of the stock panel information.

Raw Input Data presents the system with an input stream. At this stage the system does not know whether the input is valid or whether it contains meaningful data. All the same, the only way a valid input can enter the system is through this data stream.

Valid Input Data is the input data that has been verified and accepted as valid, in terms of physical format and the content organization. A valid input data may contain either a problem set or a solution set. Different processes are involved to handle each type.

Optimization Mode and Display-Only Mode is a *control flow*, a special case of information flow, where only fixed value information is passed. For a given input data, either optimization mode or display-only mode control is passed to the responsible processes depending on whether the input is of problem set or solution set type, respectively.

Layout Objects are the containers with the optional obstacles that define a layout optimization problem. These objects have paramount importance since they are actually the basic representation of the optimization problem itself.

Candidate Stock Panels: provides the context for resolving the layout optimization problem. Any solution must be built from candidate stock panels that are known to the system. Also the efficiency of a solution is to an extent determined by the properties of its candidate stock panel, such as the material price and cost associated with cutting the irregular shapes.

Optimization Settings and Parameters is a control flow that allows the user to manage which optimization algorithm is to be used, and how a given algorithm should operate. More detailed information about optimization settings and parameters can be found later in section 4.2.3.3.

Formalized Optimization Problem is a data structure that consists of the containers, obstacles, and the candidate stock panels. The data structure emerges as the final result of the all data preprocessing and is ready to be fed to the actual optimization algorithms.

Irregular Panels are a subset of the result of the first stage solution. The rest are regular panels that map perfectly to the stock panels and therefore need no further processing. Irregular panels make for the material for second stage processing, in which the search for most efficient nesting plans takes place.

First Stage Solution consists of both irregular panels and the regular panels, which are a part of the final product of the layout optimization. Only minor post-processing, such as redundant vertices removal, will be applied prior to presentation to the user.

Nested Irregular Panels is the result of the second stage optimization. Similar to the first stage solution, the nested irregular panels are also part of the product of the layout optimization.

Solution Objects are the first and second stage solutions combined. The solution objects reside in the computer memory, ready to be translated into its visual representation.

Complete Solution Data is the solution objects organized as an XML tree. The purpose of creating such representation of the solution is for storage in the disk file, from which it can be loaded to the software later for viewing.

Graphical 2-D Shapes are the data representation of the objects to be visualized. Such representation is platform-dependent or device-dependent. Unlike the rest of the system, the designer has no control over the format of the data since it is dictated by the particular output device in use.

4.2.3. Data Dictionary

Another important design tool used to develop the layout optimization software is the *data dictionary*. Yourdon (1989) defines the data dictionary as a listing of the data and control objects used in the system. In the data dictionary, such objects are defined in much greater precision than the equivalents found in the data flow diagram. The combination of the DFD and the data dictionary provides the analyst with a highly accurate view of the system (Pressman, 2004). Yourdon (1989) further specifies that the data dictionary complements the DFD in a number of ways:

- The data dictionary describes the meaning of the flows and stores in the data flow diagrams
- The data dictionary describes the composition of aggregate data packets in the flows and in the stores
- The data dictionary specifies the relevant values and units of the elementary parts of the data objects

An entry in the dictionary consists of the *header* and the *definition*. The header contains the *name* of the item, its *aliases*, and the context of where or how the item is used. The

data dictionary uses metadata to describe various operators such as *definition*, *aggregation*, *iteration*, and *selection* in the definition part. The following symbols are most commonly used as recommended by Yourdon (2006).

```
= is composed of
+ and
( ) optional (may be present or absent)
{ } iteration
[ ] select one of several alternative choices
** comment
@ identifier (key field) for a store
| separates alternative choices in the [ ] construct
```

Yourdon (2006) warns that building the data dictionary can be tedious for medium or large-sized systems. Modern relational database management systems such as DB2, Oracle, or Sybase come equipped with automated tools for defining the data dictionary. In this project however, such tools are not available and the data dictionary must be constructed by hand. For this reason, only certain items of the dictionary are defined. The complete data dictionary can be found in Appendix C.

4.2.3.1. Input Data

As previously discussed, the input data contains various control information to describe its content. The document body follows, consisting of parts common to all input data and optionally the solution information if the document is of solution set type. The common part is further decomposed into various components that define the problem as well as the viewing parameters.

name:	**raw input data**
aliases:	complete solution data
where used/ how used:	layout problem definition (input) resolved layout problem data (input)
description:	
	raw input data = signature + document body signature = "PolyWorkSpace" + content type content type = ["ProblemSet" | "SolutionSet"] document body = generic part (+ solution set) generic part = view parameters + container set + obstacle set view parameters = zoom factor + viewing offset zoom factor = *real number > 0* viewing offset = XY screen coordinates

Figure 4.2: Program Main Input

4.2.3.2. Optimization Result and Output Data

Discussion about the output data in non-graphical form actually becomes meaningless since the optimization result is reduced into a series of platform-dependent geometrical information to produce the visual output. The complete solution data saved in the disk

file does not reflect the result very well either since it contains aggregate information that can obscure the solution. The optimization result is therefore best represented by the *solution set*, which in turn is composed of a series of single solutions. A single solution defines the stock panel used, along with the first and second stage solutions.

name:	**solution set**
aliases:	None
where used/ how used:	resolved layout problem data (input & output)
description:	
	solution set = {single solution} single solution = stock panel + solution panels + nested layouts stock panel = rectangle definition

Figure 4.3: Complete Layout Optimization Solution

4.3. Design Issues

The data flow diagram and the data dictionary have served their purpose well in examining the system, helping the analyst to identify all the major issues. Speaking in terms of rapid application development, *data modeling* and *process modeling* have been sufficiently covered using those tools. Because *business modeling* has little relevance to the engineering-oriented project, software development can now proceed to the *code generation* stage, where the program is implemented. The technical discussion about the actual implementation can be found in Chapter 5. There are a few remaining design issues however that must be resolved at this point.

It is important to recognize that considerable effort has been expended to achieve thorough analysis, design, and implementation of the software. These activities have been conducted in parallel with the research activities that have informed the refinements of the requirements. Nevertheless it must be kept in mind that the ultimate goals of the research are as defined in Chapter 1. For this reason, a few design compromises have to be made to prevent minor issues from detracting the progress from its true objectives.

4.3.1. Control Hierarchy

The data flow diagram and data dictionary are very good at capturing the data transformation from the raw form to its final representation. In other words, they have provided a data-centric view of the software. The actual program, however, ought to be a task-centric system if the user is to have the ultimate control.

The *control hierarchy* or *program structure* represents the organization of the functional modules to reflect the control relationship. Like any hierarchical relationships, the control hierarchy is best represented as a tree-like diagram.

The layout optimization software uses a control hierarchy shown in Figure 4.4. The diagram consists of a set of rectangular shapes representing the software modules, and connecting lines representing the superordinate-subordinate relationship. The modules are either of control type or functional type. A control type is represented by a gray box whereas a functional type is represented by a white box. A controlling module (called *superordinate*) is drawn higher than the controlled modules (called *subordinate*) connected to it.

A module called *nesting work space* occupies the top of the control hierarchy. The user interacts with the module through the user interface feature of the program. At the second level, the control is partitioned into three separate sub-trees: the input modules, data transformation modules, and the output modules. Each controls a set of functional modules either directly or indirectly through its subordinate control modules.

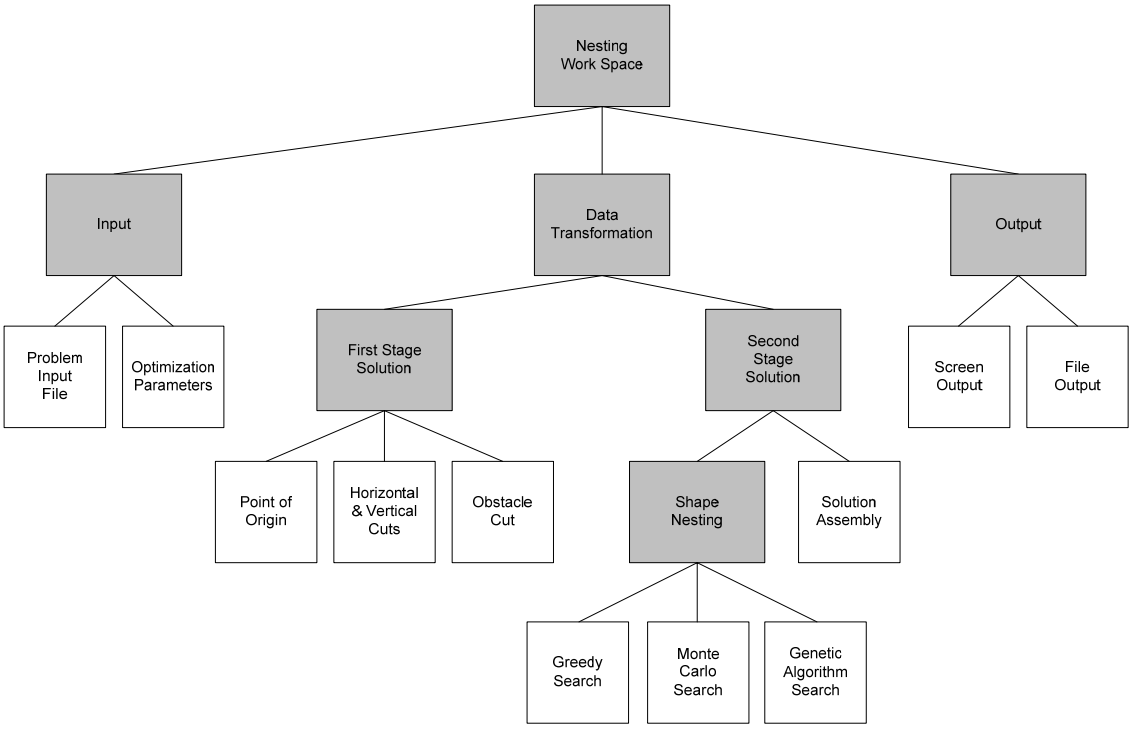


Figure 4.4: Layout Optimization Software Control Hierarchy

Most of the functional modules in the control hierarchy can be directly mapped to the process modules in the data flow diagram, indicating that the system analysis has been sound. Unlike in bigger systems where the control hierarchy can be quite elaborate, the

simple diagram in Figure 4.4 sufficiently models the control structure of the layout optimization software. Therefore no further analysis is necessary.

4.3.2. Program Input and Output

Substantial discussion has been made regarding various issues about the input and output of the layout optimization software. It remains difficult to settle for a final form of either however, because the software is destined to become part of a large system yet to exist. Nevertheless interim formats have been established for the purpose of this research.

For program input, the physical source is a text-based disk file structured in XML format. The content of the file is to be created and edited by hand, even though in the future it is anticipated that the input will be generated by an automated process instead. Little provision beyond a few simple checks is to be made for validating the integrity of the input data.

The program output has been designed with similar orientation. The computer screen, run by a specific operating system, is to be used as the main output device. The disk file is also to be used as secondary output device, with the data organized in XML format recognizable by the program input module.

4.3.3. User Interface and Visualization

The primary requirement for the program user interface (UI) is that it provides the user with full access to the commanding *nesting work space* module in the control hierarchy. While the simplest solution may be to build the module directly in the UI or vice versa, it has been decided to build them separately to make it possible to change the UI without affecting the control hierarchy.

The separation is particularly important as the UI has been designed as a research tool and not to be particularly effective with the average user. In future development, software usability will be taken into consideration when designing the UI. For the purpose of this research, the functionalities of the UI are limited only to that of providing access to the system features. The following is a list of the main user activities facilitated by the UI:

- Open input file and load the content into the computer memory
- Save the geometric objects in the computer memory into output file
- Select optimization algorithm and set its parameters

- Execute optimization process
- Monitor the software activities during the lengthy execution of important tasks

There is a set of optimization parameters that deserves special attention at this point. These are the constraints that specify what orientations are allowed when the irregular pieces are laid out in the nesting plan, e.g. *flipping* and *rotation*. The way such constraints are applied may have a direct impact on the final result, which brings up the question of who must decide what particular order to use on each run. It is the end user, with domain knowledge regarding the materials in use, that must specify how such constraints will be applied.

For a start, it seems sensible to implement the corresponding modules in a way that allows the parameters to be used in any sequence, rather than implicitly assuming a particular order in the program code. Such customizable parameter ordering is rather costly to implement, but it eliminates speculation on the part of the developer and leaves the decision to the user instead. Therefore it deserves to be a standard feature of the software. Subsequently the UI must provide a way for the user to supply his custom constraint application to the system.

The UI also serves as the platform for visualization of the layout optimization problem. Sufficient space is to be provided within the program UI on the computer screen for drawing the geometric shapes. Standard viewing features such as zooming and panning are also to be provided in the visualization space.

Because of the importance of the visual information to the user, additional measures are also to be taken to assist the user to absorb the information efficiently. Such measures include the use of colors and patterns to help the user identify different type of objects more easily. It may be a good idea to make the colors and patterns customizable by the user. At the current stage however, such customization is not a priority.

5. Software Implementation

5.1. Overview

The implementation of the MCPO concept provides the means of obtaining detailed insight regarding the nature of the problem in technical terms. It is during the actual development process that the basic knowledge previously acquired from the literature research is consolidated and enriched. Comparatively large amounts of time and other resources have been allocated to this effort due to its overwhelming importance.

Throughout the development of the software, it was found that many of the functionalities required by the application present unique logical problems demanding thoughtful and well designed solutions. It was also found that certain kinds of such problems tend to reappear at various places, although often in slightly different guises. Such recurring classes of problems merit discussion of their own due to their contribution to the overall optimization problem.

The development follows a RAD pattern characterized by the absence of a predefined set of requirements. Instead, the software starts as a crude prototype which quickly evolves into an increasingly refined product in a continuous coding-evaluation-improvement cycle.

Program modularity is regarded as very important, compelling the author to strive towards a highly modular code in spite of the lack of formal requirements and design and the foresight they may offer. Object Oriented Programming (OOP) with its inherent features such as encapsulation and polymorphism provides an excellent framework for modular application development under such circumstances. This is made possible in particular by Borland Delphi™ compiler which has full support of OOP. The *Object Pascal* language supported by Delphi allows custom classes and objects to be used extensively in the program to achieve the desired level of modularity.

5.2. Program Structure

5.2.1. Modules

The primary goal of writing modular software is to make coding and maintaining all parts of the program relatively easy, no matter how divergent those parts may be. Modular software design also facilitates code reuse, which is another key ingredient for rapid development of good quality software. In Delphi as the programming language, software modularity can be achieved in three different ways:

- **Object modularity.** As the basic building block in OOP, the objects bind data and procedures into single entities. In Delphi programming, all classes descend from *TObject*. Despite its lack of sophistication compared to many special-purpose classes it was derived from, the basic TObject class allows the ultimate flexibility of which a descendant class can be defined and used. Consequently, although Delphi libraries provide many descendants of TObject for specific uses, the great majority of custom classes written by the author descend directly from TObject.

- **Source-code-file modularity.** Delphi is not a pure object-oriented language, although it provides full support for OOP. As a variant of the older Pascal programming language, Delphi also fully supports procedural programming it was originally designed for. To take advantage of both, objects and procedures can be logically grouped in source-code files called *Units*. At the end of this section the units and their content will be discussed in more detail.

- **Task oriented modularity.** Objects that are frequently used without alterations can be integrated back into Delphi compiler's library as *VCL Components*. VCL stands for Visual Component Library, a subset of the generic Delphi objects which constitute the highest form of encapsulation in Delphi. When necessary, components can be created in design time and manipulated using Delphi's GUI without having to write a single line of code. Although quite a number of Delphi's standard components are used in this project, no custom components are written as there is no need for them that has been identified during the course of the project.

In this section, the term module is used in the context of source code file. Since in RAD the software evolves from a very simple prototype, it is not possible to separate functionalities in logical modules from the start. Instead, earlier modules are typically constructed as no more than a disorganized collection of objects and functions kept together in the same place to accomplish specific tasks. As their precise nature becomes better understood, the objects and functions are shuffled to the more appropriate modules. New modules were regularly added as the research progressed to further refine the organization of the software source code.

The following is the list of source code files that make up the program modules. The modules are ordered from the simple modules providing basic functionalities to the

complex ones performing the more meaningful tasks using the functionalities found in the former.

- **uError.pas:** a very simple module consisting of a mere series of string constants to be displayed by other routines from modules when a run-time error is encountered.

- **uDebug.pas:** provides a mechanism of conveying various pieces of information to the programmer through the program's user interface for the express purpose of debugging. This module is only used as a development tool and will be removed in the commercial version of the software.

- **uCommon.pas:** provides helper functions to perform various rudimentary tasks such as stack and queue management, scalar and vector evaluation, and so on, that will otherwise obscure and possibly foul the more complex algorithms that use them. Using the functions from this module instead of rewriting the code avoids duplication and keeps the code clarity of the host modules.

- **uGraphicBase.pas:** this is where two dimensional shapes are represented in the logical viewing space. This module provides the crucial link between the world and the screen coordinate systems. Equally important is a collection of functions and procedures that perform various basic geometric operations discussed in later sections.

- **uPolygon.pas:** defines the classes that define the internal representation of polygon in both mathematical and application terms. In a mathematical sense, the polygon is subject to geometric operations such as transformation, projection, clipping, and so forth. Such concepts are generally unknown in the application domain, where the polygon is seen as representation of a real-world entity such as a wall, an obstacle, or a piece of panel material.

- **uProblemSpace.pas:** provides the objects that hold the data of the actual layout problem in the application internal representation. Such objects are instrumental in converting the relatively abstract concepts of layout containers, obstacles, and panel shapes into the formalized structure of raw polygons that the actual calculations of the optimization engine can deal with.

- **uPolyCalc.pas:** manipulates polygons as a set instead of treating them as individual pieces. The set may represent the MCPO problem, its solution, or a subset of either.

- **uPolyClip.pas:** consists of the code for polygon slicing and clipping engines. The tasks of cutting polygon have been found to be non-trivial, the solution of which required novel algorithms to be developed and resulted in substantial amounts of program code. The nature of the problems and their solution are described in more detail in sections *5.5.8. Polygon Slicing with Straight Line* and *5.5.9. Polygon Clipping*.

- **uSolution.pas:** the heart of the program where various nesting and optimization algorithms are to be implemented. It also contains advanced geometric operations to complement those in uGraphicBase.pas. All the code that controls the execution of optimization tasks can be found in this module. It also contains the code of the *Greedy Algorithm* and *Monte Carlo* optimization algorithms.

- **uGAEngine.pas:** contains the implementation additional code of Genetic Algorithm (GA) optimization engine. The algorithm is implemented in a separate unit to allow the engine to be tested and verified before it is actually used in the context of layout optimization.

- **uPolyDoc.pas:** provides control to all the application features. The application main window defined in *fMain.pas* below obtains access to such features exclusively from this module. uPolyDoc.pas handles multiple sets of polygons and manipulates them in a single workspace. Another important task performed by this unit is the reading of sets of polygons and writing them to external files.

- **uWinDisplay.pas:** is a graphic rendering engine written specifically for Microsoft Windows™ Operating System. This module does the conversion from the abstract view coordinates to the computer screen coordinates where the user can visually see the objects. No computation code is written in this module apart from that for rendering purposes.

- **fMain.pas:** is the main window that provides the user interface and the space where the rendering takes place. No computation takes place in this module. Its

only purpose being to provide access to the other units that do the actual computations.

5.2.2. Input and Output Mechanism

The application needs the capability to handle the receipt of input and generating output data in proper format. The Extensible Markup Language (XML) has been used as the base format because of its ease of use as well as its native support in Delphi. This decision is further justified by the wide acceptance of XML as the universal standard for data exchange. By the virtue of modular design however, the different input and output formats can be accommodated later fairly easily by updating the relevant modules.

Moreover, as an XML file physically takes the form of a text file, creating and altering the input can be done easily using an external text editor. The use of an external editor avoids the additional coding work that would otherwise be needed if the input data is to be edited directly in the main application.

Although it has been envisioned that the optimization software should be capable of receiving input from various sources such as disk file, memory stream, and remote procedure calls, implementing such capabilities is decidedly outside the scope of the research. Hence for the purpose of this project, the software currently accepts input exclusively from disk file.

Similarly, the output of the program is currently limited to the computer screen and XML-based disk file. Although it is anticipated that other forms of output such as data streams to be passed to other applications and printed hardcopy may be required later in the commercial version, screen output is considered sufficient for the purpose of this research.

The application's user interface (UI) is also susceptible to change, which is a further reason why maintaining modularity in the code is so important. The current UI is designed for evaluating the performance of the MCPO solution algorithms only. The software can be adapted to industrial or commercial use later by modifying the I/O and UI modules as necessary.

The XML input file is constructed as a series of polygons, as depicted in the simplified picture of Figure 5.1. The actual structure of the hierarchy had undergone numerous modifications to accommodate various additional data. No major rework was necessary

when the structure evolved however, due to the effectiveness of the modular design of the software.

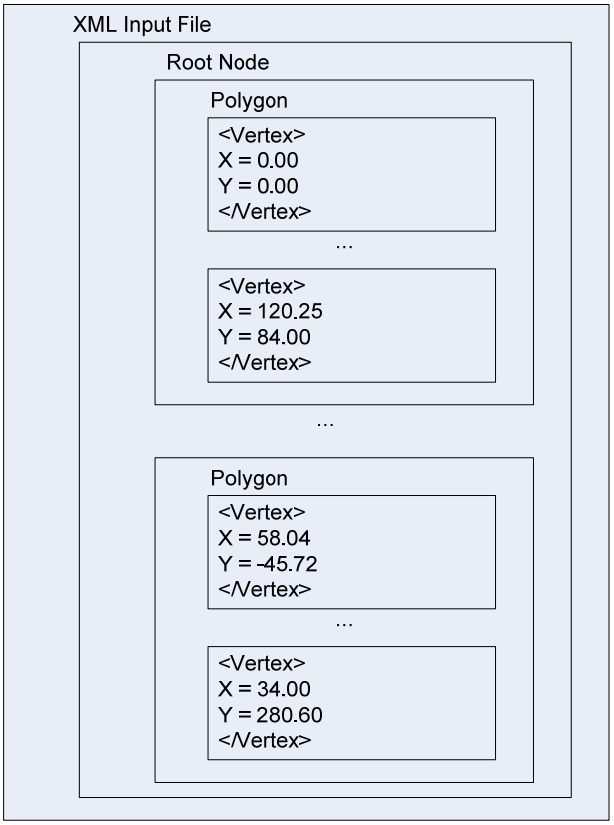


Figure 5.1: The Skeleton of XML Input File

Within the application, an object of **TPolyWorkSpace** class is responsible for handling the input and output data. As the name implies, the object actually serves as the platform on which all the polygon processing takes place. Once loaded from input file, the polygons are stored as descendants of **TPolygonBase** in a variable-length array. TPolyWorkSpace loads the input data when procedure *LoadFromFile* is invoked. Similarly the polygons in the work space can be saved back to an external file by calling *SaveToFile* procedure.

Reading and writing external files are relatively minor features of TPolyWorkSpace. The more important ones are those invoking geometric calculation routines and search of solution for the given problem. The corresponding routines accept specific data structures as input and generate output of sometimes intricate data structures. TPolyWorkSpace has the capability of reading such data structures and visualizing them. Because of these functionalities, TPolyWorkSpace can be regarded as the central point from which the optimization engine is controlled.

TPolyWorkSpace interacts extensively with **TPolyDisplay**, the abstract drawing space on which the polygon objects in TPolyWorkSpace are drawn. Although physically not involved in the actual painting of the polygon objects to the screen, TPolyDisplay has the critical role of doing all the necessary operations required by TPolyWorkSpace to visualize its contents. The role of TPolyDisplay is to provide a translation between TPolyWorkSpace and the system-dependent objects that do the actual screen output.

TPolyScreen provides the actual means of drawing the graphical objects on the computer screen. Because such operations are system-dependent, the implementation of TPolyScreen varies between platforms. Currently TPolyScreen is coded to operate in a Microsoft Windows™ environment only. Transporting the application to another operating system such as Linux should only require modifications to the TPolyScreen object, which demonstrates yet another clear benefit of modular design.

5.2.3. User Interface

The application's main user interface consists of a read-only drawing surface and a panel containing various buttons for quick access to the program functions. The complete set of the program's features is accessible from the pull down menu. Such features are invoked through a set of methods provided by TPolyWorkSpace as previously discussed.

Figure 5.2 below is a snapshot of the application's main window. Since the UI window has been designed to have minimum amount of intelligence, the various input controls on the UI only serve as a link to TPolyWorkSpace's data and methods.

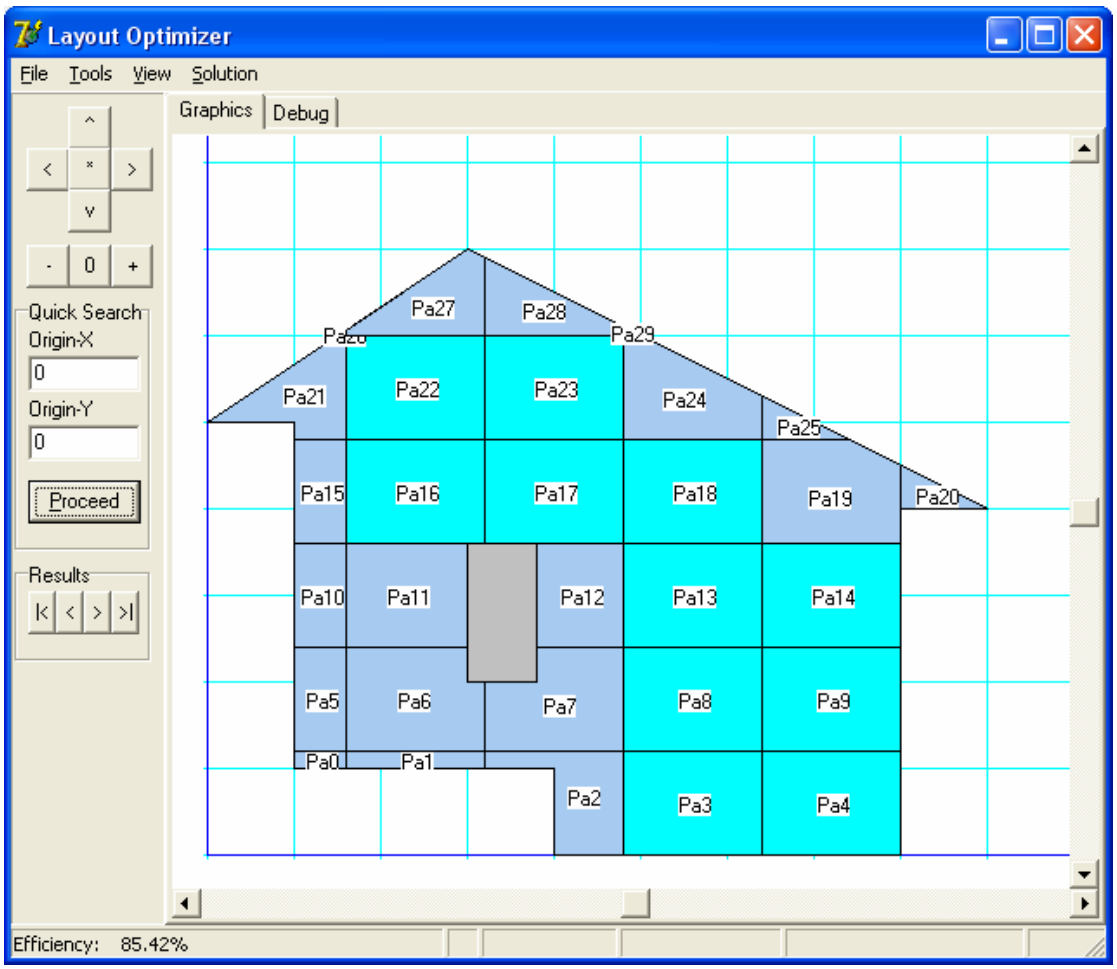


Figure 5.2: Main Window of Nesting.exe Application

Delphi's standard class **TCanvas** is used as the drawing surface in the Microsoft Windows™ environment. This particular class is suitable for vector-based drawing required by the application. Although the TCanvas object is created and owned by the application's main window, its reference is also kept by the TPolyScreen, which uses it as the output drawing space. The application's complete visual output mechanism follows the sequence shown in Figure 5.3.

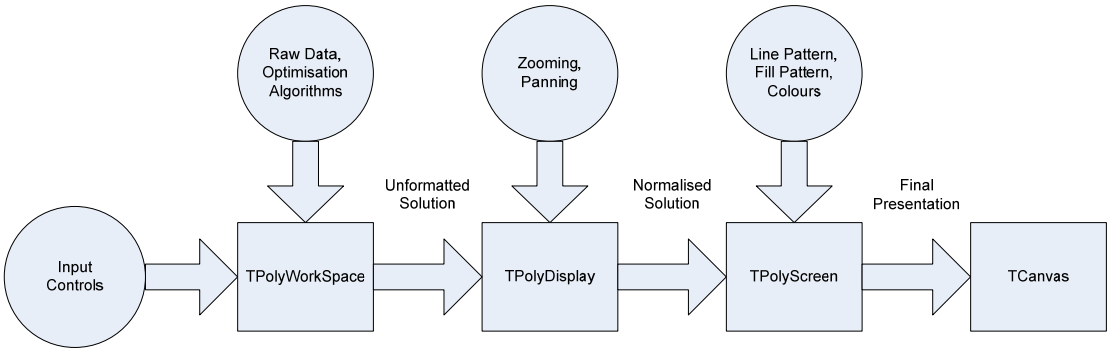


Figure 5.3: Nesting.exe Visualization Sequence

Since TCanvas is already equipped with methods to perform the actual drawing operations on the screen, the work involved in generating visual output is essentially limited to just calculating the correct screen coordinates.

At the current stage it is not known whether a class similar to TCanvas is also available for other operating systems such as Linux. It is perfectly feasible however, to implement a class similar to TCanvas, since all the shapes the application works on are made of straight lines which are relatively easy to draw. Non-linear curves such as circles, ellipses, or Bezier curves are deliberately replaced by their linear equivalent in the current application. Such a restriction is acceptable given the fact that the potential benefit of accommodating non-linear curves is marginal at best, which is easily offset by the major work required to implement them.

5.3. Data Structures

Since Delphi supports procedural and object-oriented programming, both approaches can be utilized to suit specific programming needs within the same project. While OOP has the definite advantage of encapsulation and polymorphism, managing the objects and their pointers requires special care which often complicates the code more than it is worth. The reverse is equally true with the simpler procedural-oriented data structures, which are no more than passive data containers lacking the sophistication of Delphi's classes and objects but hardly need memory management at all. Because of such reasons, both types of data structure are used in the project. The most important ones are described in the following sub-sections.

Although Delphi's *classes* and *objects* have different meanings (class refers to the type whereas the object is the instance of the class), they can be considered the same for the purpose of this discussion. Therefore for the remainder of this section the term object refers to both unless specified otherwise.

5.3.1. Active Data Structures

The term active data structure refers to Delphi's object, which encapsulates both data and methods. This kind of data structure is used when most of the data is manipulated within the objects and little is passed between them. Objects are used heavily in the developed application for the sake of code modularity as previously discussed.

For all their power as a programming concept, objects are also delicate to handle during their lifetime. Unlike simple variables which can be accessed directly, object variables

are only memory pointers. Consequently the memory space for the actual object must be allocated and released by its explicit creation and disposal. At the minimum, failure to release the unused memory space will result in the program leaking memory resources. Typically, careless memory management results in the program becoming unstable during runtime.

In many places in the source code, multiple variables can point to a single object. This multiple reference situation causes removing a particular object from memory without properly updating all variables pointing to it to corrupt the program.

5.3.2. Passive Data Structures

Passive data structures differ from the active ones in the complete absence of embedded methods. In Delphi, the passive data structure is referred to as *record*, as it is in standard Pascal language. The record structure contains only data, hence it is a much simpler construct than the object type discussed earlier.

Unlike an object, the memory space is automatically allocated by the compiler on the program stack when a record variable is declared. Similarly the space is removed from the memory when the execution thread exits the block the variable is scoped for. Simply put, memory management is of no concern to the programmer when records are used.

It is quite possible to dynamically create and destroy records explicitly, such as when dealing with linked lists. Such an operation is roughly equivalent to handling the Delphi objects as previously discussed. However dynamic memory allocation is an exception rather than the rule when handling simple Pascal records. This particular technique is not used in the developed application, save for the linked list structure for the polygon clipping algorithm discussed in section 5.5.9.

The main drawback of passive data structures is the possible data corruption which is the penalty for the absence of built-in methods. When objects are used, it is easy to apply integrity checks at any point during its lifetime, which is typically done when data is passed to the object. None of these can be done with the record type whose checks and validations must be done by external routines.

With those characteristics, passive data structures are ideally suited to use as simple data containers that can be passed through a series of external processes with little deviation happening to their values during that time. This is typically the case of the raw polygons when subjected to geometric operations.

In the case of the developed application, the optimization module uses a number of record data types to represent the geometric entities that make up the optimization problem space. Such record data types are described in more detail as follows:

TVertex is a representation of a point in a two-dimensional plane. The record contains X and Y axes whose values are stored as *real numbers*. TVertex is a fundamental data structure in the entire project as virtually every geometric operation involves the handling of either a series of, or a single TVertex.

TPoint is a native Delphi record that represents a point on the screen. TPoint contains X and Y axes similar to those in TVertex. The axes in TPoint are of *integer* data type however, as they are used in mapping picture elements (pixels) to the discrete matrix of computer display buffer. TPoint is not manipulated in any of the geometric calculations as its use is limited to display purposes only.

TVertices, which is a series of TVertex, is the representation of a polygon of arbitrary number of vertices. TVertices corresponds to the TPolygon object previously discussed, and is used to streamline the process of geometric calculations. All the functions and procedures performing such calculations take TVertices as a parameter instead of TPolygon.

TPrimitiveTriangle represents a simple triangle. The record consists of three TVertex variables to denote the three vertices of the triangle. TPrimitiveTriangle is important for various geometric operations such as surface area calculation and point/shape inclusion as described later in the Geometric Operations section. TPrimitiveTriangle also corresponds to the **TViewTriangle** described later.

TSegment represents a fixed length line. The record consists of two TVertex variables that contain the vertices of both its ends. TSegment is used extensively within the program since in many cases a polygon is regarded as closed loop of segments instead of its most basic representation as an array of vertices.

There are many other active and passive data structures used, however for such special purposes that the data structures will be discussed in the succeeding sections within the context of their usage.

5.3.3. Graphics Pipeline

As common practice in computer graphics programming, a graphical entity goes through a series of transformation widely known as a *graphics pipeline* consisting of a number of transformations to different coordinate systems before it materializes as a series of pixels on the computer screen. Computer graphics programming is a complex subject, and the graphics pipeline approach allows developers to write robust graphics software with reasonable ease. For this reason, the graphics aspect of the application development is dealt with using such an approach. The simplified graphics pipeline is depicted in figure 5.4 below.

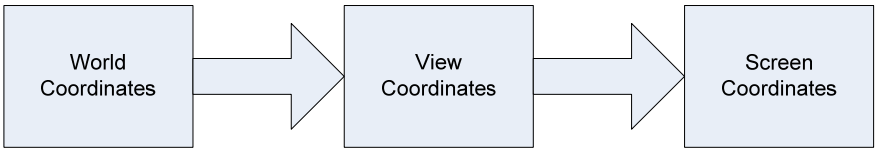


Figure 5.4: Simple Graphics Pipeline

In this project, the *world coordinates* system is used to represent the information about the shape in its physical environment and is typically used as input and output data. The *view coordinates* is used in representing the shape in the working view space where various mathematical operations take place. Finally the results are visualized on the computer monitor using its *screen coordinates*.

5.3.3.1. Shape Objects using World Coordinates

Although all kinds of shapes can be regarded as polygons, special cases exist for which creating corresponding classes is beneficial. Three classes have been identified: the ones representing a *line*, a *rectangle*, and a *generic polygon*.

TPolygonBase serves as the base class for the three identified classes. This is an abstract class, meaning it cannot be instantiated directly as it contains methods yet to be specified in its descendants. TPolygonBase contains variables common to all its descendants such as origin point, pivot point, and orientation angle with which the polygon is rotated at is pivot point. Among other things the class has the capability of writing itself to an open XML file.

TLine is the simplest descendant of TPolygonBase. TLine represent a finite-length segment of a line, which is not a valid polygon as it does not exist in the physical world. Nonetheless, the line is implemented independently because of its

instrumental role in various preprocessing tasks. The line consists of exactly two vertices representing both of its ends.

TRectangle is unique that it represents the rectangular stock panel used in the MCPO problem. The rectangle has four vertices which can also manipulated by altering its length and width properties. These properties are not found in other descendants of TPolygonBase.

TPolygon is the generic representation for any polygon having any number of vertices. Within the context of layout optimization problems however, TPolygon has the restriction of having at least three vertices to reflect the possible number of vertices of 2D objects in the real world. The class does not have a mechanism for detecting *non-simple polygons*, i.e. polygons whose outlines would intersect with each other. A non-simple polygon has no equivalent in real world objects, and to handle them would complicate the program code unnecessarily. Because of this reason, no attempt has been made to accommodate non-simple polygons and any input data containing such polygons is simply regarded as invalid and therefore rejected. More discussion about non-simple polygons can be found in section *5.5: Basic Geometry Algorithms*.

5.3.3.2. Shape Objects using View Coordinates

TViewShape is the only object needed to manipulate and display the polygons, although a special object TViewTriangle as described below is also used. The TViewShape object has the capability of transforming itself within the viewing space as well as performing a few basic geometric operations. One of the most important is perhaps the *triangulation* operation, where the original polygon is decomposed into a series of triangles. More detailed discussion regarding triangulation can be found in the Geometric Operations section later in this report. Note that the term triangulation in this context should not be confused with its more common use for describing the method of locating an object from its distances from known reference points.

TViewTriangle is a helper object used for debugging purposes. The object is used primarily to prove that a triangulation operation actually works. TViewTriangle also has the capability to draw itself on the screen for visual inspection.

5.3.3.4. Shape Object using Screen Coordinates

Along with TViewTriangle, **TScreenShape** is the only object that uses screen coordinates. The only purpose of this object is to display its corresponding TViewShape to the computer display buffer using various colors and filling patterns. Apart from transforming the view coordinates to the screen coordinates, the only calculation done by TScreenShape is *centre of mass calculation*. The calculation is done to determine the coordinates of a string label to be displayed with the shape. Centre of mass calculation will be discussed in more detail in the Geometric Operation section.

5.4. Basic Geometry Algorithms

As the most basic level, a polygon is represented by an array of vertices. The polygon can be visually constructed by connecting adjacent vertices using straight lines. The last vertex in the sequence is then connected to the first to close the loop. Overall, this simple data structure is quite satisfactory for the software's requirements. Nevertheless it does have a number of problems that must be addressed, namely:

- **Non-unique representation:** to represent a shape, one needs to first select the starting vertex from all vertices of the polygon. Second, the direction in which the rest of the vertices are traced, either clockwise or anti-clockwise, must also be determined. Therefore a polygon consisting of *n* vertices can be represented in exactly *2n* ways. To avoid mistaking identical polygons because of their different representations, special *congruence* calculations are required.

- **Non-simple polygon:** The defining characteristic of a *simple polygon* is that none of its sides intersect with each other. A *non-simple polygon* on the other hand has at least two "sides" intersecting. All polygon shapes in the real-world environment belong to the simple polygon type because it is physically impossible to be otherwise. Non-simple polygons, which are regarded as an error condition, can easily occur accidentally when such a simple data structure is used.

Various geometric problems are encountered during the research and development of the application. The rest of this section discusses a few such problems that require substantial effort to solve. A number of them turn out to be subject to research efforts on their own given that solutions are not readily available, and indeed there is a limited amount of published work dealing with such problems.

Bearing in mind that the software application is merely a vehicle for examining and solving the MCPO problem, only limited amounts of time and effort can be devoted to solve the generic geometric problems encountered. The implication is that most of the basic geometric problems are solved using a pragmatic approach when possible. The resulting algorithms are therefore designed for ease of implementation, and are potentially relatively inefficient compared to alternatives with a larger implementation overhead. For all its faults, this approach is nevertheless necessary to keep the project within its defined scope.

5.4.1. Line and Segment Intersection Detection

Line and segment intersection detection is probably one of the most basic yet most extensively used geometric operations. Since only linear shapes are handled, intersection between shapes is relatively simple to detect.

The most flexible way to represent an infinite length straight line in a program code is the implicit linear function:

$$AX + BY + C = 0$$

The intersection point between two linear functions can be found by calculating *x* and *y* values to satisfy both functions. Although the basic calculation is simple, provisions must be made to handle special cases. Naïve calculation for such special cases invariably leads to a division-by-zero operation, which is perhaps the worst kind of run-time error to encounter.

- Horizontal Line: the value of A in the implicit function equals zero. Mathematically, this means the value of Y is constant for any given X. Attempting to blindly pinpoint Y with a standard calculation is therefore a mistake, which will manifest in the division-by-zero error condition.
- Vertical Line: similar to horizontal line case, the value of B in the implicit function is equal to zero. This is because the value of X is constant and Y is utterly irrelevant.
- Parallel Lines: in this case the values of A and B are identical for both functions and an intersection point does not exist.

A segment is a finite length line with defined endpoints. Segment intersection detection is therefore a subset of line intersection detection problem. In this case the algorithm

must perform additional tests to ensure that when an intersection exists between two segments, it must occur within the boundaries defined by the endpoints of both.

Figure 5.5 below shows the segment intersection detection problem. The picture on the left shows an intersection between segments AB and CD exists at P within their boundaries. In contrast, there is no intersection between segments AB and CD in the right diagram although the lines projecting from them do intersect at P'. In the latter case the segment detection function should return FALSE, which will not agree with the result line detection function invoked for the same problem.

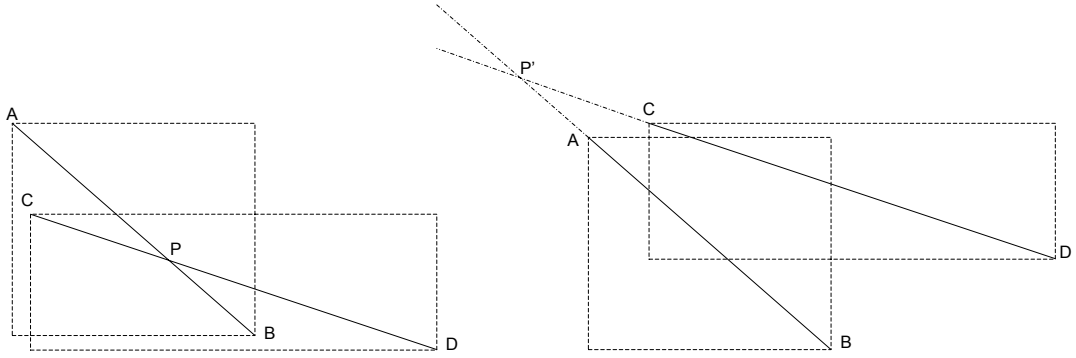


Figure 5.5: Detecting Segment Intersection

Various problems require segment intersection calculations to be solved. An example is non-simple polygon evaluation, which is done by detecting intersections among the polygon's outline. Segment intersection is detected by first finding the intersection point of the corresponding infinite-length lines – if there is any – and verifying whether that point lies within the ranges of both segments in all axes.

5.4.2. Polygon Triangulation

Certain geometric calculations require a polygon to be decomposed into a set of elementary triangles. The picture below shows such decomposition of a 7 vertices polygon to five triangles. The decomposition process, which is also called *polygon triangulation*, is the prerequisite to other operations such as *surface area calculation* and *inside-outside polygon query* to be discussed later.

Extracting triangles from a convex polygon is found to be a simple and straightforward process. Triangulating a concave polygon such as in Figure 5.6 however, is much more complex. Since both convex and concave polygons must be handled, a triangulation algorithm capable of solving both is needed.

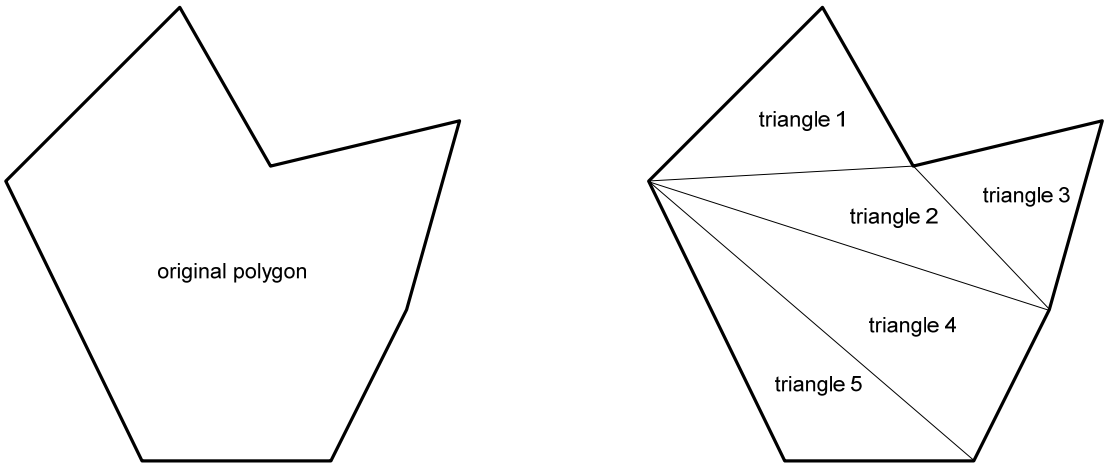


Figure 5.6: Polygon Triangulation

Various algorithms have been developed to solve the general polygon triangulation problem. The apparent simplicity of the general polygon triangulation problem is actually deceptive. For years researchers could only speculate whether an efficient algorithm really exists until as late as 1988, when Tarjan and van Wyk constructed an algorithm that runs in $O(n \log \log n)$ time (Chazelle, 1990).

The early work done by Michel Garey and his colleagues in 1978 resulted in a triangulation algorithm that works on $O(n \log n)$ time (Garey, Johnson, Preparata, & Tarjan, 1978). The algorithm works in two complex stages, making it hard to translate into computer code.

A somewhat simpler algorithm was later proposed by Fournier and Montuno in 1984. The algorithm first breaks the polygon into monotone polygons, which can be easily triangulated afterwards (Fournier & Montuno, 1984). The performance of the algorithm is similar to Garey's algorithm at $O(n \log n)$ time. Fournier and Montuno's algorithm specifically handles all possible cases differently, resulting in a complex structured code which is hard to validate.

Probably the most efficient solution is the one constructed by Bernard Chazelle (1990), which operates at linear time $O(n)$. The basis of his algorithm is the *horizontal visibility map*: the partition of the polygon obtained by drawing horizontal chords from the polygon's vertices. Chazelle's algorithm however, is very complex and difficult to implement.

The most practical approach is to use the *ear-cutting algorithm*, where a polygon is recursively reduced by clipping off vertices protruding from the polygon's hull. Triangles formed from such vertices and their two immediate neighbors are aptly called *ears*. The ear-cutting algorithm is not particularly efficient, with execution time reaching $O(n^2)$ in the worst cases. Nonetheless it is relatively simple to implement and its performance appears adequate for this problem and therefore the ear cutting algorithm is used as the solution for polygon triangulation problem.

5.4.3. Polygon Congruence

As previously explained, the data structure allows a polygon to be represented differently in the memory. To avoid potential confusion during more complex operations, it is necessary to write a polygon congruence detection routine.

The solution is constructed on the premise that the Euclidean distance of any pair of points on a rigid body is constant regardless to the body's orientation to the reference framework. Using this principle, congruence is detected by matching the relative distances of all vertices of a polygon with that of the other polygon it is compared to. Additional measures are also taken to allow detection when different starting points and tracing directions are used.

The Euclidean distance *dist* between two points (x_1, y_1) and (x_2, y_2) on a 2-dimensional plane is calculated using the simple Pythagorean equation:

$$dist = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Because the equality between distances is all that is needed for congruence detection, the expensive square root operation can be omitted without affecting the result. Therefore in the interest of better performance, the comparisons are made on square distances *sqr_dist* instead:

$$sqr\_dist = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Since the square distance is calculated for every pair of vertices on the polygon, the execution time of the algorithm is $C(n,2)$, the number of pair combinations between its vertices. Further because:

$$C(n,2) = \frac{n!}{(n-2)!} = \frac{n(n-1)(n-2)!}{(n-2)!} = n(n-1) = n^2 - n$$

The execution time of this congruence detection algorithm is therefore $O(n^2)$, which is reasonable when n is low. In the current software implementation, the number of vertices is currently limited to $n = 100$. More advanced algorithms may be considered in the future should this limitation prove unacceptable.

As a side note, congruence is still detected when the polygons are shifted and rotated since the calculations are only made on distances of the vertices relative to each other.

5.4.4. Convex Shape Detection

A polygon is convex when there are no cavities on its outline, whereas the concave polygon is one that has one or more of such cavities. A cavity is defined by considering three sequential vertices, and is present if the central vertex is inside the straight line joining the two end vertices. Such cavities are easy to identify visually, as can be seen by examining Figure 5.6, but complex to identify computationally. Although the difference may sound subtle, its implication is not trivial. Concave polygons are much more complex to work on, and often need to be decomposed into convex sub-polygons before certain operations can be achieved.

The solution of convex-concave detection problem is based on the fact that the vector between a pair of two vertices constantly changes in one direction when one traces a convex polygon. In the example below, the convex polygon to the left is traced counter-clockwise with the vectors always changing direction to the left.

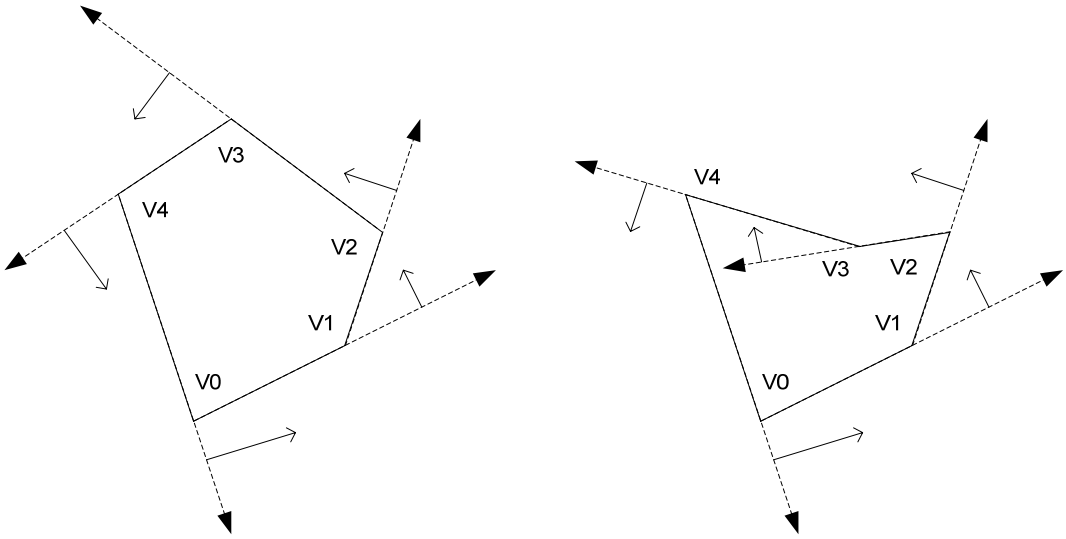


Figure 5.7: Convex and Concave Shapes

On the other hand, the concave polygon on the right has a vector changing direction to the right at vertex V_3 whereas the rest are turning left. Note that when V_3 is discounted, the two polygons above are identical.

The convex – concave detection algorithm can be summarized as follows:

1. For a given vertex V_i, determine a line L_i which connects V_i and V_{i+1}
2. For the V_i and V_{i+1} pair of vertices, calculate whether vertex V_{i+2} lies to the left or right of line L_i calculated above
3. Repeat steps 1 and 2 until all vertices in the polygon is evaluated
4. If the direction change is consistent for all vertices (i.e. all to the left or all to the right), the polygon is convex
5. Otherwise the polygon is concave

The side-of-line calculation is based on the basic implicit linear function for a given pair of vertices (x_1, y_1) and (x_2, y_2).

$$f(x,y) = (x - x_1)(y_2 - y_1) + (y - y_1)(x_1 - x_2)$$

where

$f(x,y) = 0$ when point (x, y) lies *on* the line

$f(x,y) > 0$ when point (x, y) lies *to the right* of the line

$f(x,y) < 0$ when point (x, y) lies *to the left* of the line

In this algorithm, point (x, y) used for the V_i and V_{i+1} pair of vertices is vertex V_{i+2} as indicated in the algorithm.

The execution time of the algorithm in its current form is $O(n)$ which implies satisfactory performance for any given n. Because of this reason, no further optimization is seen as necessary for convex-concave detection.

5.4.5. Polygon Surface Area Calculation

It is absolutely necessary to be able to calculate the surface area of a polygon. This is particularly true in the context of layout optimization problems, where the efficiency of the solution is ultimately determined by the surface area of the wasted material.

Various mathematical formulae exist to do the calculation on various regular-shaped polygons. Using specific formulae to calculate different shapes is very impractical however, and does not offer a solution when irregular-shaped polygons are involved.

A much more feasible approach is to calculate the surface area of a polygon as the sum of surface areas of its elementary triangles. Since the elementary triangles are already provided by the triangulation routine previously discussed, all that remains to be done is the triangle surface area calculation.

The triangle surface area is calculated by using *Heron's formula*:

$$area = \frac{\sqrt{(a+b+c)(a+b-c)(b+c-a)(c+a-b)}}{4}$$

where a, b, c are the lengths of the triangle's sides. The formula in its original form above is numerically unstable on triangles with small angles. To alleviate the problem, the formula is slightly modified and simple pre-processing is added.

$$area = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}$$

and the sides are sorted according to their lengths so that

$$a \geq b \geq c$$

The sum of surface area of the triangles yields the surface area of the original polygon.

5.4.6. Inside or Outside Polygon Query

There are cases where the knowledge of whether a given point lies inside a polygon is critical. The most important ones are polygon overlap detection and polygon slicing described in succeeding sections.

Similar to the surface area calculation, the elementary triangles are used to determine whether a given point lies inside or outside a polygon. The implicit linear function is again used as the basis of determining whether a point resides inside a triangle.

To find whether a point P lies inside a triangle, the algorithm evaluates that for all segments of the triangle, P lies on the same side as the remaining vertex. In the example below P is inside the triangle ABC if all the following requirements are satisfied:

- P lies on the same side as C to segment AB
- P lies on the same side as A to segment BC
- P lies on the same side as B to segment CA

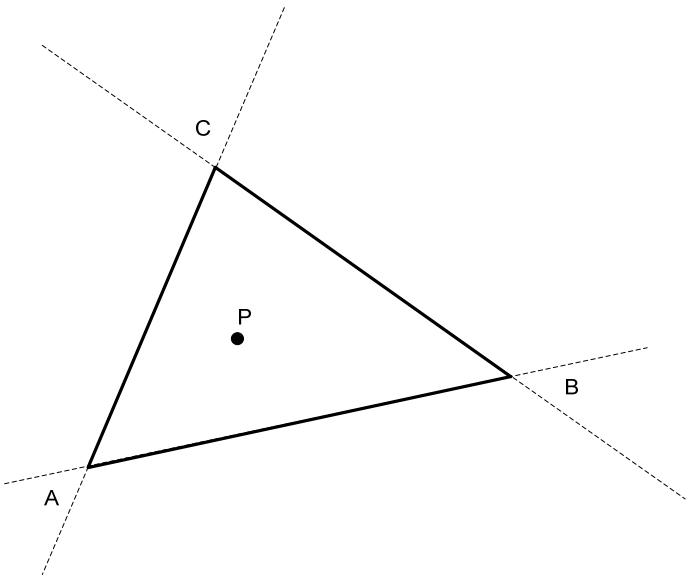


Figure 5.8: Point P is Inside Polygon ABC

To determine whether a point lies inside a polygon, that point is tested to see if it lies inside any of the polygon's elementary triangles. The evaluation returns TRUE when such a container triangle is found. Otherwise the evaluation returns FALSE when none is found after all the triangles have been examined.

The concept can be expanded further to evaluate whether a polygon lies *entirely inside* another polygon. Such a check is easy to perform by testing that all vertices of the polygon lie within the boundary of the host polygon. Using a similar method, it is equally easy to check whether a polygon lies *entirely outside* another polygon. Although very simple, these checks provide powerful tools for more complex operations such as polygon overlap detection.

5.4.7. Polygon Overlap Detection

Overlapping polygons represent an error condition to any MCPO solution. Detecting such overlaps is therefore a critical task to be solved. The detection proves a non-trivial task as illustrated by various possible ways two rectangles ABCD and EFGH can overlap in the pictures below.

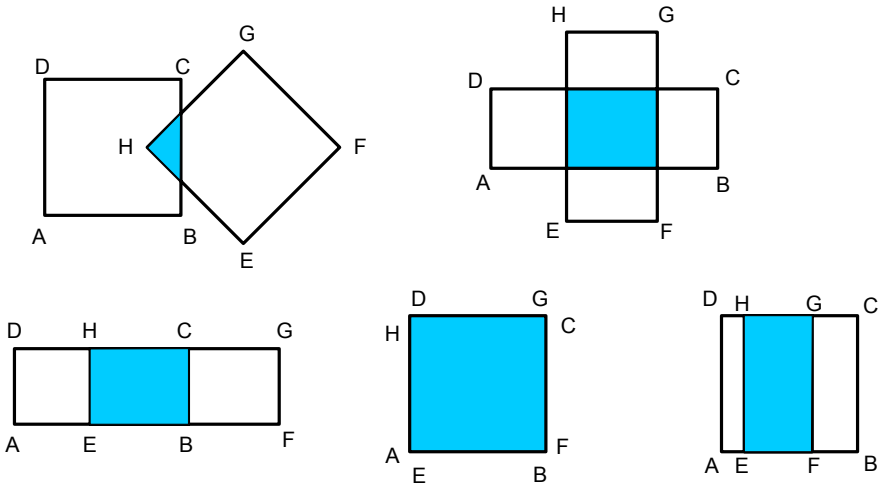


Figure 5.9: Polygon Overlap Examples

The two top pictures show the simple cases of overlap, which can be detected when either of the following is true:

- At least one vertex of a polygon is found within the surface area of the other polygon
- The sides of one polygon are intersecting with the sides of the other polygon

The remaining three on the bottom, however, are the examples of more complex situations where intersections exist and none of the above tests can be used. Because of the diversity of overlap conditions, a detection result cannot be reliably obtained if the test is done on a per-case basis.

David Mount (1992) constructed an algorithm to detect overlap between two simple polygons of arbitrary shape. The algorithm uses separators and scaffolds to simplify the outline of the polygons and recursively refine the hulls until either intersections are found or the hulls merge back to the original polygons' outlines. Mount's algorithm is powerful and efficient, running at $O(m \log^2 n)$ time, where n is the number of vertices and m is the complexity of the polygonal curve separating the two polygons (Mount, 1992). The algorithm is quite complex however, making it impractical to implement.

A much simpler solution is to test for all cases previously mentioned if the polygons involved are limited to triangles. Two triangles are considered to overlap when:

- At least one vertex of a triangle clearly lies inside the other triangle

- The triangles are identical and occupy exactly the same area in the coordinate system

On the other hand, two triangles are considered not to overlap when:

- The two triangles occupy completely different areas in the coordinate system
- One vertex of a triangle lies on the outline of the other triangle, while the remaining two vertices are clearly outside
- Two vertices of a triangle lie on the outline of the other triangle, while the third vertex is clearly outside

The triangle overlap detection provides a solid base for the polygon overlap detection function. A generic solution for overlap detection is to decompose the polygons into their elementary triangles, followed by a test of whether any of the following conditions is true:

- Any of the elementary triangles of a polygon overlaps with any elementary triangle of the other polygon, or
- A polygon lies entirely within another polygon

Using the above series of tests, none of the complex cases in Figure 5.9 can escape detection any more. All the simpler cases are detected correctly as well.

The algorithm is not particularly efficient, since the polygons must be triangulated, for which the ear-clipping algorithm has the complexity of $O(n^2)$, as part of the pre-processing. Later, each vertex of the other polygons is examined against all triangles of the first polygon, making the total complexity of the algorithm $O(n^3)$.

Clearly the performance of Mount's algorithm is superior with $O(m \log^2 n)$ when n is large. The novel algorithm however, works on a more straightforward logic facilitating rapid realization of the actual software module and thus allowing the research and development effort to focus on issues more directly related to MCPO problems.

5.4.8. Polygon Slicing with Straight Line

Referring back to the definition of MCPO problems, cutting the exposed areas of the original container to fit the dimensions of the stock panel is necessary by the time the first sub-problem is solved and the second sub-problem is to be constructed. A simple example of the polygon cutting problem is given below. In this example, an irregular

shaped polygon is sliced by a straight line. The result is three sub-polygons where the intersection points between the original polygon and the cutting line make for vertices of the new polygons.

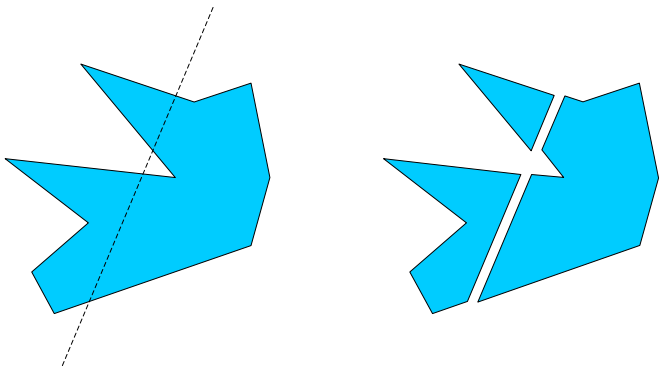


Figure 5.10: Straight Line Polygon Cutting

In the absence of published research on the subject, a custom algorithm has been developed to perform such cuts. The algorithm works on the assumption that for every intersection point between the cutting line and the polygon, there is exactly one intersection point opposite to it. In the abstract form, the algorithm has the following outline:

Straight line cut pre-processing:

```
1.   Trace the polygon from the first vertex to the last
2.   Evaluate whether an intersection is found for every
     segment between neighboring vertices
3.   Store the vertices and intersection points in an array,
     according to the order they are found
4.   Scan the entire array, and for each intersection point,
     identify its opposite
```

Straight line cut sub-polygon construction:

```
5.   Make an attempt to trace neighboring vertices from the
     first in the array to the last
6.   When an intersection point is encountered, jump to its
     opposite
7.   Repeat steps 5 and 6 until the end of the array is
     reached
8.   Store the tracing sequence as the vertices of newly
     created sub-polygon
9.   Remove vertices that have been visited, mark visited
     intersection points as normal vertices
10.  Repeat steps 5 to 9 until the list is empty
```

The pictures below illustrate the process of cutting one such sub-polygon from a ten-vertex polygon on which four intersection points are detected. By tracing the array from V0 to V9 using C0 and C1 as jump points, the new sub-polygon is constructed. The rest of the sub-polygons are cut away from the remaining polygon in the next iterations in a similar manner.

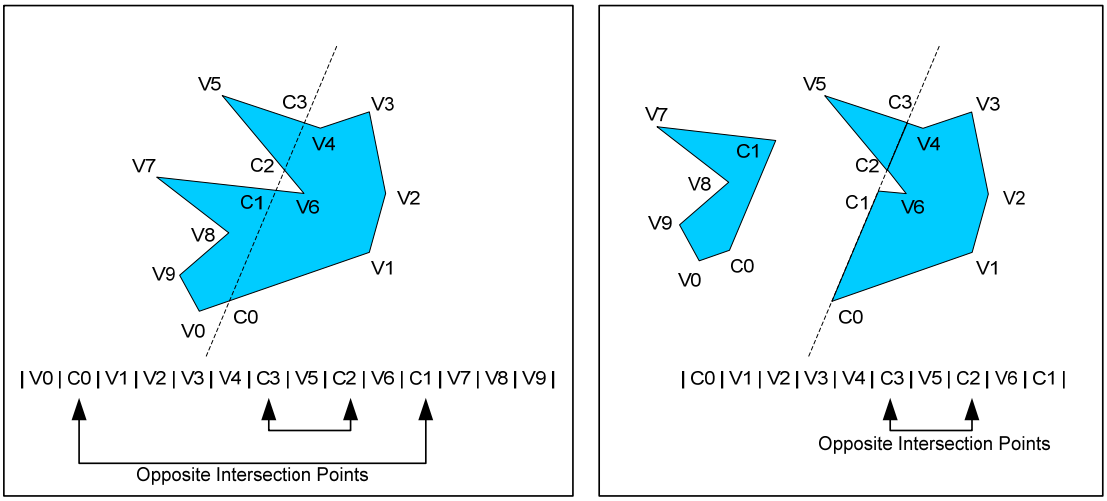


Figure 5.11: Polygon Cutting Algorithm at Work

For any given intersection point, the simplest way of finding its opposite is by selecting the other intersection point closest to it. This naïve approach works generally well when the polygon is traced from its convex vertices, such as V0 in the above example. That is not true however, when the polygon is traced from a concave vertex, such as V6 in the same example. In this case C2 will be selected as the opposite or C1 intersection point, leading to invalid results.

The problem is further compounded by the fact that the cutting line sometimes goes through the polygon vertices. This is especially problematic when the vertex is concave, such as V6 in Figure 5.11 example. Intersection at a concave vertex means that particular point has two opposite intersection point instead of just one.

Such problems are solved by improving the basic algorithm above to accommodate a number of different types of intersection point, coded in the program as an enumerated type with values of *ctNoCutOff*, *ctAtLeg*, *ctAtJointSingle*, *ctAtJointDouble*, *ctInvalid*. In the preprocessing phase, the polygon is traced from the first vertex to the last, where all vertices and intersection points are stored in the reference array marked with the appropriate type.

- ctNoCutOff: when there is no intersection detected between the current vertex and its next neighbor.
- ctAtLeg: intersection is detected between two vertices. This is a normal intersection with only one corresponding opposite intersection point.
- ctAtJointSingle: the intersection is detected at a convex vertex. Similar to ctAtLeg, there is only one corresponding opposite intersection point.

- ctAtJointDouble: the intersection is detected at a concave vertex. This is a special case where two opposite intersection points exist instead of just one.
- ctInvalid: this is used during the sub-polygon construction to mark vertices that have been traced and used

Additionally, a vertex not crossed by the cutting line must either lie to its left or to its right. This information is also stored for the use in the sub-polygon construction phase.

Straight line cut sub-polygon construction:

5. Decide the direction in which the cutting line will be traced
6. Find the first valid intersection point according to the cutting line tracing direction
7. When such an intersection point is found, construct a sub-polygon to the *left* of the cutting line
8. Store the tracing sequence as the vertices of newly created sub-polygon
9. Repeat steps 7 to 8 to construct sub-polygon to the *right* of the cutting line
10. Change the type of all traced ctAtJointDouble nodes to ctAtJointSingle
11. Mark all traced nodes of other types to ctInvalid
12. Repeat steps 5 to 11 until all nodes are marked ctInvalid

The improved algorithm proves much more robust than the original. The result of the polygon cutting process has been found correct in all possible cases when the code was tested. No error has been found during live calculations either. Further work may be required to prove that the algorithm applies to all cases, though it is likely that this will be through empirical evidence rather than an analytical theorem of proof.

5.4.9. Polygon Clipping

A different kind of cutting problem is encountered when a polygon needs to be cut using the outline of another polygon overlapping it. To avoid confusion, the shape to be cut is called the *subject polygon*, whereas the polygon whose outline is used as the cutting template is called the *clip polygon*. The left picture below shows the subject polygon in the shape of a blue rectangle about to be cut with the clip polygon represented by the green triangle. The picture on the right shows the two polygons resulting from the cut.

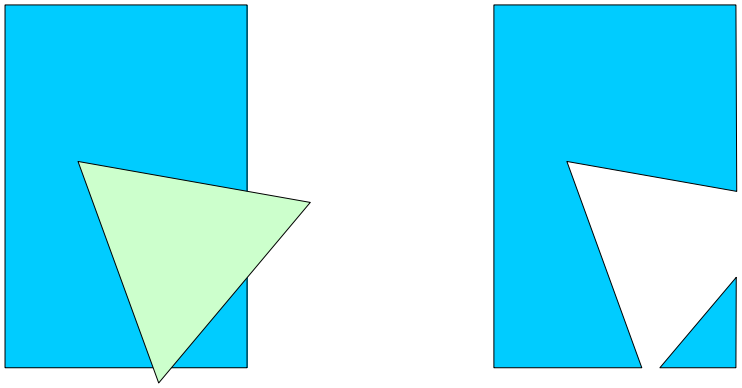


Figure 5.12: Polygon Clipping Problem

This type of cutting is necessary in MCPO problems when there are areas within the container that should not be covered, such as a window on a wall. In such a case, the portion of the panel occupied by the window area must be cut off.

Before contemplating the design of a suitable algorithm for solving the polygon clipping problem, it is necessary to evaluate the possible cases of clipping scenarios and their expected solutions. The given example above is probably one of the simplest and easiest. Figure 5.13 below shows all actual cases found during research and the development and deployment of the optimization software.

An ideal clipping algorithm must be able to efficiently handle all five cases shown in Figure 5.13. The fourth case, however, is peculiar since the result calls for a special data type to represent it. Since polygon representation in layout optimization software only keeps the information about its outline, the result of the fourth case cannot be accommodated without special provisions. Because there is no practical use of keeping such information within the context of layout optimization anyway, it has been decided to disregard the holes altogether and treat the fourth case as if it is similar to the first case. This is justified simply because the outline of the subject polygon is unchanged by the clipping action.

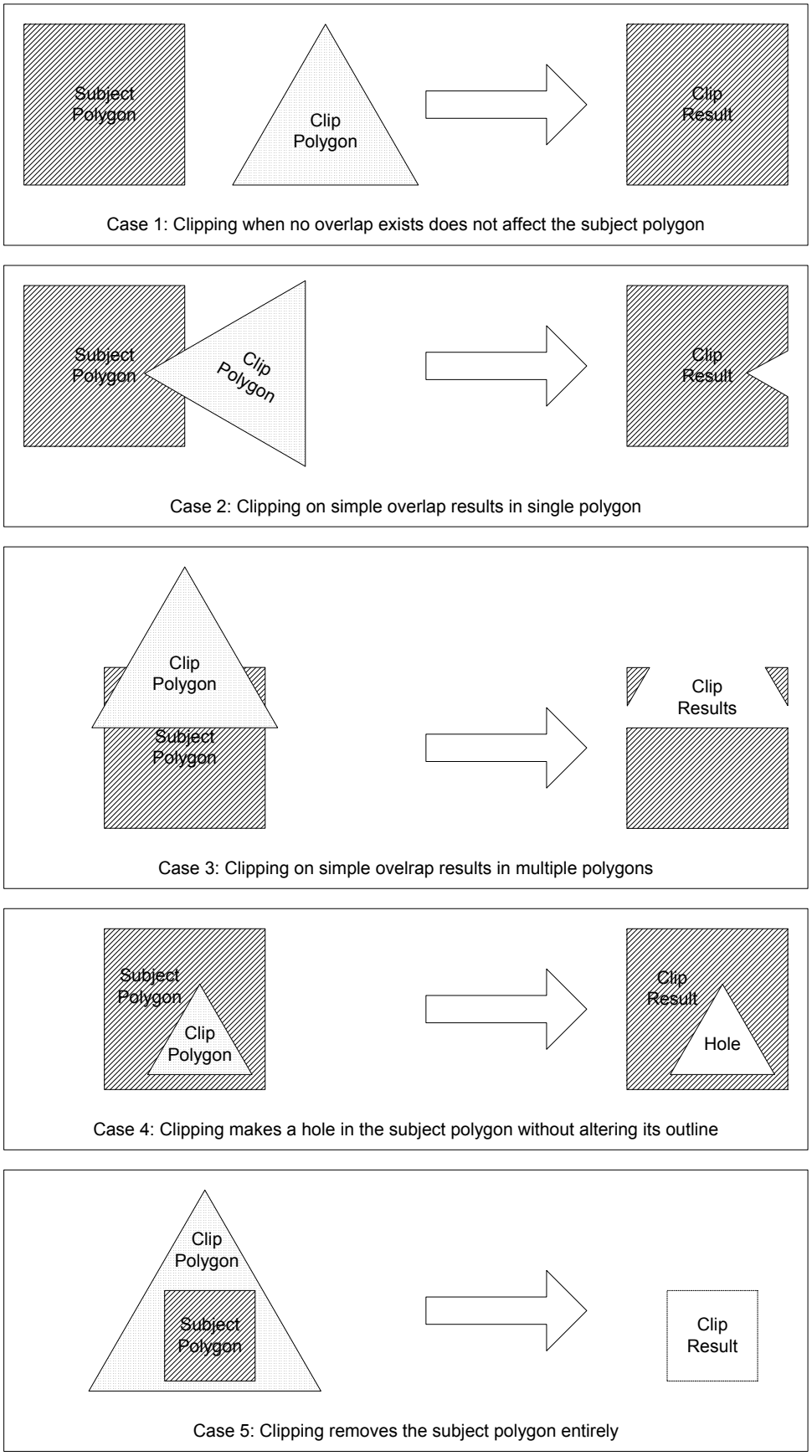


Figure 5.13: Possible Polygon Clipping Cases

There are surprisingly few polygon clipping algorithms to be found in the literature. The development of polygon clipping algorithms is explained by Maillot, 1992; Zhang & Sabharwal, (2002). According to them, Cohen and Sutherland proposed in 1968 what was probably the earliest polygon clipping algorithm, which works through the formal encoding of line and segments. The Cohen-Sutherland (CS) algorithm works with rectangular clip polygons only, but it laid the groundwork for more sophisticated algorithms which came later.

The CS algorithm was improved in terms of performance by the works of Sutherland and Hodgman published in 1974. Their solution is called the *reentrant polygon clipping* algorithm. The Sutherland-Hodgman algorithm gained wide popularity as reflected by their use as a measuring standard by a number of researchers of the field.

Liang and Barsky (1993) later proposed an improved algorithm which treats each segment of the subject polygon independently from each other, cutting the affected segments separately before assembling it to form the final shape. Liang and Barsky claimed the execution time of their algorithm to be only half of that required by the standard Sutherland-Hodgman algorithm.

In the interest of speed, Maillot refined Liang-Barsky algorithm by using integer arithmetic instead of the original floating point calculations (Maillot, 1992). With the advent of multimedia computing, Schneider and Van Weltzen further introduced clipping algorithm specifically tailored for Single-Instruction Multiple-Data (SIMD) multimedia processors (Schneider and Van Weltzen, 1998).

Although numerous performance improvements have been made from the basic CS algorithm, all the algorithms mentioned above share the same basic restrictions:

- The clip polygon has a rectangular shape only
- The valid clip result is only that found inside the clip polygon

The above restrictions severely limit the value of those algorithms in the layout optimization. Firstly, there is no guarantee that the clip polygon in the layout optimization problem will have a rectangular shape. Secondly, the main use of the polygon clipping algorithm is to find shapes that lie *outside* instead of *inside* the clipping region – a direct contradiction to the second restriction. For these reasons, the algorithms discussed above cannot be used in the layout optimization software.

About the only generic clipping algorithm for arbitrary shape of both subject and clip polygons available in the literature has been proposed by Bala R. Vatti. The algorithm is based on the assumption that the edges of one side of the clip polygon will end up as edges of the opposite side of the resulting polygon (Vatti, 1992). Both subject and clip polygons are first decomposed into a series of *bounds*, or the collection of segments facing either to the left or right of the original polygon. The bound is further defined as a series of vertices sorted by their value on the Y-axis, starting with a local minimum and ending with a local maximum. A convex polygon will have exactly one left bound and one right bound, whereas a concave polygon may have more than one of each.

The edges are then scanned from bottom to top to find the intersection points as well as the vertices that contribute to the construction of the sub-polygons. A set of rules is used to determine the nature of the intersection points and their role in constructing the sub-polygons. The edges information is implemented as a linked list, which is fed to the actual clipping algorithm.

Despite Vatti's claim that the algorithm is robust and efficient, it has not been found a suitable solution for the cutting problem at hand because the algorithm is very complex and requires a great deal of preprocessing. The amount of work involved during preprocessing phase is especially apparent when building the linked list to keep the edge information. The algorithm also employs an extensive set of rules which cannot be implemented easily. Finally, the basic algorithm cannot handle the cases where a string of vertices are found on the same Y-coordinate value. To accommodate such cases calls for a considerable expansion of the basic algorithm, further complicating the implementation task. Because of the above reasons, implementing the Vatti clipping algorithms was not a feasible option for this project.

To resolve this problem, a novel algorithm with a completely different approach has been derived and implemented. The algorithm makes extensive use of various other geometric calculations already implemented in the software and whilst it is recognized that it has not been shown analytically to work in all cases it has proven to be a pragmatic solution to a very complex problem with no exception found during testing and use.

The clipping algorithm works by emulating the notion of a pair of scissors cutting a piece of paper using a polygon-shaped flat template. To achieve this effect, the algorithm works in two phases: the pre-processing and the sub-polygon construction. In

the first phase, the intersection points are identified and stored in linked lists. In the second phase, the information from the linked lists is used to construct the resulting polygons.

Polygon clipping pre-processing:

```
1.  Construct a linked list to represent the clip polygon
2.  Construct a linked list to represent the subject
    polygon
3.  Trace the clip polygon from the first vertex to the
    last
4.  For all segments of the clip polygon traced in step 3,
    trace the subject polygon from the first vertex to the
    last
5.  When a segment from the clip polygon intersects with a
    segment from the subject polygon, create a node for
    that intersection point
6.  Insert the node created in step 5 between appropriate
    vertices of the clip polygon
7.  Insert the node created in step 5 between appropriate
    vertices of the subject polygon
8.  Discard the original vertices from both the clip
    polygon and the subject polygon if they are identical
    with the intersection points
9.  Discard the original vertices from the subject polygon
    if they are found inside the clip polygon
```

Both the clip polygon and subject polygon use linked lists of the same structure. The list element contains four pointers, two for the neighboring vertices of the clip polygon and another two for the neighboring vertices of the subject polygon. The result of this pre-processing is two linked lists fused together at intersection points. Figure 5.14 shows the process of the linked lists progressing from the raw polygons to their final form. Although the polygons occupy the same plane, their linked lists are shown on different layers for clarity.

The resulting sub-polygons are constructed from the linked lists by tracing the clip polygon in one direction, taking off a closed loop of vertices whenever an intersection point is found. Such a loop represents a sub-polygon made from the outline of both the clip polygon and subject polygon.

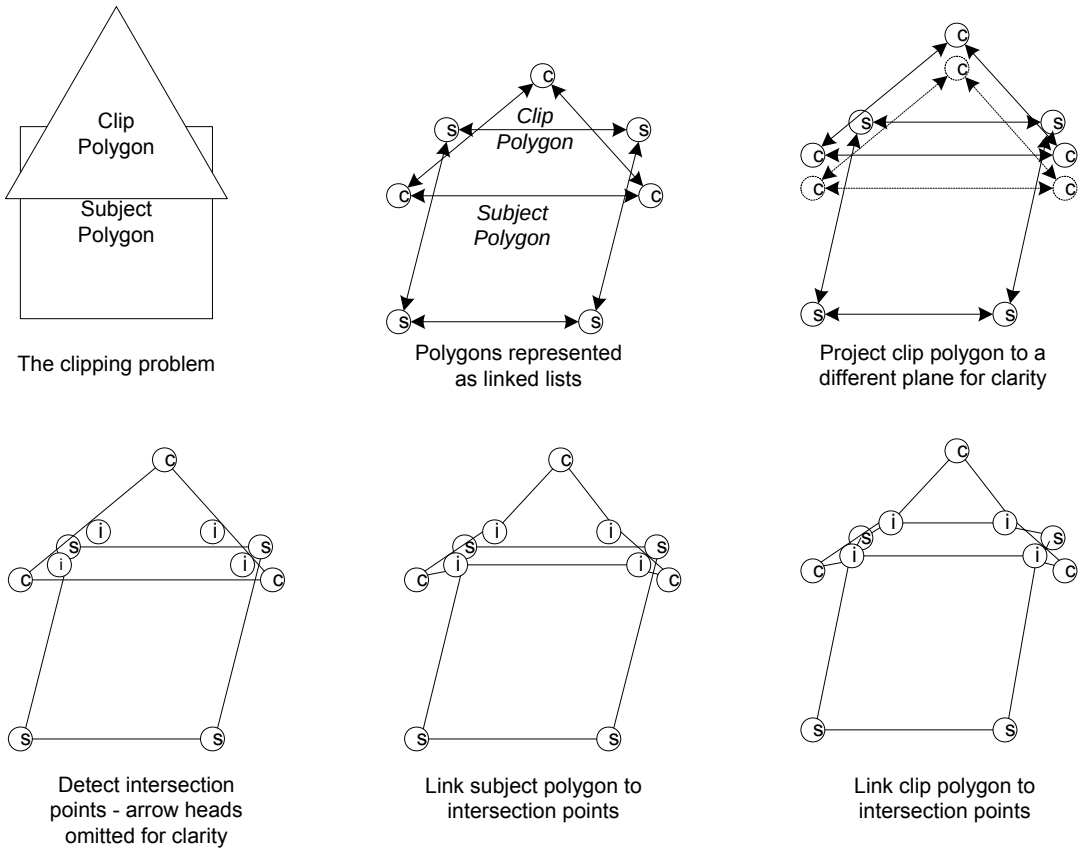


Figure 5.14: Constructing Linked Lists for Polygon Clipping

Polygon clipping sub-polygon construction:

```
10. Find the base vertex at a new sub-polygon by tracing
    the clip polygon in a certain direction (e.g. to the
    left) until an intersection point is found immediately
    after an ordinary vertex
11. If such a base vertex is found, continue tracing the
    clip polygon vertices until an opposite intersection
    point is found
12. From that intersection point, switch to tracing the
    subject polygon in the reverse direction (e.g. to the
    right) until the base vertex is found
13. If an intersection point is encountered in the interim,
    switch back to tracing the clip polygon in the original
    direction
14. Save all the nodes visited during the loop as a new
    sub-polygon
15. Mark all visited nodes as 'invalid' to deny their use
    in subsequent iterations
16. Repeat steps 10 to 15 until all nodes of the clip
    polygon are marked as invalid
```

Although linked lists are notoriously fragile when misused, it can be a powerful programming solution when managed correctly. The polygon clipping engine has been written with this caveat in mind. The resulting code is robust, showing none of the symptoms of memory management failure.

The performance of the algorithm as a whole has been found equally satisfactory in terms of execution time and the correctness of the results. Such an assertion is well supported by various test and live runs even though it is quite possible that the algorithm may fail on some unforseen cases. The engine has been tested for all known clipping problems, with the correct result found on each run. Similarly, there has been no problem found when the engine is used on the actual optimization runs.

5.4.10. Centre of Mass Calculation

Any physical object has its *centre of mass* (CoM), a point where all the object's mass and weight are perfectly balanced. The CoM calculation does not serve any significant purpose in MCPO. However, the calculation is necessary when associating a text label to a polygon during display. When the text label is placed arbitrarily around a polygon, it is easy to lose the association when large a number of polygons are displayed. To minimize this undesired effect, it is most natural to place the label at the "centre" of the polygon.

Figure 5.15 below demonstrates the difference. On the left, the text labels are placed to the upper right corner of the rectangular boundaries of the associated polygons, with an untidy and potentially confusing result. In contrast, the association is much more intuitive in the right picture where the labels are placed on the centre of the polygons.

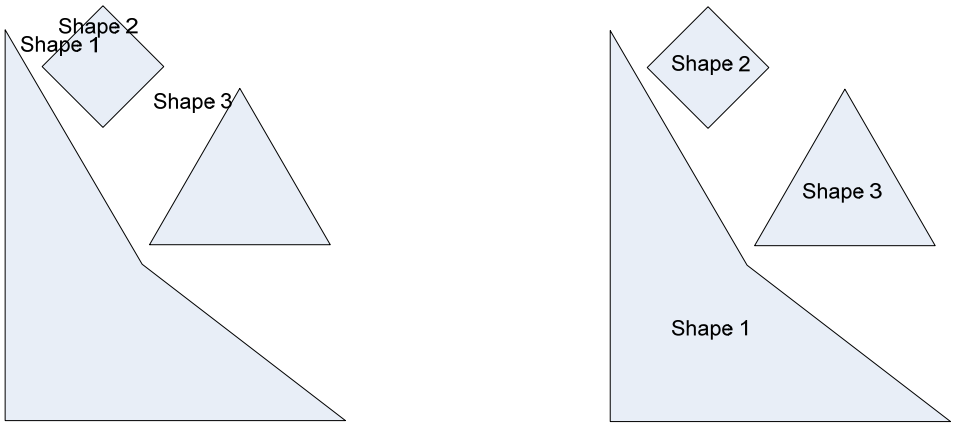


Figure 5.15: An Example of Centre of Mass in Use

The CoM calculation is based on the assumption that the shapes are homogenous. Under such assumption, the calculations for a given polygon are made in three stages: *line calculation*, *triangle calculation*, and *generic polygon calculation*.

Computing the CoM for a line segment (x_1, y_1) and (x_2, y_2) is done simply by finding its midpoint.

$$x_{CoM} = \frac{x_1 + x_2}{2}$$

$$y_{CoM} = \frac{y_1 + y_2}{2}$$

To calculate the centre of mass for a triangle, first the CoM for each of its sides is determined. Then the length of each segment is also calculated. Finally the triangle's CoM is determined by the average of the CoM's as weighted by the corresponding segment length. Hence for a triangle with known CoM for all its sides at points A, B, C, and the length of the respective sides of L_A, L_B, L_C, the centre of mass on its x-axis is calculated as follows.

$$x_{CoM} = \frac{x_A L_A + x_B L_B + x_c L_c}{L_A + L_B + L_C}$$

A similar calculation is done to find the CoM for the y-axis.

The CoM for a polygon is calculated from the sum of the CoMs of its elementary triangles, weighted by their surface areas. Therefore for a polygon consisting of n elementary triangles, with the i^{th} triangle having CoM_i and surface area A_i, the composite CoM for the x-axis is calculated using the following equation:

$$x_{CoM} = \frac{\sum_{i=1}^{n} CoM_i(x) A_i}{\sum_{i=1}^{n} A_i}$$

The CoM for the y-axis is also calculated in the same manner as above.

5.5. Optimization Algorithms

With the tools for all necessary geometric operations implemented and tested, finally the software has taken the shape where the actual optimization can be realized. There are three algorithms studied here: a greedy algorithm, a Monte Carlo technique, and a Genetic Algorithm. This section provides detailed discussion about major issues encountered when implementing each of them.

5.5.1. Greedy Algorithm

For the MCPO problem, the greedy algorithm constructs a solution sequentially by always trying to fit the most profitable piece into the available free space. This is a

short-sighted strategy whose performance can be extremely poor in complex solution spaces due to its inability to escape from local optima. Nonetheless, the mechanism of a greedy algorithm is intuitive and therefore easy to implement into reliable code and also provides a baseline performance to be measured against other methods.

While the concept is simple, the implementation in nesting problems is much more involved because the basic greedy algorithm works only with *scalar* values. To solve a nesting problem, the algorithm needs to be modified to take *vector* values into account. Vector values differ fundamentally from scalar values in that simple arithmetic operations do not apply. Accommodating vector values in a greedy algorithm proves a non-trivial task.

For any given iteration in a nesting search, the greedy algorithm must resolve four key problems:

1. Which candidate piece to select
2. Where to put that particular piece in the nesting container
3. What orientation the piece should be placed in
4. Whether flipping should be applied for the piece if orientation constraints allow

Most of the corresponding parameters, i.e. problems 1, 3, 4, can be represented as scalar values. The second problem however requires a vector parameter for representation.

The basic greedy algorithm essentially addresses only the first problem, whereas the remaining three are not covered because they are specific only to the domain. Valid answers to those additional three will in effect justify the decision made for the first problem by proving that the piece in question can be successfully nested. Recall that the search for valid answers must not violate the two fundamental constraints of the nesting problem:

- The pieces must lay entirely within the boundaries of the container
- The pieces must not overlap with each other

At this point it is also important to appreciate that the objective of the layout optimization is to minimize wasted material. Modeling the problem for a greedy algorithm therefore requires the understanding of how the values of the pieces are quantified to allow the resulting waste to be directly calculated. The physical panel used for the actual building has length, width, and thickness, totaling in three dimensions. In

the model used however, thickness is ignored, leaving two dimensions only. The amount of material is therefore most suitably measured by *surface area* rather than *volume*.

This leads to surface area being selected as the main parameter for the greedy algorithm search. When a candidate piece is to be evaluated, the surface area of the vacant space in the container is calculated. The algorithm then attempts to fit the biggest piece in the pool whose area is smaller or equal to the vacant space into the container. If this attempt is unsuccessful, the next biggest piece is evaluated. The process is repeated until a piece can be legally fitted into the container, which also results in that piece being removed from the pool. If none of the candidate pieces in the pool can be selected, a fresh container is used and the process is repeated.

The algorithm terminates when all the pieces have been used. Because the pieces are the product of the original layout area when it was cut up according to the shape of the container, there will be at least one way to fit a piece into an empty container. Therefore the algorithm is always guaranteed to terminate.

The greedy notion of this algorithm is realized by sorting the pieces based on their surface area in descending order before the actual optimization takes place. Figure 5.16 shows an example of the pieces that emerge immediately after the first stage solution has been constructed.

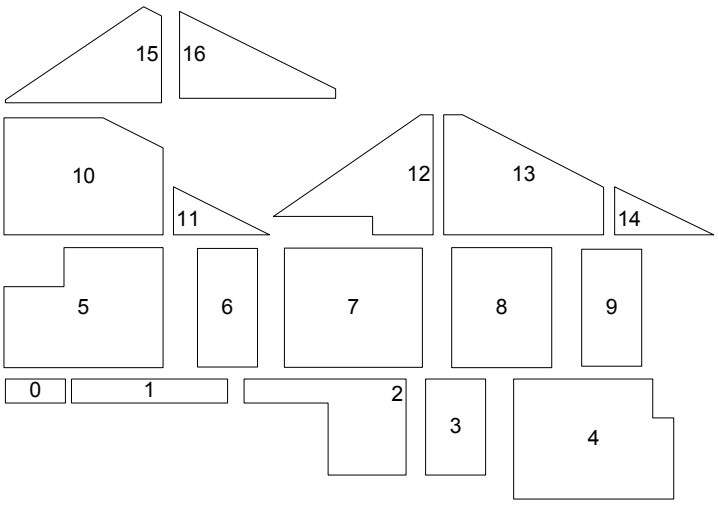


Figure 5.16: Nesting Problem Sample Pieces

Figure 5.17 shows the same pieces reordered in descending order from left to right, top to bottom according to surface area, and ready for optimization.

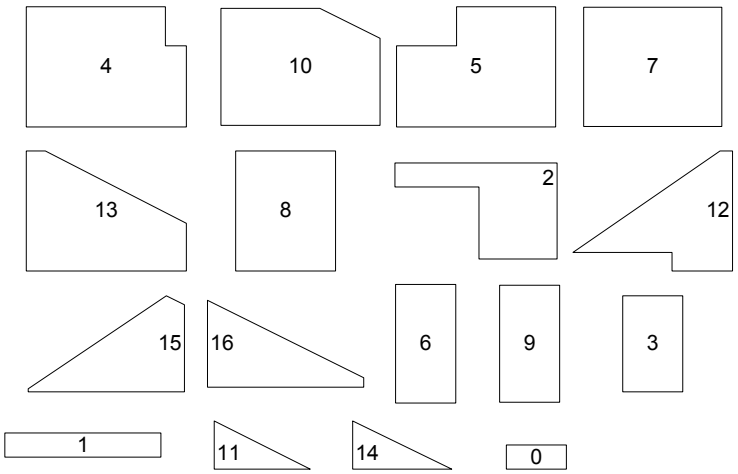


Figure 5.17: Pieces Reordered by Surface Area

The second problem to be solved about a particular piece is about where it should be placed within the container. Using Figure 5.18.a as an example, it is evident that when the container is considered continuous, the candidate panel can be placed inside in an infinite number of ways.

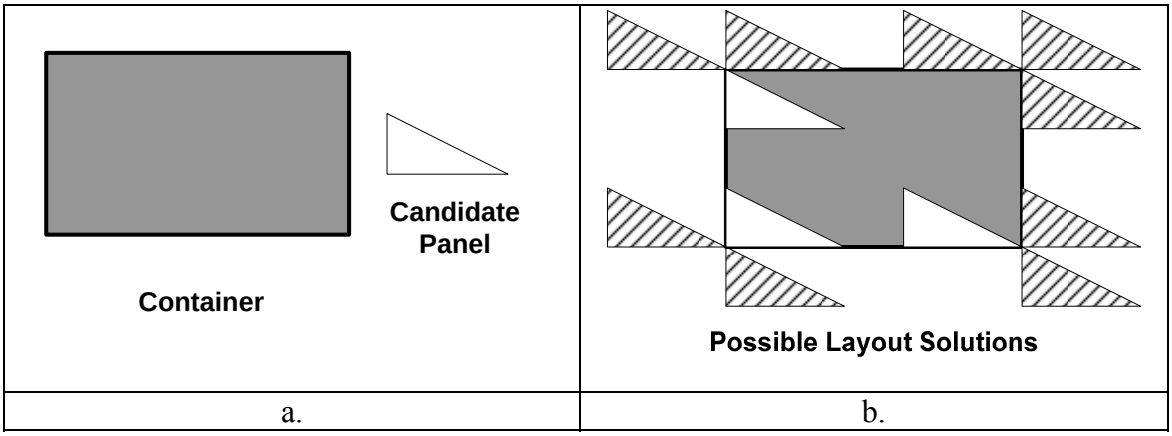


Figure 5.18: Layout Solution by Vertex Incidence

Reducing the container to a discrete set of possible placement choices is vital to make search possible. Given the exponential nature of the size of the overall optimization problem as a whole, limiting the number of possible ways of placing candidate panels in the container from that discrete set is also necessary.

This particular implementation uses *incident vertex* placement, which is an approach similar to linear programming. If the area of the container is considered as the feasible area, then the potential optimum solutions are associated to its vertices. Only those vertices will be evaluated as incident vertex candidates for the panel at hand. The panel is then shifted to various places to make its vertices overlap with those of the container.

Figure 5.18.b shows the evaluation of how a small triangular piece can be placed inside a rectangular container using such a method. It appears that twelve possible solutions exist, of which three are valid as a nesting solution. Because the solution is not singular, a further decision must be made to select the "best" from these equally valid options. There are two options available in response: those based on the *first fit* and the *best fit* strategies.

5.5.1.1. First Fit Strategy

For a given piece, the greedy algorithm selects the first legal placement solution it comes across. No further investigation is made on whether other solutions also exist. This is a simplistic approach, which implicitly assumes that any results, including suboptimal ones, are acceptable.

Because the polygon representation is not singular, there are typically a number of different possible outcomes when a piece is nested using first fit strategy. Figure 5.19 demonstrates a few possible results when a triangular piece is nested in a rectangular container. The algorithm evaluates the container from the first vertex to the last, using them as reference points. For each container vertex, the algorithm then evaluates the possible solution by overlapping the piece's vertices with the current reference point.

Figure 5.19.a shows the best case where the vertices of the container and the piece are ordered in such a way that making the first vertices of both shapes overlap produces a valid result. Figure 5.19.b shows a slightly different ordering of the triangle resulting in two unsuccessful evaluations being made before a legal solution is found. Figure 5.19.c demonstrates even more unsuccessful attempts resulting when the container vertices are arranged differently to that in Figure 5.19.a.

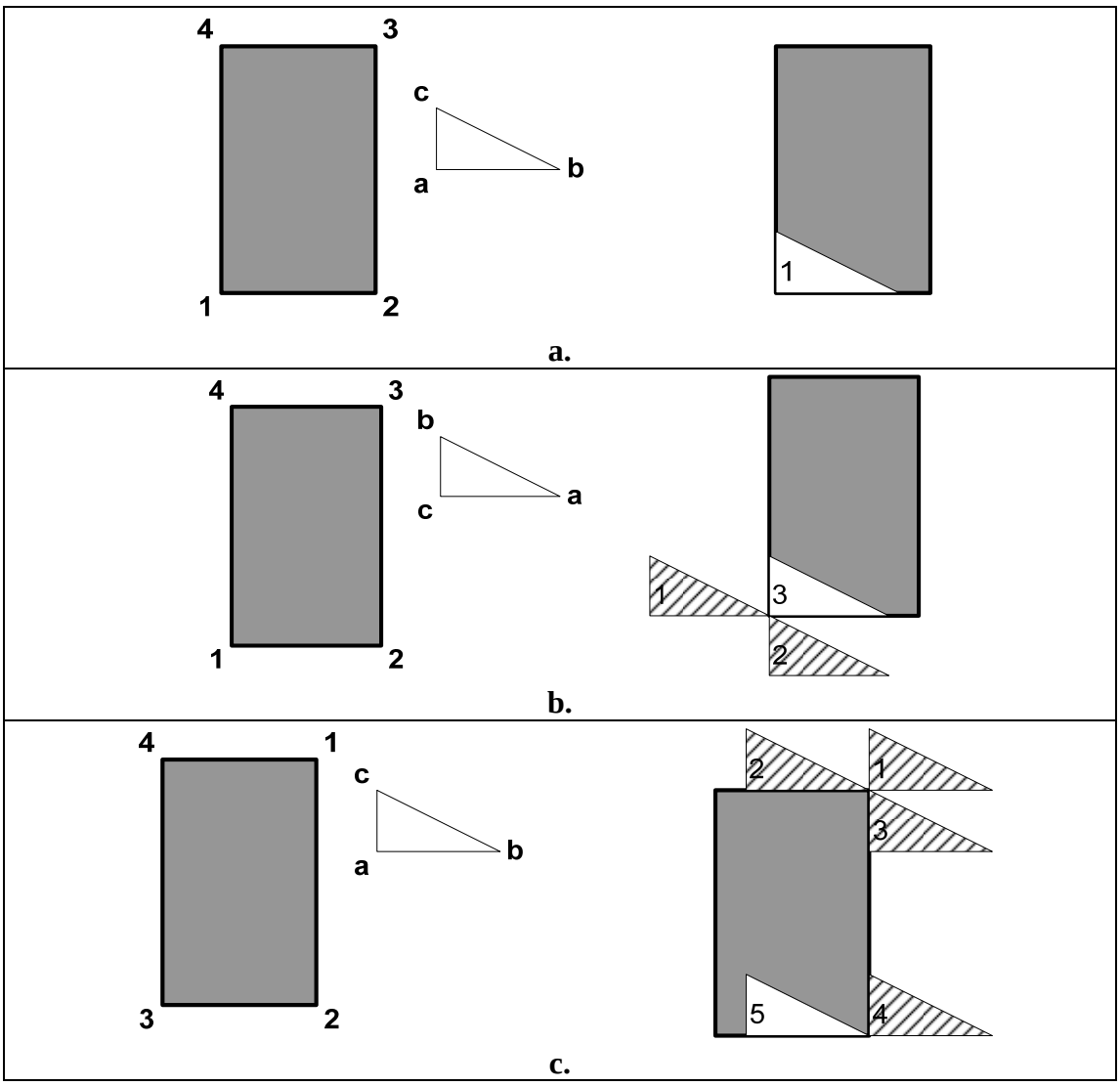


Figure 5.19: A Few Legal Results of First Fit Placement Strategy

Evaluating every single possible solution may seem wasteful, and reordering the vertices prior to optimization to achieve a favorable situation like Figure 5.19.a appears to be an attractive option. This is not possible to implement however, because there are no generic and definite rules to be found in the literature on how the vertices must be ordered to get such a result. Discovering such rules – if they exist – may require a separate project well beyond the scope of the current research. For this reason, no attempt has been made to improve the algorithm performance from this vertices ordering aspect.

Note that in this example, only a single orientation is shown for the sake of clarity. In the actual optimization, all coincidence vertices of flipped and reoriented shapes are also evaluated.

For all its faults, the first fit strategy does have its clear advantages. Apart from being simple to implement, the search is also fast because it ends as soon as a solution is found. In the worst case scenario, the algorithm will evaluate all possible solutions before it realizes that none of them is legal. In the best case scenario, a legal solution is found at the first try, such as the example shown in Figure 5.19.a. Without dwelling on an actual statistical proof, it is safe to assume that the typical case would be that a legal solution is found before all candidates are evaluated.

5.5.1.2. Best Fit Strategy

In the context of layout optimization, the best fit strategy reflects an attempt to improve the odds of achieving a better final result by selecting a legal nesting solution that gives the best chance to put more high-value pieces in the subsequent iterations. Unlike the first fit strategy which settles with the first legal solution it finds, the best fit strategy evaluates all legal solutions before deciding which one to use.

Figure 5.20 shows three legal ways a triangle *abc* can be placed inside a rectangular container. These three candidates will be evaluated to determine which one is the "best". The notion of best solution is elusive and problem-specific however, requiring analysis about what goal the algorithm is set to achieve and what means are available to achieve it.

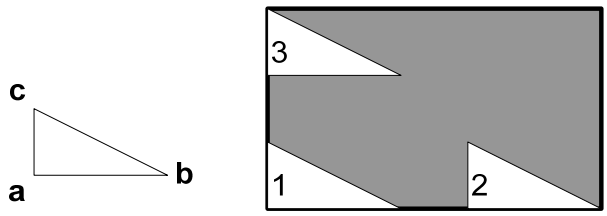


Figure 5.20: Candidate Solutions for Best Fit Placement

Because the objective of layout optimization is to put the pieces so as to occupy as much container space as possible without overlapping, the logical posture of the best fit strategy is to maintain a continuous and convex free space after each piece is placed. Hence, the best solution for a given iteration is the one that provides the least possible obstructions in the remaining unoccupied space.

With the criterion of the best solution established, the next task is to develop an effective and inexpensive way to make the necessary evaluation. Unfortunately there is no straightforward way the amount of obstruction within the vacant space can be measured. A less direct calculation based on *overlapping edges* is used instead. For a

given candidate solution, the length of the edges of the piece that overlap with the outline of the container is calculated. If previously placed pieces exist, the length of overlapping edges with those pieces is also added. The best solution is defined as the one with highest total length of the overlapping edges.

Although there is no proof that the length of overlapping edges is directly related to the amount of obstruction in the vacant space, the effectiveness of this approach is quite evident. In Figure 5.20, solution 1 is regarded as the best since the triangular piece has two edges $\overline{ab}$ and $\overline{ac}$ overlapping with the container outline. Solutions 2 and 3 have one overlapping edge each, with decreasing length. Note that solution 1 leaves a convex free space, effectively meeting the criterion of the best fit solution. In contrast, the shapes of the free area of the remaining two are concave, significantly reducing the chance of nesting large pieces afterwards.

5.5.2. Parameter Representation for GA and MC

As discussed in Chapter 3, the Genetic Algorithm (GA) and the Monte Carlo (MC) implementations share a common characteristic of representing the solution parameters in a bit string. In the GA, the bit string is called a chromosome. There is no special name given for the bit string in MC. The bit string, however, serves exactly the same purpose of providing the algorithms the means of fine-tuning the parameters by manipulating the bit patterns. Additionally, the bit string is physically implemented exactly the same way in both algorithms. For these reasons, the term chromosome is taken as appropriate to refer to the bit string in the context of both algorithms.

To be of any use, the parameters represented in the chromosome must effectively capture the problem that the algorithms are expected to solve. Modeling the problem into a form readily represented as a chromosome is an important task entirely unrelated to how the algorithms will physically manipulate the bits within the said chromosome. Because of its importance, the discussion that follows will be devoted to analyzing the problem and constructing its representation as a chromosome.

5.5.2.1. Parameter Modeling Issues

Substantial effort has been expended in designing the chromosome. Not only because there are multiple parameters involved in layout optimization problems, but some of the parameters are also inter-dependent. To construct a suitable model, it is quite worthwhile to examine the parameters that define a second-stage solution in MCPO. Such parameters are:

1. The total number of stock panels required
2. The list of pieces that are nested within each stock panel
3. The placement coordinates of each piece within a stock panel
4. The rotation and flipping applied to that particular piece

Evidently the first parameter is dependent on the second parameter. Similarly the second parameter is largely dependent on the third and fourth parameters. In the face of this, the only information available to determine the value of those parameters is the list of irregular panels represented by their vertices. This all leads to a situation radically different from standard sheet layout problems found in the literature.

To reiterate, in standard sheet layout problems commonly found in the literature, only a *single* container is provided. The solution designer is therefore allowed to use the chromosome to directly represent the container and map the genes within the chromosome to the nested pieces. Static blocks of bits can be used to represent the placement coordinates of each piece, its rotation, and so on.

This static mapping cannot be easily applied to MCPO, since the number of containers itself is a variable to begin with. The only possible way to accommodate all the parameters within a single chromosome using a static mapping is by allocating a large block of bits for each stock panel to make it able to contain *all* the pieces, and ensure that enough stock panel blocks are provided within that single chromosome to anticipate the possibility of having only one piece per panel. Unsurprisingly, the resulting bit string is very large and prohibitively inefficient to be implemented.

A much more feasible solution is to deliberately use only a few parameters in the main model, and to relegate the task of populating the rest of the parameters somewhere else. Since the first two parameters identified above are the most crucial, they are selected to be represented in the chromosome.

The solutions provided by both the GA and MC therefore only contain the information about how many stock panels are used and the list of pieces that are nested within each stock panel. The problem of how those pieces are actually nested remains unsolved at this level.

Resolving the third and fourth parameters is important to determine whether the solution for first and second parameters is legal. It is most appropriate to make finding their

correct values an integral part of the fitness evaluation function for the original chromosome.

There are two logical ways to solve the above secondary problem at a technical level. The first is by utilizing the same sequential placement routines as used in the greedy algorithm. The second is by mapping the now-static parameters as chromosomes to be processed by the same GA or MC engines used to solve the first two parameters.

The first option has been proven a good choice because it is fast and deterministic in nature. The second option fails on both accounts because having a simulation to determine the fitness function of another simulation squares the total processing time, and the nesting layout found during fitness evaluation cannot be reliably reproduced because of the stochastic nature of the simulation. The last point is especially crucial because only the original chromosome will be retained during the search process, and therefore constructing the nesting layout after the search must yield exactly the same result as found by the fitness function.

The above analysis reflects a very significant finding of this research. As indicated by the discussion in Chapter 3, both GA and MC algorithms are typically used to implement a *simultaneous placement* nesting strategy. The fact that all nesting optimization algorithms implemented in this project eventually use a *sequential placement* strategy rules out the possibility of comparing the performances of the two.

5.5.2.2. Chromosome Definition

After all the relevant decisions been made as discussed above, the problem is now sufficiently reduced to enable the actual modeling of the chromosome. There are only two parameters remaining to be coded in the chromosome:

1. The total number of stock panels required
2. The list of pieces that are nested within each stock panel

Direct coding to the genes in the chromosome is still not possible because the second parameter is of a variable length. To solve this problem, indirect coding employing the concept of *clusters* is used.

In this technique, static blocks in the chromosome are mapped to the pieces to be nested. This represents the worst case solution, where each piece requires an individual stock panel to be used. From the first step of the solution, it is known that all pieces to

be nested are smaller than the stock panels therefore this provides an upper threshold for the maximum number of panels required. Continuing with the example given in Figure 5.16, each panel is associated with a fixed-width block of bits in the chromosome. This block contains only a single variable of integer type, namely the cluster ID. Figure 5.21 shows the association between the panels and the blocks in the chromosome.

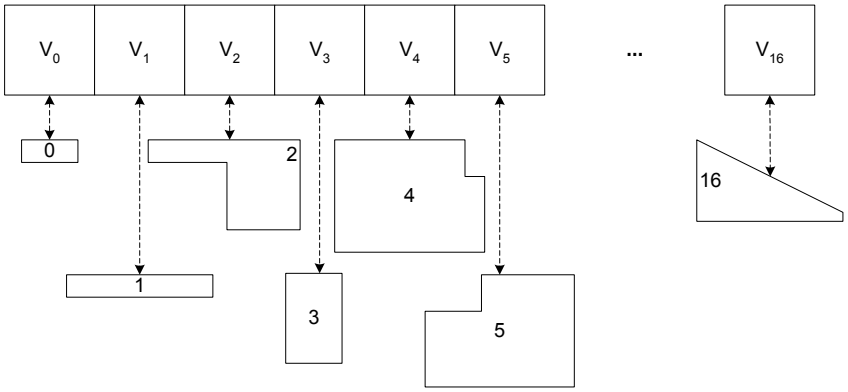


Figure 5.21: Gene to Panel Mapping

The value of each variable points to an imaginary cluster to which the panel belongs. Figure 5.22 shows an example of a populated chromosome with the imaginary clusters that result. Because only 17 panels exist, the binary string can use five bits to hold the cluster ID.

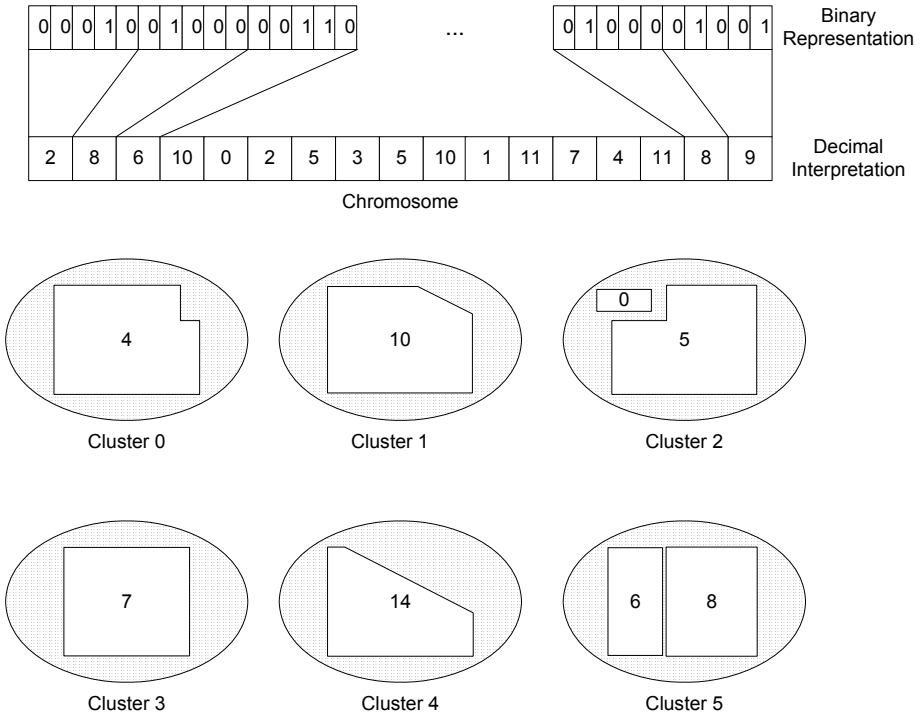


Figure 5.22: Interpreting a Candidate Chromosome

Using Figure 5.21 as reference, it is easy to decode the chromosome to find that the Panel 0 is a member of Cluster 2, whereas Panel 1 is a member of Cluster 8, and so on. Similarly, Cluster 0 appears to have only a single member, i.e. Panel 4, whereas Cluster 2 has two members: Panel 0 and Panel 5.

A cluster is regarded as legal if all its members can be nested in a single stock panel. As previously discussed, part of the fitness function's task is to discover whether such nesting is possible. In the case of an invalid cluster being encountered, there are a number of possible ways to respond. This issue will be covered later in Section 5.5.2.3.

The use of clusters effectively addresses the variable length problem of the nesting list. Because the list only exists implicitly in the chromosome, no assumption about the number of clusters needs to be made beforehand. Furthermore by allowing the pieces to map themselves to the clusters, it is guaranteed that the number of clusters will always be less than or equal to the number of pieces.

Typically, the number of bits allocated for each panel is a good deal more than required to express all the possible Cluster IDs for a given optimization problem. Consequently, assigning the pieces with a random Cluster ID number will often result in single-member clusters with widely scattered IDs. While this phenomenon does not affect the validity of the result, it does potentially bias the optimization engine into giving an inefficient result. This problem is easy to solve however, by using a *modulo* operator to convert all IDs to the acceptable range.

There are many ways to physically implement the chromosome. The simplest and easiest is to use a Boolean variable to represent each bit with the chromosome itself taking the form of a Boolean array as can be seen in the Pascal code written by Goldberg (1989). While this kind of representation is good enough for simple problems involving a few variables, it is not suitable for the multiple-container nesting problem at hand. The chromosome in this project typically contains scores of variables that sometimes number well over of a hundred. Representing each bit within the variable with 8-bit Boolean data type proves prohibitively expensive in terms of memory resources and computation time.

A one-to-one mapping for bit representation is a much more logical alternative. There is a dilemma during the physical implementation, however, on which Delphi native data type to use to represent the variables. Because the variables are only used to represent

cluster IDs, the most appropriate candidates must be unsigned integer types. The basic 8-bit unsigned Byte data type may be too small because it can only hold 256 possible values. The 16-bit unsigned Word data type on the other hand, is way too large. A 10-bit integer would be ideal, as it can hold up to 1024 possible values. It is not anticipated that the number or clusters used in the layout optimization would exceed this number.

The solution is to superimpose 10-bit integer variables in a physical string of 16-bit Word data type. The individual variables can be assigned, extracted, and modified using a series of masking and shifting operations. Figure 5.23 illustrates the custom data structure.

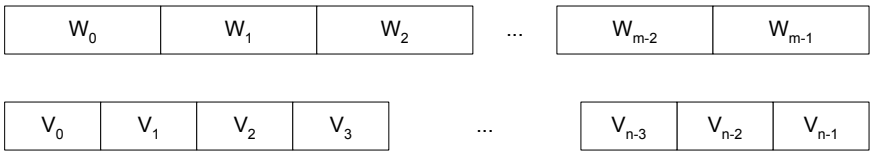


Figure 5.23: Chromosome Physical Representation

A 16-bit Word array of *m* elements is used to represent *n* 10-bit integer variables. In the actual program, an array of 640 16-bit Word variables is used to represent an array of 1024 10-bit integer variables. No memory space is wasted because the number of bits is exactly the same.

5.5.2.3. Accommodating Invalid Chromosomes

A chromosome in the context of layout optimization is accepted as valid only when all pieces can be successfully nested in their associated stock panel. Its opposite is the invalid chromosome, which contains one or more clusters whose members cannot be nested in a stock panel. Because the search performed by both GA and MC algorithms is set-oriented, there is no guarantee that all the clusters extracted from a chromosome are valid. Invalid clusters are found very frequently in the actual tests because many of the individual pieces are quite large compared to the size of stock panels, invariably claiming most of the available area after only one or two nested pieces.

Invalid chromosomes have much less impact on the Monte Carlo technique than they do on the Genetic Algorithm. As discussed in Chapter 3, the MC technique generates a new bit pattern on a completely random basis at each cycle. The bit pattern of the chromosome at any particular point has no influence on the shaping of the bit pattern in the next iteration. As a result, the MC algorithm needs only to retain the best known valid chromosome somewhere in memory and ignore the invalid ones.

In contrast, mindlessly discarding invalid chromosome is not an option using the GA. Because the direction of the search is dictated by the collective patterns in its population of chromosomes, great care must be taken to ensure that the population can survive and retain good quality patterns at each turn of generation. A dilemma inevitably arises: should an invalid chromosome be retained in spite of its utter lack of value as a solution; or should it be discarded and risk the population dwindling and becoming stagnant after just a few generations? The sensible answer must lie somewhere between those two extremes. The following discussion covers three possible policies that can be applied to the fitness function to solve the problem.

5.5.2.3.1. No-Action

The easiest option to handle invalid chromosomes is to pretend that they are nothing more than somewhat unfit individuals. A heavy penalty is given to the chromosome whenever a nesting attempt for its clusters fails. Such a penalty contributes to the overall fitness value of the chromosome, allowing the individual to survive and pass its genes to the next generation.

Although it sounds fair in theory, the actual implementation of this approach is quite difficult. There seems to be no right way to determine how much penalty to be given whenever a nesting failure is encountered. While too low a penalty value will perpetuate invalid individuals in the population, setting the value too high may cripple the chances of an otherwise good solution surviving because of a single nesting failure the chromosome contains.

There is no fitness calculation mechanism that has been devised to reliably resolve this problem although possibly satisfactory solutions exist for further exploration. For instance, a penalty proportional to excess area of invalid clusters can discourage invalid individuals from breeding while high value clusters of the same individual may negate the effect of the penalty. However, additional geometric calculations would be required to implement the ability to calculate a penalty function based on excess area. However, the validity of such a mechanism can be a worthwhile subject for future investigation.

5.5.2.3.2. Append at Tail

A less draconian approach is to interpret the invalid chromosome in a way that turns it into a valid albeit disadvantaged solution. Because the actual nesting routine is sequential in nature, a successful operation is always guaranteed at least for the first few pieces. When the attempt to add a piece fails, that piece can be taken out of the original

cluster and put into another cluster. When all existing clusters are exhausted, new clusters are created to accommodate the rejected pieces. This way, more clusters may eventually be used than originally specified by the chromosome.

With the *append at tail* (AAT) strategy, a new cluster is created at the end of the list whenever an unfit piece is found. All subsequent unfit pieces are added to that cluster. Because the clusters in the list are evaluated from the first to the last, the potentially overcrowded additional cluster at the end will also be evaluated and reduced, with yet another additional cluster added to make the new tail. This newest cluster will have its turn to be evaluated, possibly resulting in more clusters with less pieces. The process is repeated until no more invalid cluster is found. The fitness value of the chromosome is calculated from the wasted area found in all these clusters.

5.5.2.3.3. Redistribute from Beginning

A slightly more sophisticated approach is to attempt nesting the rejected piece in an already created panel before creating the new cluster at the end of the list. This strategy corrects the imbalance of piece density that may occur in the AAT approach. Recall that in AAT, rejected pieces are always added to the cluster at the end of the list. This means that under-populated original clusters will never receive any of the floating pieces. On the other hand the additional clusters always start overcrowded, meaning that they will never be under-populated.

The strategy of *redistribute from beginning* (RFB) addresses this imbalance by ensuring that all of the existing clusters are given a fair chance to accommodate the rejected piece. When a piece cannot be fitted in its own cluster, the algorithm first tries to place it in all populated clusters. Only when this attempt fails is a new cluster created at the end of the list. Theoretically this approach will result in a more even distribution of the pieces, and ultimately a better overall fitness value. It is slower however, due to the extra work involved with the populated clusters.

5.5.2.3.4. Impact to Optimization Algorithms

Although AAT and RFB are sensible correction policies in the interest of obtaining better solutions, they tend to obscure the performance of the host algorithm. In extreme case, one might as well just create one big cluster that accommodates all the pieces and let RFB create and distribute the pieces in a similar way to how the greedy algorithm works. The role of computationally expensive simulation search such as GA and MC automatically becomes moot, perhaps to the point of being irrelevant altogether.

This unfortunate effect is caused by the corrective action intervening with the calculation of the actual bit string. While the bit strings that result from GA operations are not changed when the fitness values are calculated, the non-existent fitness value of an invalid string is substituted with that of its valid equivalent. The surrogate bit string is never introduced back to the population to replace the original. In the absence of feedback mechanism, invalid cluster correction tends to deflect the GA search away from the potentially most productive directions. Considering the value of the invalid cluster correction policies, future work in introducing valid bit strings to the population is necessary to realize the true performance of the optimization algorithm with the current chromosome design. The apparent analogy for reintroducing the corrected strings back into the population would be genetic modification and there is a reasonable chance that such modification could lead to improved performance.

5.5.3. Monte Carlo Technique

Simulation with the Monte Carlo technique uses only a pair of bit strings: the *working chromosome* and the *current best chromosome*. For the number of iterations specified by the user, the working chromosome is subjected to random manipulation and its fitness value is calculated. Whenever the fitness value of the working chromosome is better than previously found, the bit string is copied to the current best chromosome.

The random manipulation for MC is simply done by flipping random bits in the chromosome. The user supplies the numerical constants that control the number of bits that may be flipped, and the probability of a selected bit to actually be flipped.

A fitness value is calculated by considering the total amount of *vacant surface area* found in the nesting containers. Because the objective of the optimization is to minimize this area, a lower fitness value is taken as the better fitness value. For each nesting plan, vacant surface area is simply calculated as the area of the container subtracted with the total area of all the pieces nested inside.

The MC technique represents an undirected search. The algorithm employs no particular strategy other than exploring the multidimensional search space rather aimlessly by randomly changing direction along certain axes at each cycle in the hope of coming across a good solution. Although there is a certain degree of inertia provided by the unchanged bits, they do not in any way contribute to directing the algorithm towards likely better solutions.

5.5.4. Genetic Algorithm

As opposed to the MC technique, the genetic algorithm performs the search in the directions that promise the best result. Instead of just a single working chromosome, a population of chromosomes is used. The search direction is controlled by various bit patterns contained within the population. The GA shares the same chromosome structure and fitness function as those used in MC.

5.5.4.1. Basic GA Algorithm

At the conceptual level, the GA implementation for nesting optimization follows the outline given in Chapter 3. The actual code is based on the simple GA implementation in Pascal written by Goldberg (1989). The associated literature is instructional in nature and the code was clearly written with the purpose of demonstrating the inner mechanism of GA rather than providing the audience with a high-performance version. Not surprisingly, this particular implementation of the algorithm is awkward to use and inefficient performance wise.

Major modifications were necessary to allow Goldberg's code to be used in the layout optimization software as discussed below. The original Goldberg's code remains immensely valuable however, in providing a solid base for this project's actual implementation.

5.5.4.2. Enhanced GA Algorithm

The first fundamental modification is to restructure the code to take advantage of the object-oriented feature of the Borland Delphi™ compiler. Although the standard Pascal code can be compiled directly with Borland Delphi™ without any form of adaptation, adopting an object-oriented form affords the modularity and flexibility for the otherwise monolithic, rigid design. Modularity is especially important because the GA engine would be verified against a few other optimization tasks before its actual use in the layout optimization software. The GA code was duly encapsulated into a Delphi object, making it possible to use exactly the same code to solve various optimization problems.

Another major improvement from the original code is the physical representation of the chromosome. As discussed in section 6.6.2.2, the original code uses a full byte to represent a single bit in the chromosome. This representation is excessively wasteful, particularly when large populations of chromosomes, each containing hundreds of integer variables, are anticipated. A bit-for-bit physical representation explained in

section 6.6.2.2 has been adopted instead, with subsequent updates at various segments in the code.

5.5.4.3. Population Sorting and Chromosome Mating

The most significant attempt at obtaining better future solutions from an existing set of chromosomes is *population sorting*. When a population is generated, its members are sorted according to their fitness values. Chromosomes with better fitness values are placed higher in the list, implying higher chances of being selected to mate.

The chromosomes are selected in pairs for mating, during which *crossover* occurs. A chromosome that has been selected is not eliminated for the selection of the next pair, and stands the same chance of being selected again as before. The reason behind this policy is to allow a supposedly good individual to contribute more than once in creating the next better generation.

Population sorting is the key aspect that differentiates this particular implementation of a GA from a completely random search such as the Monte Carlo technique. Without population sorting and the survival for the fittest rule it implies, the GA will degenerate into a series of indiscriminate mating between random chromosomes with no real chance of optimizing the result.

5.5.4.4. Preserving Good Clusters

As hinted in section 5.5.2.3, a chromosome may contain a number of good clusters, i.e. clusters that translates into a nesting plan with small waste area, as well as bad or invalid clusters. Leaving the good clusters untouched while actively working on the rest may be a good strategy.

Because the existence of the clusters is only implied by the pieces that “belong” to them, the bit pattern of good clusters is immediately found in the bit pattern of the variables within the chromosome referring to them. In other words, the bit pattern of the good clusters is *static*. Ergo, a more advanced concept in GA associated with the bit patterns, the *schemata* can be brought into play.

How the schemata can be used to further enhance the GA implementation for layout optimization has not been explored in this project, mainly because of the perceived complication associated with capturing and handling the bit patterns. Such an investigation remains an interesting subject however, and given the potential to improve

the performance of the algorithm, further research into the area in the future may prove worthwhile.

5.5.5 Verification Functions

Before the optimization algorithms can be used in the actual nesting problem, their implementation must be verified to ascertain that they perform the way they were designed to and are able to produce valid results. Verification is especially important for stochastic methods, a group to which Genetic Algorithms and the Monte Carlo technique belong. Stochastic methods, which imply the use of random variables, are especially difficult to validate analytically because processes monitored are not repeatable.

The greedy algorithm in contrast, is a deterministic technique characterized by the complete lack of use of random variables. The correctness of the algorithm and its implementation therefore can be examined analytically. Verification therefore becomes more straightforward in the case of greedy algorithm.

In the case of the GA and MC, special techniques are required to verify the correctness of the implementation code. The verification takes the form of resolving optimization problems of which the solutions are known. With this approach, the optimization engine is regarded as a black box. No attempt is made to track the activity of the algorithms, only the final result is evaluated.

At the implementation level, the verification routine is realized as the fitness function. The main algorithm itself is left unchanged. Because of the modular design the algorithms have been adapted to, switching between fitness functions can be done with very little effort.

Gordon, Mathias, and Whitley (1994) list a number of test functions that were used to verify their GA variant. Two among those mentioned, the *Rastrigin* and *Schwefel* functions, are widely known as standard test functions for verifying simulation algorithms.

The Rastrigin, a multidimensional function, presents a challenging problem because of the presence of a large number of local optima. The objective of the test is to find the coordinates of x* where the value of the function is minimum. The function itself is defined as follows:

$$R: f(x_i|_{i=1,n}) = A \cdot n + \sum_{i=1}^{n} x_i^2 - A \cdot \cos(2\pi \cdot x_i) \quad x_i \in [-5.12, 5.11]$$

A ≡ a product of ten constants

n ≡ number of dimensions

The absolute minimum point for Rastrigin function is known at $x^* = (0,..0)$. Figure 5.24 shows the plot of the function with $n = 2$. Note that the peaks are not only evenly distributed, but also symmetrical along all the axes.

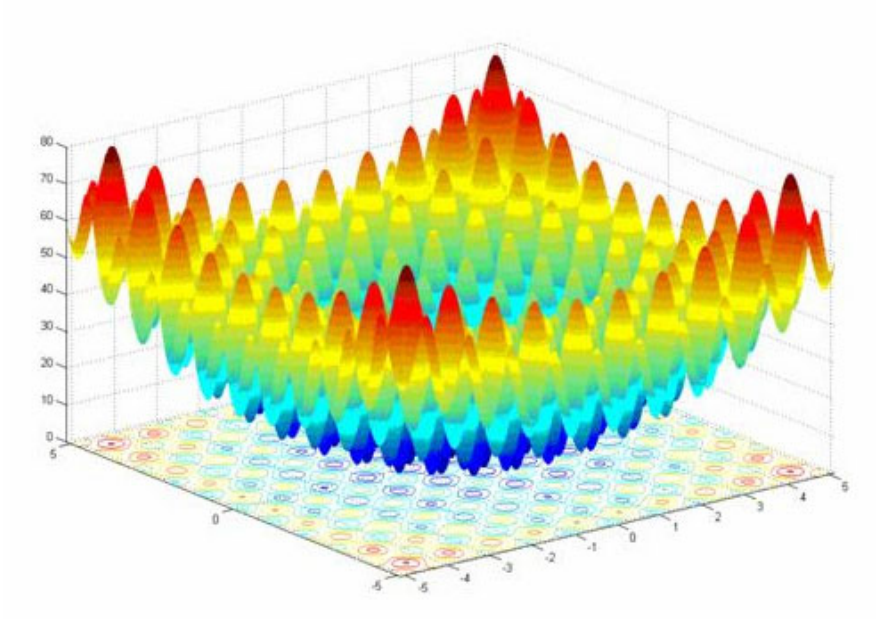


Figure 5.24: Plot of Rastrigin Function (Hedar, 2006)

Similar to the Rastrigin function, the Schwefel function is also multidimensional. The optimization objective is also to minimize the function value. The function is given as follows:

$$S: f(x_i|_{i=1,10}) = 418.9829 * \mathrm{n} + \sum_{i=1}^{10} -x_i \sin\left(\sqrt{|x_i|}\right) \quad x_i \in [-512, 511]$$

The absolute minimum point for Schwefel function is known at $x^* = (1,..1)$. Figure 5.25 shows the plot of the function with $n = 2$. The topology differs significantly from that of Rastrigin function, with second-best point typically far away from the global optimum. The Schwefel function can be extended for $n > 2$ and its complexity increases considerably, with n=10 being considered a non-trivial problem to solve.

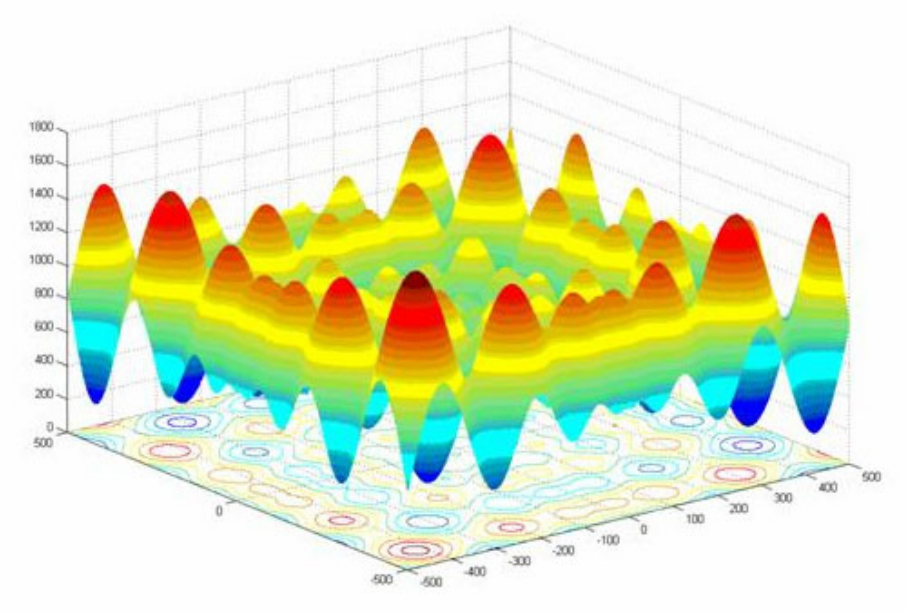


Figure 5.25: Plot of Schwefel Function (Hedar, 2006)

6. Experiment Results

6.1. Experiment Strategy and Issues

The number of parameters used in solving a layout optimization problem is such that a considerable number of possible combinations exist. This problem rules out exhaustive investigation because for each combination of parameters, the actual optimization run is computationally expensive. Only a limited number of optimization runs can be performed given the resource constraints of the project. The experiments therefore need to be configured and executed in such a way that allows the behavior of the software to be monitored and measured through only a small number of optimization runs.

The initial series of experiments have been dedicated to verifying the correctness of the optimization algorithms and their implementations. Verification is especially important for simulation-based optimization algorithms, i.e. the Monte Carlo technique and the Genetic Algorithm, because their stochastic nature is often misleading because it may allow "correct" results to emerge from erroneous processes. The issue is less important for the remaining algorithms implemented in the software because they are deterministic, which implies that flaws in the code can be detected immediately by their failure to yield valid results. In practice, verification for almost all of the modules in the software has been done as integral part of the implementation phase. Consequently, only the experiments that concern the implementation of Monte Carlo and Genetic Algorithm techniques are to be covered in this chapter, though results for these approaches on the MCPO problem will be compared to those achieved by the deterministic solution algorithms.

The remainders of the experiments have been conducted with the aim of achieving the primary goals of this research as defined in Chapter 2. To reiterate, the objective of layout optimization is twofold:

- Generate a layout of a set of stock rectangular panels which covers the container region
- Generate a set of layouts where irregular remaining shapes of the original container can be fitted back into the stock rectangular panels

Two crucial tasks need to be performed by the optimization engine for the first phase: determining the *point of origin* on which the bottom-left corner of leftmost panel will be placed, and selecting the particular stock panel that returns the most favorable result.

The second optimization phase is somewhat easier for the optimization engine because it is only required to search for a set of layout plans according to user-specified parameters.

A few trial runs quickly reveal that for a given point of origin used in the first phase optimization, there is a sizeable amount of computation that follows before its corresponding final result can be obtained. The problem is particularly severe when a simulation-based optimization algorithm is used in the second phase. As will be discussed in later sections, resolving a moderate-sized problem using simulation-based algorithm for a single point of origin can easily take hours or days, even when reasonably powerful computer hardware is used.

A major contributor to the computation cost, however, is the multiple candidate stock panels associated with each optimization case. Because selecting the most productive stock panel dimensions from a pool of candidates is one of the prime objectives of MCPO, this feature cannot be dispensed with and the resulting computational cost must be accepted.

It is clear that exploring multiple points of origin is not a feasible option except in very simple cases. Real-life examples are typically complicated enough to render multiple points of origin prohibitively expensive to compute, regardless of the strategy in selecting those points. Because of this reason, all the optimization runs will be conducted with a single predetermined point of origin only. The chosen point of origin is at (0, 0) in the workspace coordinates, which is arbitrary because of the non-unique way the container can be placed in the workspace. Future research can explore ways of making an intelligent selection of the point of origin.

There is still an array of parameters whose values need to be determined before the second stage optimization can take place. In a commercial setting, the users can tune all of these parameters through the UI according to their own preferences and reasoning. In these experiments however, the main interest lies in finding the comparative performance between algorithms in terms of execution time and the quality of the results. Consequently, few of the parameters will change during the course of the experiments. Values set for those parameters and the justification behind them will be provided on per case basis.

Apart from the algorithm performance, the experiments will also provide some additional information that may be of importance. Especially interesting is the effect of placement strategy (first-fit compared to best-fit), piece flipping and rotation to the overall efficiency of the solution.

6.2. Verification on Numerical Functions

As has been discussed, the goal of the first part of the experiments is to verify whether the simulation-based algorithms have been correctly implemented. The requirement for the verification is simple: a "correct" implementation must consistently show converging pattern towards the optimum solution, and terminate after a limited execution time, i.e. never enter an infinite loop. Both simulation based algorithms, the Monte Carlo technique and Genetic Algorithm will be tested. Two verification functions discussed in Chapter 5 are used for each: the Rastrigin Function and the Schwefel Function.

6.2.1. Rastrigin Function

The Rastrigin Function as defined in Section 5.5.5 is used:

$$R:\ f\left(x_i\ |_{i=1,n}\right) = A \cdot n + \sum_{i=1}^{n} x_i^2 - A \cdot \cos\left(2\pi \cdot x_i\right) \quad x_i \in \left[-5.12, 5.11\right]$$

In this example, the two-dimensional function is used (n = 2) with A = 10. The absolute minimum point is known at (0, 0) with the function value of $f\left(0,0\right) = 0.00$. It can also be seen from Figure 5.24 that multiple peaks exist in the search space, regularly spaced with local minimum function values increasing in direct proportion to distance from the global minimum.

The variables are simply represented by two 10-bit blocks in the chromosome. Partitioning the normal [-5.12, 5.11] range as specified in Section 5.5.5 will result in 0.01 increments which is relatively coarse. Instead of the normal [-5.12, 5.11] range therefore, a smaller range of [-1, 1] is used to make the increment smaller. The [-1, 1] range in each dimension is partitioned equally in a 10-bit vector, resulting in each bit increment corresponding to an approximately 0.002 increment in the search space.

6.2.1.1 Monte Carlo

The Monte Carlo optimization performs the search by attempting to flip every single bit of the chromosome in each of the 500,000 iterations. The flip probability is set at 0.5, which makes each bit very unstable, but guarantees that a large area will be searched. Five optimization runs with an identical set of parameters have been performed.

Figure 6.1 shows the converge pattern of the MC optimization. All the trace lines exhibit steep improvement in the initial few cycles. The lines generally reach "acceptable" solution within less than one hundred cycles, from which the gradients become level until the absolute optimum solution is reached.

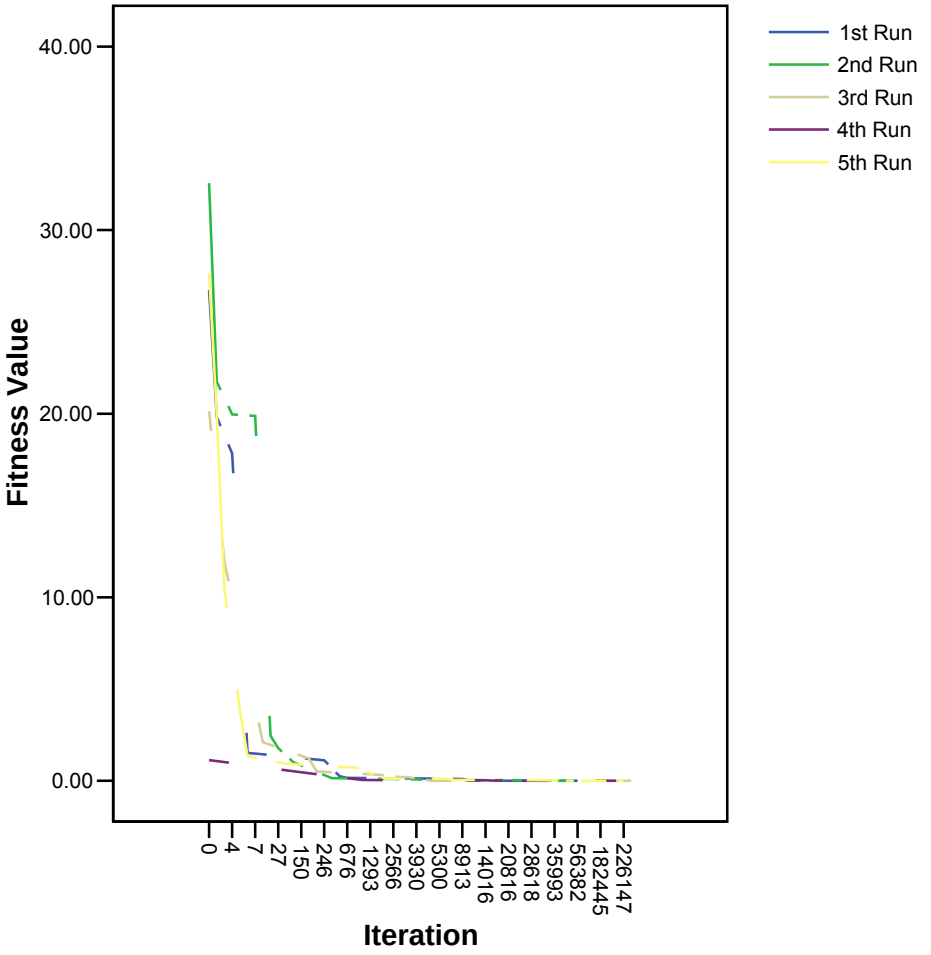


Figure 6.1: Monte Carlo Search Convergence for the Rastrigin Function

The algorithm succeeds in finding the absolute optimum solution in all five cases well before the maximum allocated 500,000 iteration is reached. In fact, the algorithm always finds approximate solutions very close to the absolute before 10,000 iterations. Clearly 500,000 iterations is significantly more than required in this case, indicating an opportunity of performing the search with comparable results at much fewer iterations.

Nonetheless, the experiment demonstrates that MC works satisfactorily as it stands. Because there is no way of knowing how many iterations will be required before acceptable results can be obtained, an attempt at optimization in this respect will not be worthwhile and thus no modification will be applied to the MC implementation for

further use, as it is primarily intended as a benchmark against which to assess the performance of the GA.

6.2.1.2. Genetic Algorithm

Optimization using the Genetic Algorithm is done using the total number of evaluations that match the previous experiment with the MC technique. A population of 500 individuals evolving through 1,000 generations is used to make the equivalent of the 500,000 evaluations used in MC.

The two remaining key parameters of the GA however, the crossover probability and the mutation probability, do not have a counterpart in the MC technique. In our experiment we use a crossover probability of 0.6 and a mutation probability of 0.03 as these are commonly held to be reasonable settings for a simple GA.

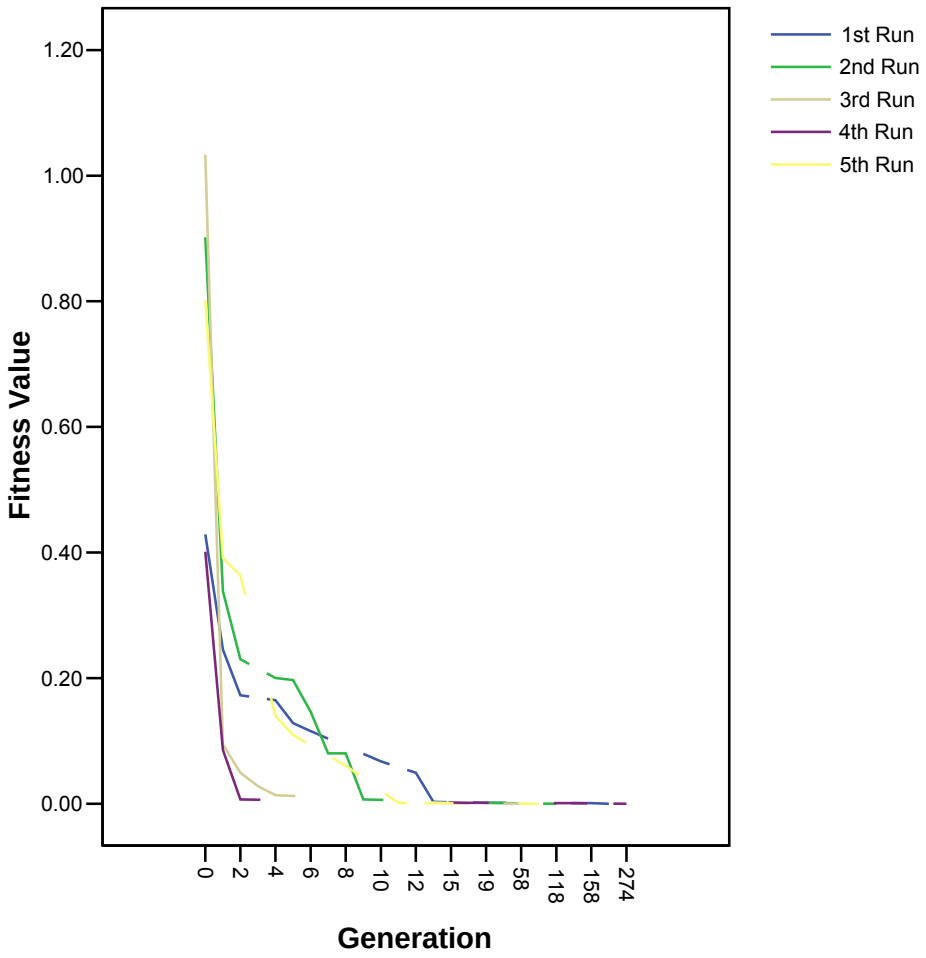


Figure 6.2: Genetic Algorithm Search Convergence for Rastrigin Function

Admittedly there is no direct correlation between these two control parameters with the flip probability of MC technique. It matters little however, since the objective of this exercise is to verify the correctness of the GA implementation rather than making a

direct comparison with that of MC. Results for the GA convergence are given in Figure 6.2.

As evident in Figure 6.2, the search successfully converges to the global minimum in every run. Similar to that in MC, steep improvement is achieved in the initial few generations, after which it becomes more and more level until the optimum solution is found. At around 20 generations (10,000 evaluations), the best candidate solution has come very close to the absolute minimum. The global minimum solution itself is reached after no more than 275 generations (137,500 evaluations), even in the worst case.

Recall that each variable is represented by a 10-bit string, resulting in 2^{10} = 1,048,576 possible values in the two-dimensional search space. Because there are 500 individuals in the population, GA is able to find the optimum solution in 137,500 evaluations in the worst performing run. In this instance, the optimum solution is found by exploring just 0.13% of the total number of candidate solutions in the search space.

These two experiments clearly demonstrate that both MC and GA implementation can perform consistently in finding the global minimum of the Rastrigin Function. The GA, however, shows better performance by finding the global minimum after an average of 53,200 evaluations as compared to 210,212 achieved by MC, though even this higher number of evaluations is a relatively low percentage of the total solution space.

6.2.2. Schwefel Function

Referring to Section 5.5.5 once again, the Schwefel function is defined as:

$$S: f\left(x_i \mid_{i=1,10}\right) = 418.9829 * \mathrm{n} + \sum_{i=1}^{10} -x_i \sin\left(\sqrt{|x_i|}\right) \quad x_i \in [-512,511]$$

The global minimum value is found at $x^* = (1.00,..1.00)$ with the function value of $f(1.00,..1.00) = 0.00$. Unlike the Rastrigin function however, the peaks in the search space are less regularly distributed. Perhaps more importantly, the global minimum and its second-best minimum are widely separated, making it difficult to recover from local optima.

A 10-bit block is used to represent each [-512, 511] variable range, partitioning it so that each bit increment corresponds to one unit increment within the search space. Because of the one-unit increments, the exact variable value of 420.9687 will never be evaluated

making it impossible to find the true global minimum. Its approximation however can be found at x = 421.

Unlike the Rastrigin function, the Schwefel function is tested for different numbers of variables to highlight the performance difference between GA and MC. This strategy will also confirm that any number of variables less than the maximum of 640 specified in Section 5.5.2.2 can be reliably mapped to the same chromosome structure. A 10-bit block is used to represent each [-512, 511] variable range, partitioning it so that each bit increment corresponds to one unit increment within the search space.

6.2.1.1 Monte Carlo

Similar to the Rastrigin function test, the MC search is done with a total of 500,000 iterations. The same flip probability of 0.5 is also used. Figures 6.3, 6.4, and 6.5 show the convergence patterns of MC search for n = 2, 4, and 10 respectively.

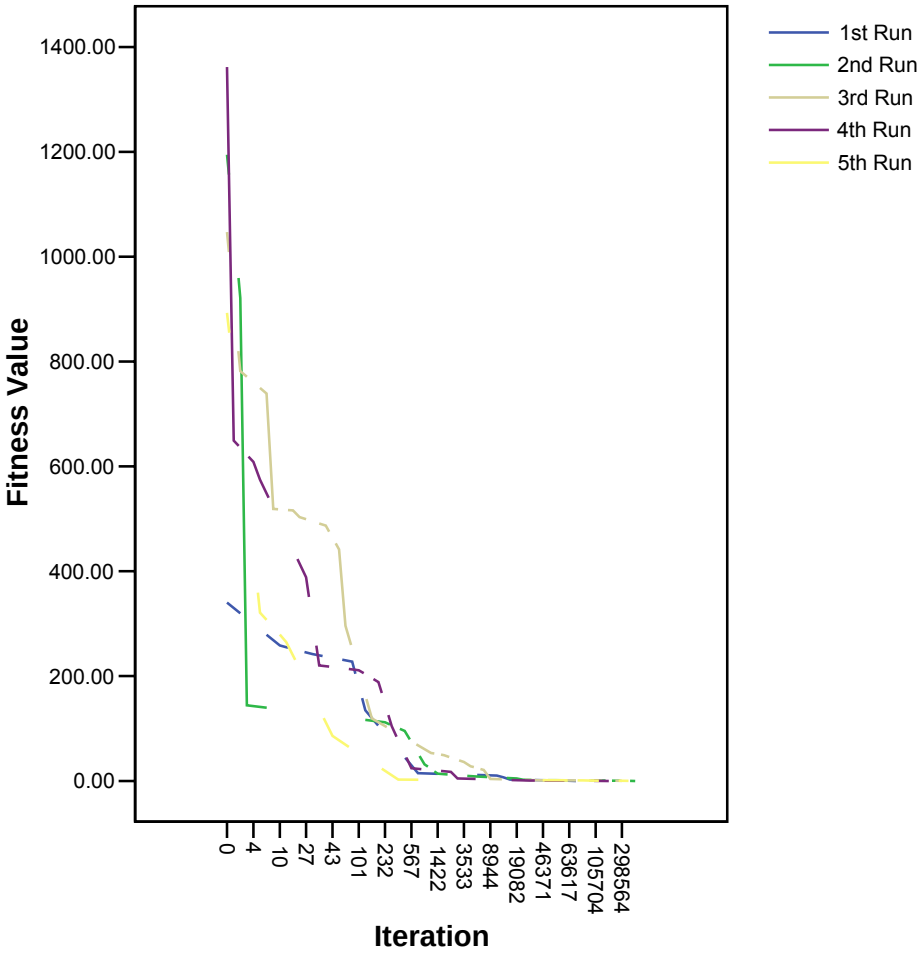


Figure 6.3: MC Search Convergence for the Schwefel Function with n=2

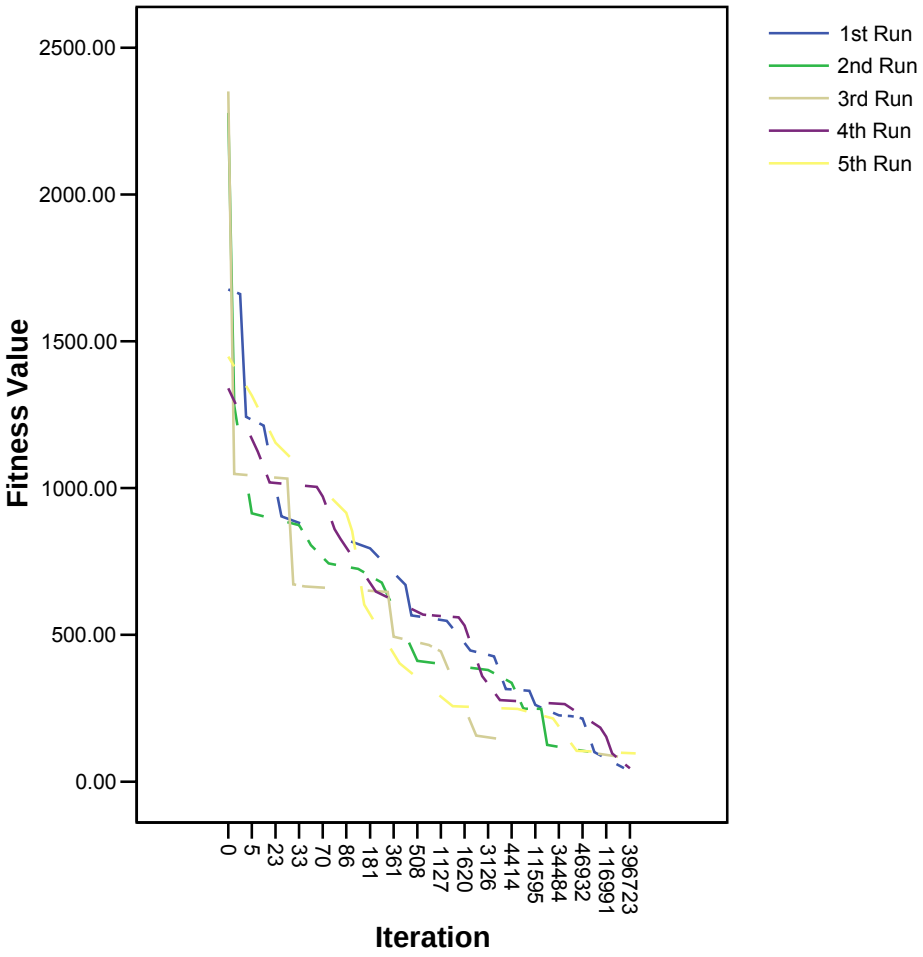


Figure 6.4: MC Search Convergence for the Schwefel Function with n=4

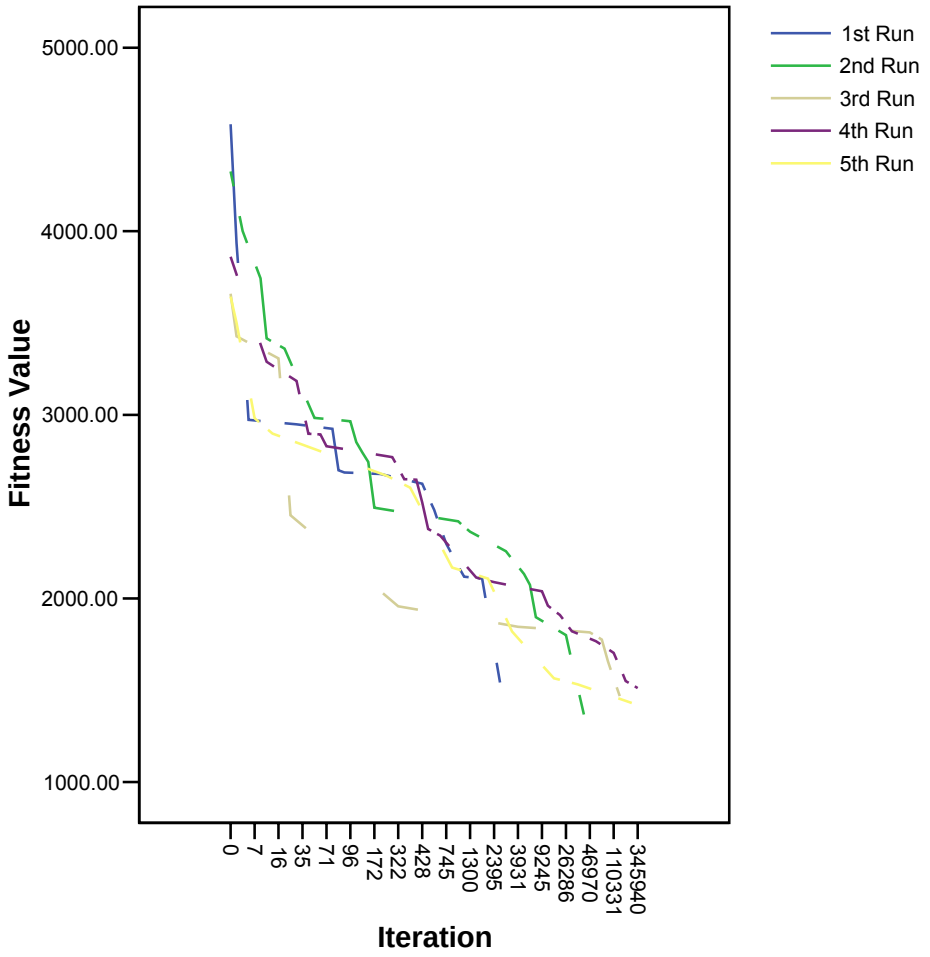


Figure 6.5: MC Search Convergence for the Schwefel Function with n=10

For $n = 2$, MC search is able to steadily converge and find a solution with a fitness value of less than 1.00 after about 100,000 evaluations. The convergence pattern becomes more irregular when n=4, with a minimum fitness value found in the 45.00-100.00 range after 300,000 iterations. Increasing the number of variables to n=10 further deteriorates the MC performance, resulting in fitness value in the 1,300.00-1,600.00 range at about 300,000 iterations.

The MC technique performs adequately in finding a good solution of the Schwefel function when only a small number of variables are used. This assertion is supported by the fact that the search consistently discovers better solutions as it progresses, and always terminates with at least a local minimum discovered. The search performance drops considerably when a large number of variables are used however.

6.2.1.2. Genetic Algorithm

Also similar to its Rastrigin function counterpart, the GA optimization for the Schwefel function uses a population of 500 individuals that evolve in 1,000 generations. As in the Rastrigin function search, the values of 0.6 and 0.03 are also used for crossover and mutation probability constants, respectively.

Figures 6.6, 6.7, and 6.8 shows the resulting convergence patterns of the search for n = 2, 4 and 10. Similar to the search on the Rastrigin function, the GA search on the Schwefel function converges rapidly when few variables are used. With n=2, the search converges quickly, with the absolute minimum found at under 70 generations or 35,000 evaluations. On average, a fitness value of less than 1.00 is found in just 10 generations or 5,000 evaluations. Such performance is considerably better than that of MC for the same number of variables.

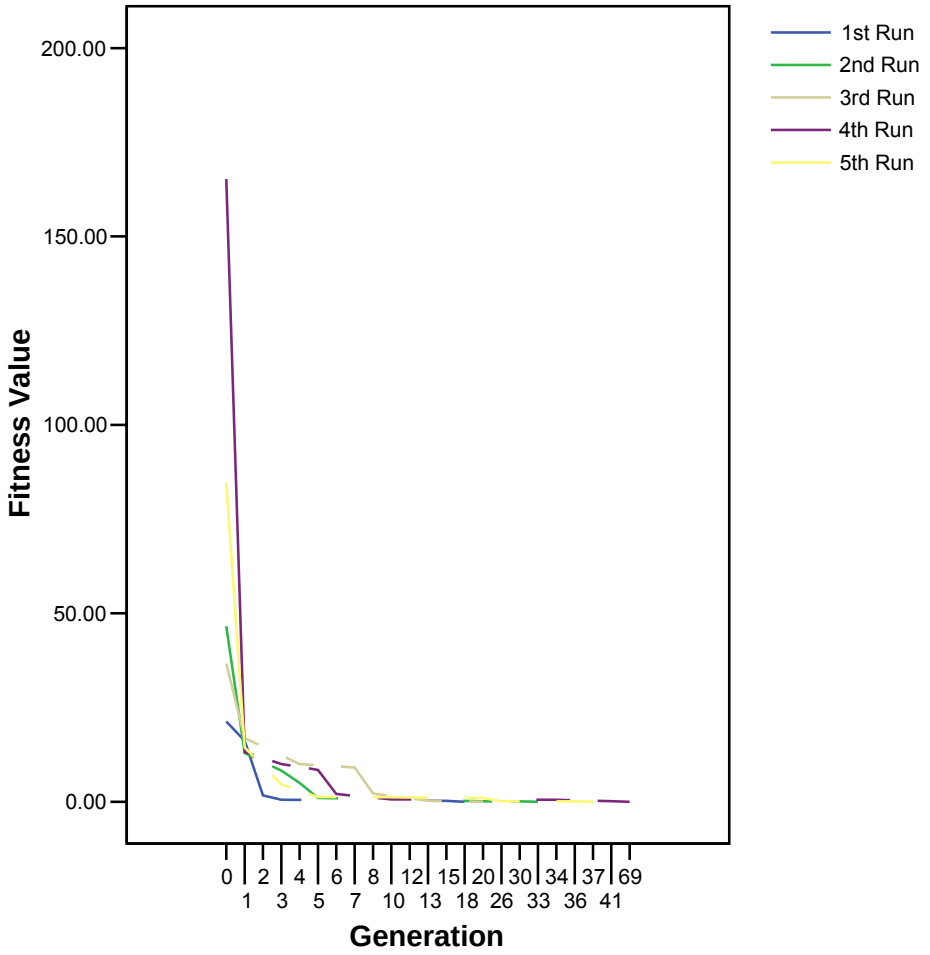


Figure 6.6: GA Search Convergence for the Schwefel Function with n = 2

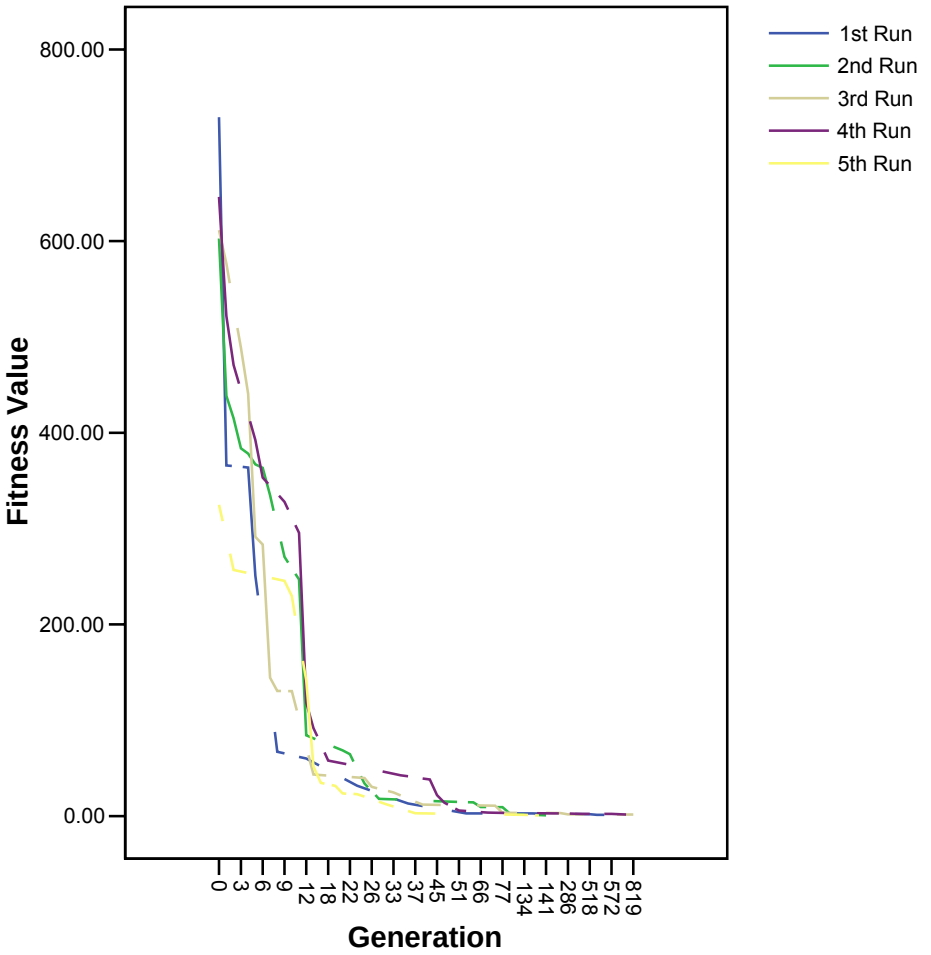


Figure 6.7: GA Search Convergence for the Schwefel Function with n = 4

The better performance of GA becomes evident for *n*=4 where not only the convergence is faster, but the result is also better. The GA is consistently able to find solutions whose fitness values fall within the 0.600-1.700 range in about 100 generations or 50,000 evaluations. This compares favorably to fitness values in the 45.00-100.00 range after 300,000 evaluations using the MC method.

Similar to the MC result, the GA fails to find the absolute minimum in all attempts when the number of variables is set at *n*=10. The GA is far more successful at finding better solutions however, with fitness value consistently falling within the 200.00-400.00 range compared to MC's range of 1,300-1,600.

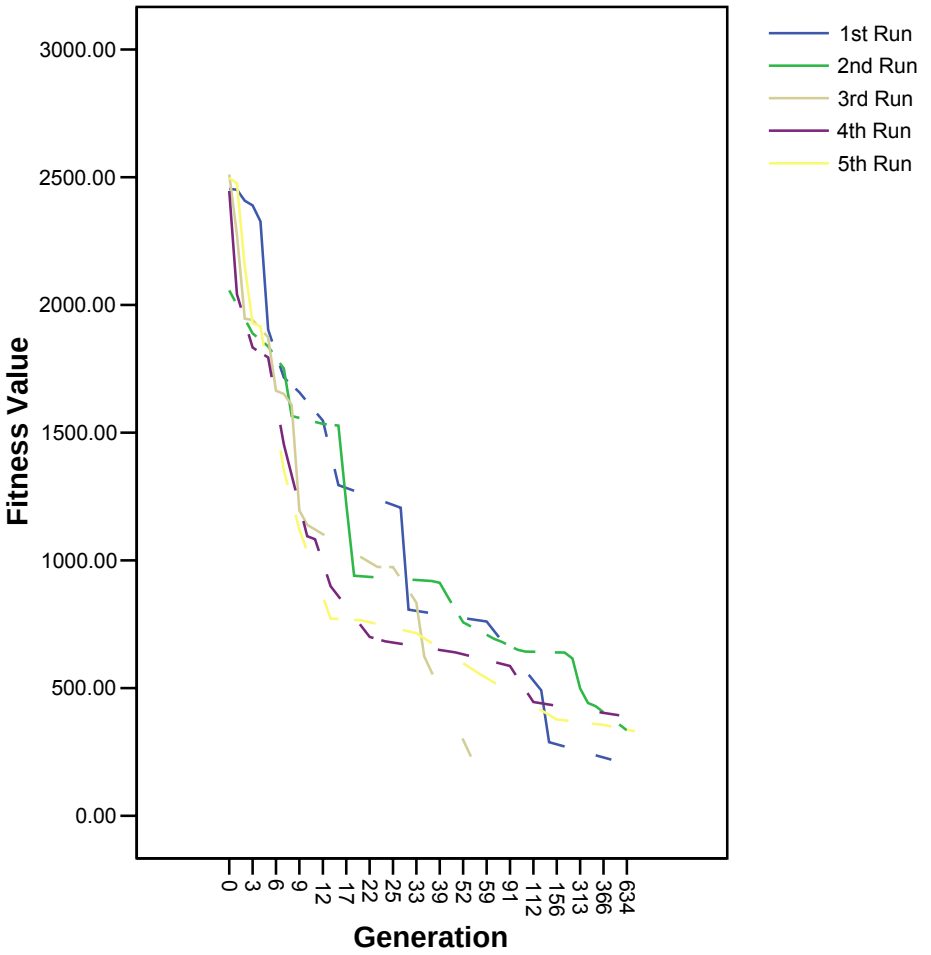


Figure 6.8: GA Search Convergence for Schwefel Function with n = 10

There are a number of possible reasons for the algorithm's failure to find the global optimum. The irregular distribution of the function peaks must be one major reason. The other is the relatively unsophisticated implementation of the GA used in this particular instance. Yet another possible reason is that the constants used for crossover and mutation probabilities may be suboptimal.

The results for the n=10 variation can be compared to a more mature GA implementation available in the public domain (Dolan, 2006). This GA, implemented in the Java programming language, is considerably more fully featured than the simple GA implemented as part of this research. With similar control parameters this implementation finds near-optimal solutions to the n=10 problem with an average of around 70,000 evaluations. This further supports the hypothesis that the implemented GA is struggling to solve the problem due to the lack of sophistication.

In any case, the GA does succeed in converging to at least a local optimum and terminates when a valid solution is found. It is also clearly much more effective than a random walk. Such qualities alone may be sufficient for the purpose of the experiments that follow. Should the GA not solve the MCPO problem sufficiently well, the results achieved can be used as a benchmark against which future, more sophisticated algorithms can be compared. It is important to bear in mind that as a rule the global optimum is not known in an actual MCPO problem. The capability of reliably converging towards a good solution is what can be realistically expected from the algorithm and may be sufficient to find a solution that is "good enough" tradeoff between the quality of the solution and the time required to find it.

6.3. Case 1: Simple Rectangular Layout

The first experiment involves the layout optimization of a 300x300 square container, with a 50x100 rectangle-shaped obstacle within as shown in Figure 6.9. The bottom left vertex of the shape is (50, 50). The optimization procedure seeks a solution with which 50x100 rectangular shaped stock panels can be used to cover the container area, using (0, 0) as the point of origin. The origin is outside of the area to be covered and this simple example allows the impact of this to be observed.

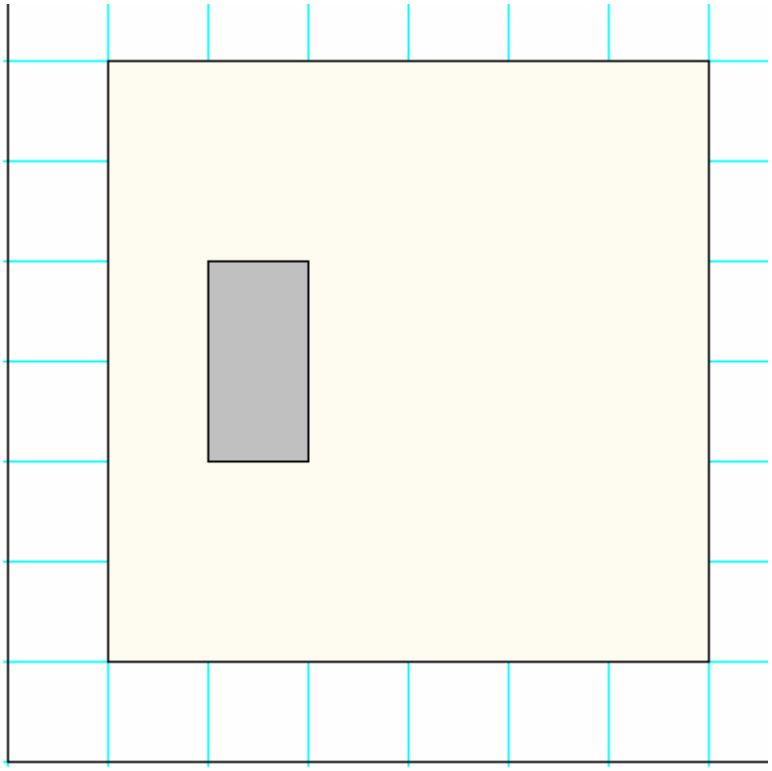


Figure 6.9: Simple Rectangular Container Optimization Problem

This is a trivial example, the purpose of which is to demonstrate that the optimization process is actually able to find a solution for a simple problem. Figure 6.10 shows the solution of the first part of the problem, whereas Figure 6.11 shows the solution of the second stage optimization using the greedy algorithm. Light blue color is used to indicate regular, whole panels whereas dark blue color indicates irregular panels.

The solution efficiency is defined as the container area divided by the available area provided by the stock panels. Tables A.1, A5, and A.9 in Appendix A show all three optimization algorithms consistently successful in finding a 100% efficiency solution. The simple rectangular container, however, is not typical. Solutions with less than 100% efficiency are the norm as subsequent experiments will show.

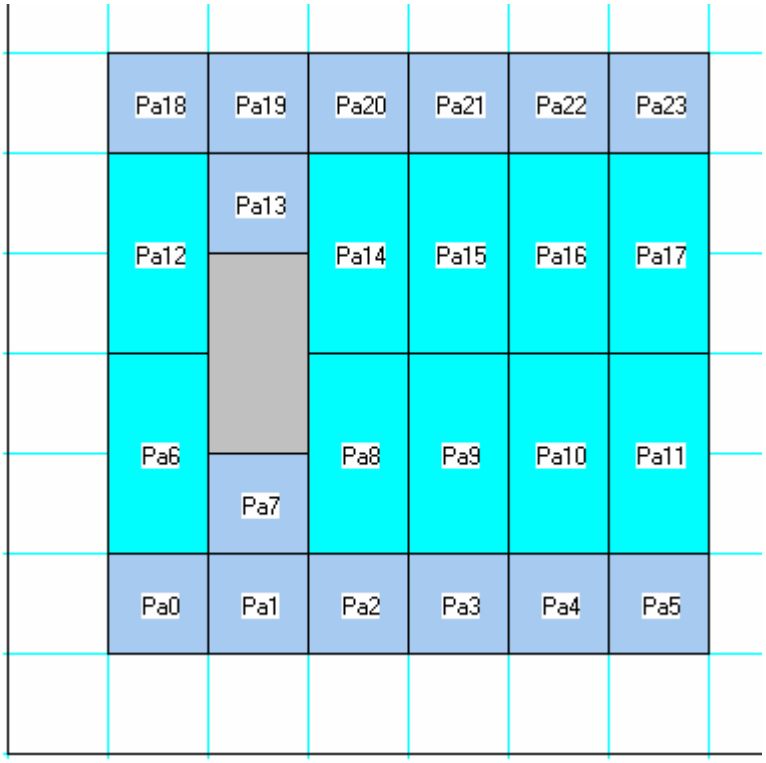


Figure 6.10: Panel Placement Solution for Simple Rectangular Container Problem

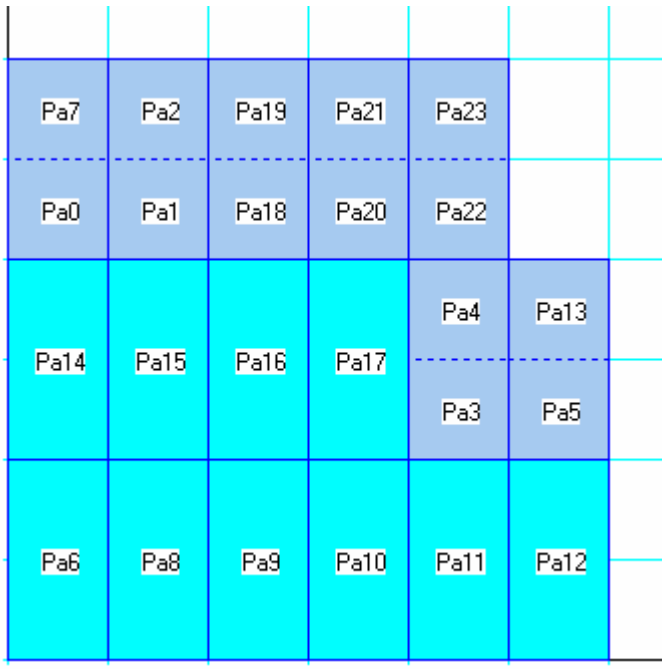


Figure 6.11: Nesting Solution for Simple Rectangular Container Problem

Despite the 100% material efficiency, it is clear that the solution for the simple rectangular container problem above is not ideal when point of origin (0, 0) is used. The absolute best efficiency is achieved when (50, 50) is used as the point of origin instead, as evident in Figure 6.12. Only entire panels are used in this case, implying not only 100% efficiency but also the complete absence of cutting the material.

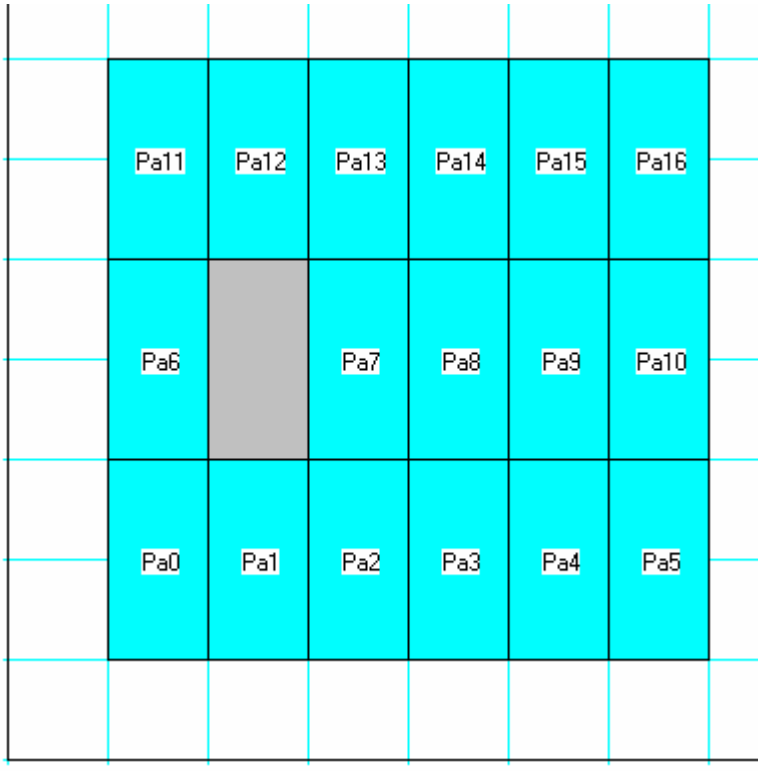


Figure 6.12: Best Nesting Solution for Simple Rectangular Container Problem

It has been explained in Section 6.1 however, that only a single point of origin (0, 0) is to be used throughout the experiments. Therefore to be consistent with the experiment strategy laid out early in this chapter, results such as that in Figure 6.12 are not to be considered any better than that in Figure 6.10. Future work will focus on the development of the geometric functions required to select a more appropriate origin based on querying the shape of the polygon to be filled to determine the best starting point. This would eliminate the need to conduct multiple optimizations to determine the best origin.

6.4. Case 2: Single Wall Layout

The second experiment involves the layout optimization of a single container with both convex and concave corners. As shown in Figure 6.13, the outline of the container takes the form of the wall at the side of a building. The shape has height and width of 450 and 350 units of measure, respectively. Assuming that the panels do not have grains or patterns, rotation at 90 degrees increments is allowed during the nesting process.

Two types of stock panels are being considered to generate the solution: one has the dimensions of 80x60, the other 54x80. Because the shapes can be easily scaled to their life-size equivalent, there is no need to map units of measure used in this example to the standards actually used in the building industry.

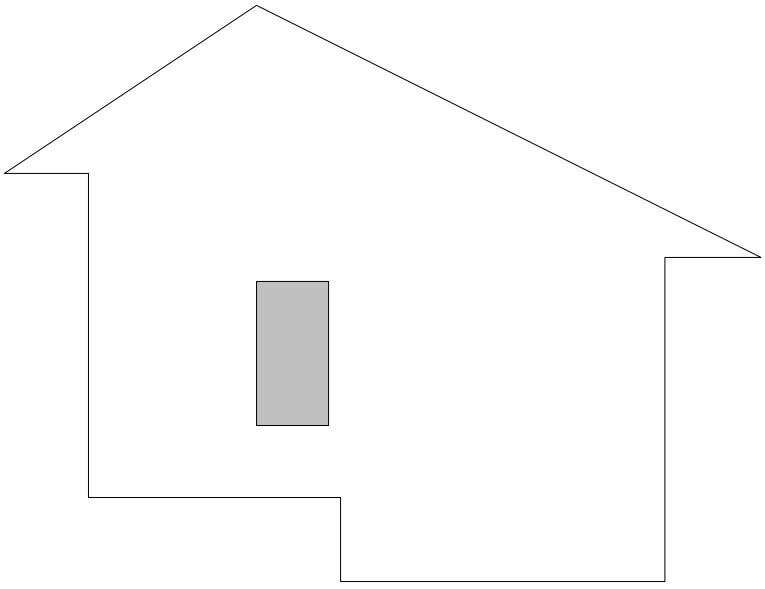


Figure 6.13: Single Wall Optimization Problem

The first stage solution is shown in Figure 6.14. As with the previous case, it is clear that the use of the (0, 0) origin is affecting the quality of the resulting solution adversely. Being able to identify, without human intervention, a better origin would in

this case lead to improving the quality of solution by reducing the cutting required to produce the layout, even if the number of panels were not reduced.

Such improvement would be most evident in the number of pieces required to be nested in the second stage. An example of a second stage solution is shown in Figure 6.15. Unlike the simple rectangle container problem however, it is not possible to achieve a solution with 100% efficiency.

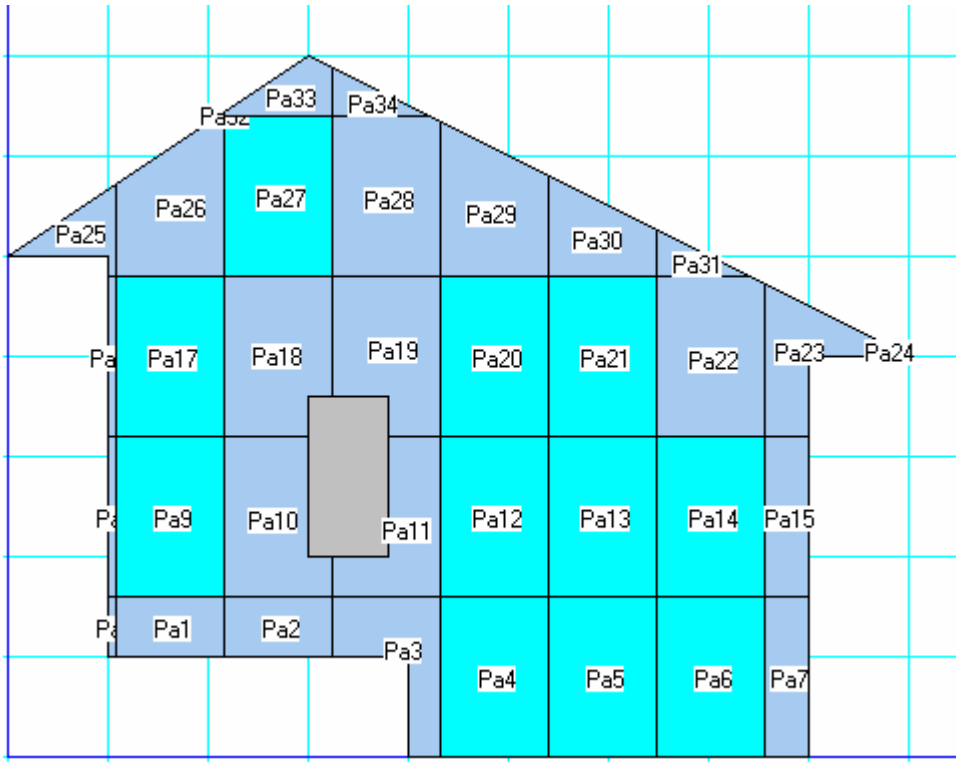


Figure 6.14: Panel Placement Solution for Single Wall Problem

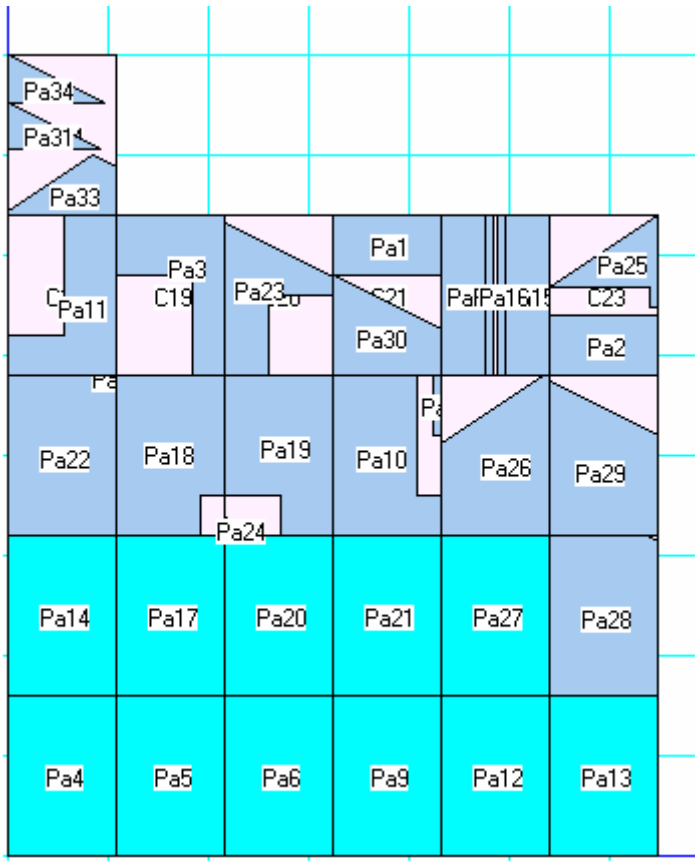


Figure 6.15: Nesting Solution for Single Wall Problem

Table A.2 in Appendix A shows various results obtained for single wall optimization problem when the greedy search algorithm is used. The achieved efficiency is higher than expected at 87.31% in most cases with the best at 89.30% when the first-fit strategy is used and 180 degrees rotation is allowed. Because greedy search is deterministic, the result is always identical for a given set of parameters. Therefore, only a single optimization run is executed for each parameter set.

As shown in Table A.6, optimization using the Monte Carlo method is less successful with efficiency ranging from 81.86% to 85.42%. The number of iterations and the flip probability are set to 10,000 and 0.2 respectively. Such numbers have been selected after a series of trial runs to reflect a perceived good combination of moderate-size search with a relatively low rate of bit mutation.

The Genetic Algorithm achieves similar performance as shown in Table A.10, achieving 85.42% efficiency for all parameter settings. In all cases, a population of 100 individuals is used to evolve in 100 generations. A crossover probability of 0.6 and a mutation probability of 0.1 are used, also after such numbers appear to be adequate in a

series of trial runs. In the absence of better methods for setting the parameters, this approach seems sufficient for our purpose.

Perhaps the most important finding in this experiment is that greedy search outperforms the two other algorithms despite it being incapable of recovering from premature convergence. The MC and GA on the other hand seem unable to capitalize their advantage in negotiating local optima. The net result is not only the greedy search being capable of finishing the job much faster (one nesting attempt instead of 10,000), but also with a better quality result. However, it is important to point out that the total number of panels required for the single wall is the same even though the utilization of material is lower. The savings in material become significant as the application of the approach is extended from the optimization of the single wall, to the room, and ultimately to the whole building. Potential exists to significantly reduce the total amount of material required if the approach can be applied to the optimization of whole buildings.

Another important finding is the impact of rotating the pieces to the efficiency of the final nesting result. More freedom of altering the orientation of the pieces does not automatically translate to a more efficient nesting solution, as consistently indicated in tables A.2, A.6, and A.10.

Finally, the best-fit placement strategy does not guarantee a better solution than the first-fit strategy, both in terms of area utilization and shared edge length. This finding is rather unexpected, because the best-fit strategy has been expressly aimed at maximizing the length of the shared edge.

6.5. Case 3: Simple Roof Layout

The third optimization problem is taken from one of the sample problems used by Sibley-Punnett and Bossomaier (2001) for their roof layout optimization.In this particular case, multiple containers are used. The simple roof layout differs from the previous problems by the multiple containers involved. Figure 6.16 shows the top view of the roof. Sections of the roof have been labeled 1-4 to assist identification. Figure 6.17 shows the sections the same roof taken apart and laid on a flat surface.

At this point, it is important to note that while the work of Sibley-Punnett & Bossomaier provides examples of the actual roof layout optimization problems, no specific details are given regarding the performance of their algorithms on the specific cases. For this

reason, it is not possible to make direct comparison between the results obtained by these researchers and the results of this experiment.

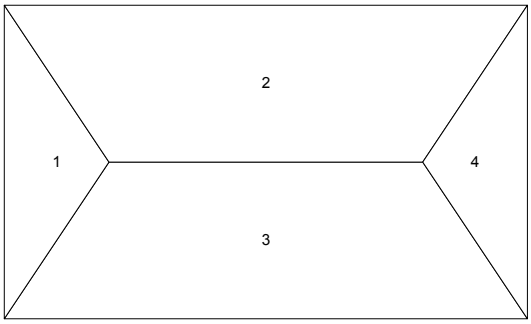


Figure 6.16: Simple Roof Viewed from Above

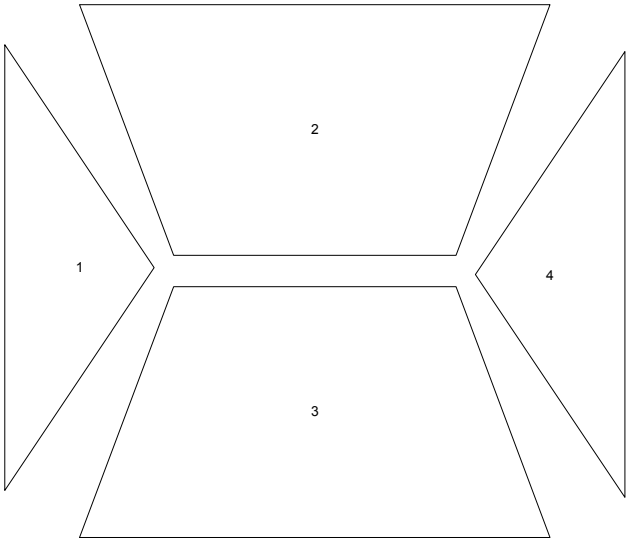


Figure 6.17: Sections of the Simple Roof

The simple roof layout also differs from previous problems from the material constraints of the sheets used. Sibley-Punnett and Bossomaier assume that the panel used takes the form of corrugated iron or similar material. The implication is that the panel has distinct upper and lower sides, rendering flipping illegal. The material also has ridgelines and guttering that dictates that only 180 degrees rotation is allowed.

Yet another constraint to be taken into account in this optimization problem is the overlap between adjacent pieces when installed on the actual roof. Such overlap exists in the actual roof construction both for aesthetic reasons and to prevent leakage. In this example however, such overlap is ignored to avoid unnecessary complication.

Figure 6.18 shows the first stage solution with the four sections of the roof laid side by side. The nesting layout in Figure 6.19 shows the corresponding second stage solution.

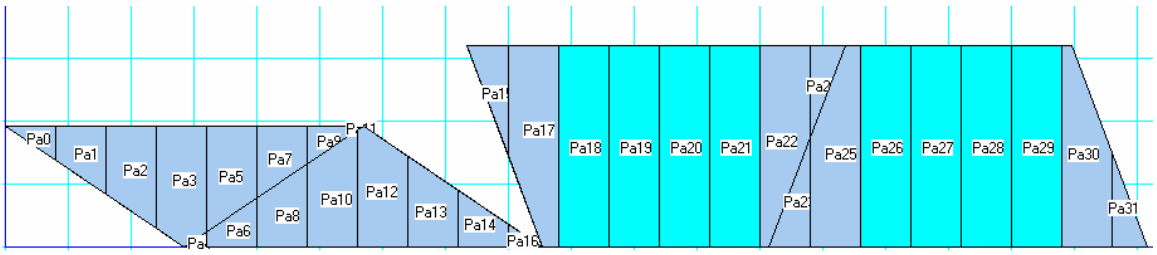


Figure 6.18: Panel Placement Solution for Simple Roof Problem

Table A.3 in Appendix A shows solutions with very high efficiency of 90.51% and 95.84% being achieved by greedy search. The highest efficiency is obtained when 180 degrees rotation is allowed.

In stark contrast, Table A.7 reveals that the Monte Carlo method is only able to produce solutions with efficiency ranging from 74.05% to 81.46%. All the parameters have been set identical to those in the single wall layout problem previously discussed.

The Genetic Algorithm yields even more disappointing results, achieving efficiency of only 74.05% to 77.58% in its solutions as shown in Table A.9. All GA parameters have also been set identical to that in single wall layout problem.

The superiority of greedy search becomes much more apparent in this experiment. Neither the MC nor GA is able to create solutions with efficiency that matches even the lowest of that generated by the greedy search.

As in the previous example, the total number of panels required is the same in each case, apparently not offering the savings that should be possible given the extension to multi-surface optimization. In the majority of the cases, better efficiency is obtained when rotating the pieces by 180 degrees is an option and even better results are possible if free rotation is allowed. The constraints on the problem due to the material are limiting the ability to reduce the number of panels required. As in the previous experiment, the best-fit placement strategy does not provide direct help in achieving better overall efficiency. It does consistently yield better results in terms of shared edge length, however.

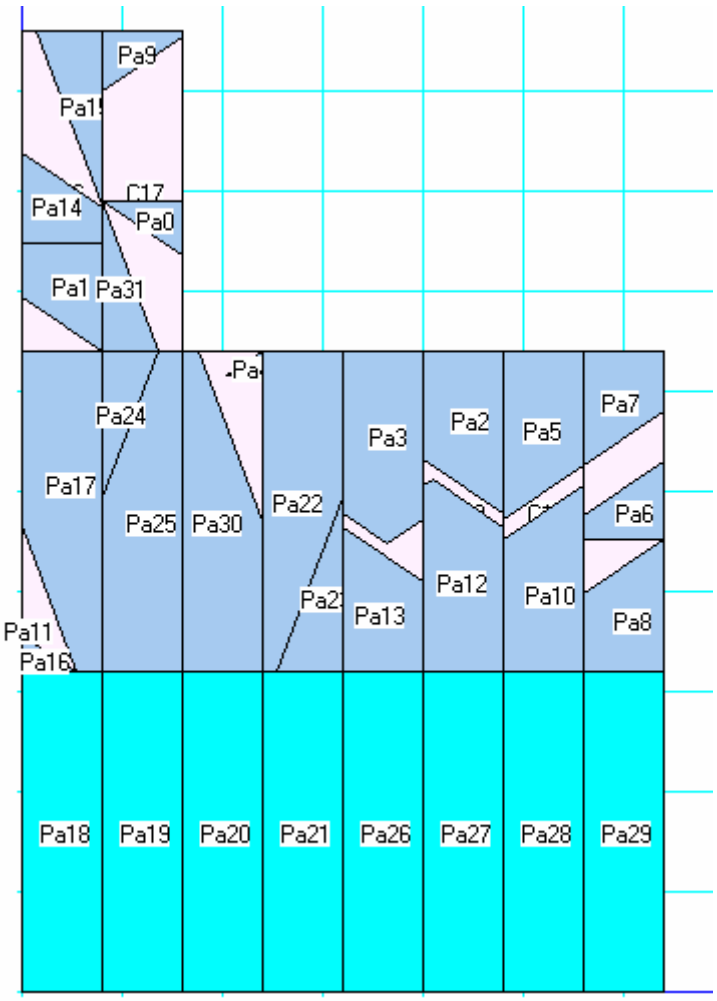


Figure 6.19: Nesting Solution for Simple Roof Problem

6.6. Case 4: Complex Roof Layout

The fourth and final experiment uses another example from Sibley-Punnett and Bossomaier (2001). In this case, a complex roof consisting of multiple sections is used. Unlike the simple roof example, there are twice as many sections of greatly varying sizes that make up the roof. Concave shaped sections are also used, as opposed to all-convex shapes in the simple roof layout problem. Other roofing material-specific constraints still apply however. Figure 6.20 shows the top view of the complex roof.

Similar to the previous case, results obtained from this experiment cannot be directly compared to that acquired by Sibley-Punnett & Bossomaier due to the lack of the required data. Nonetheless, the example is adopted for experiment because of its value in representing more complex actual layout optimization problem.

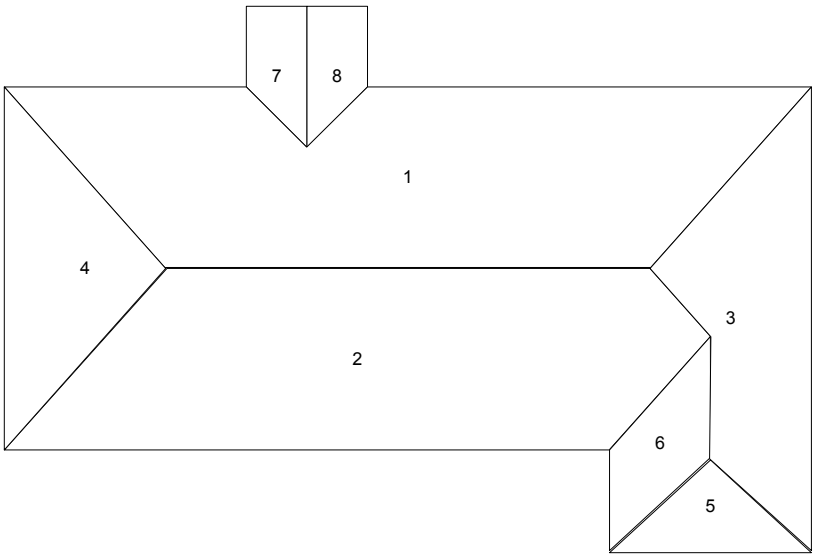


Figure 6.20: Complex Roof Layout

Figure 6.21 shows the first stage solution for the problem. Because of the large size of the roof, only segments 3, 5, and 6 are completely visible. The nesting plan which is part of the second stage solution is given in Figure 6.22.

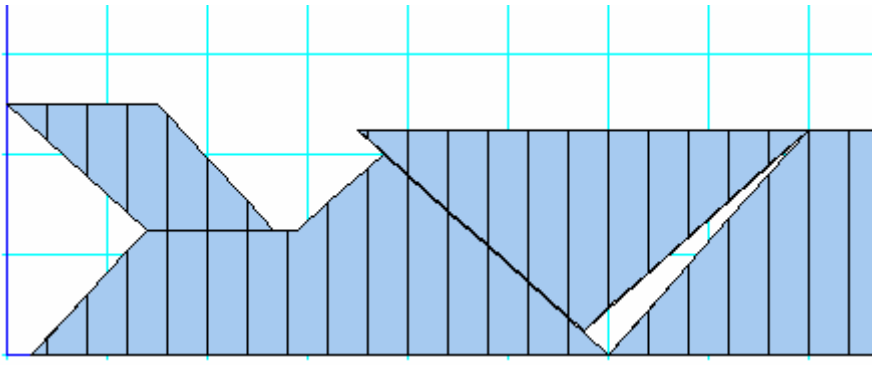


Figure 6.21: Panel Placement Solution for Complex Roof Problem

As in previous experiments, the greedy search clearly outperforms the other algorithms in terms of computation time and solution efficiency. Table A.4 in Appendix A reveals very high efficiency rates from 95.50% to 96.99% being achieved using this method.

None of the optimization results produced by the Monte Carlo method achieve comparable efficiency. Table A.8 shows the efficiency ranging from 62.70% to 72.18%, nowhere near that achieved by the greedy search.

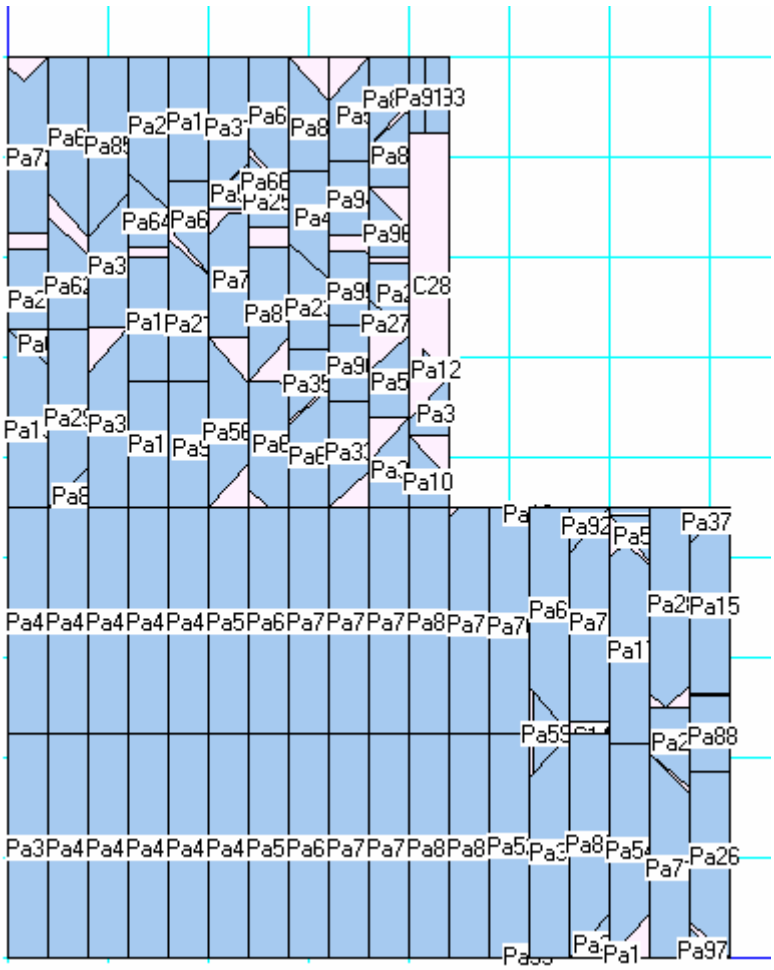


Figure 6.22: Nesting Solution for Complex Roof Problem

The Genetic Algorithm performs somewhat better than the Monte Carlo technique this time. Solution efficiencies ranging from 70.54% to 73.03% have been attained, which is higher than those from MC even though still far behind the greedy search.

Similar to the preceding experiment, better efficiency is generally achieved when 180 degree rotation is an option. There is an exception with the GA however, where the best efficiency is found when no rotation is allowed. Again the material constraints are limiting the potential for reducing the total number of panels required and another similarity found with the previous experiment is the longer shared edge consistently obtained when best-fit placement strategy is used. The best-fit strategy also seems to contribute towards better overall efficiency, although exceptions still exist.

7. Discussion

Up to this point, the investigation of two-stage layout optimization has covered a number of disparate subject areas. A distinct characteristic of the research is that a big portion of the effort has been spent designing, constructing, and validating the software in the framework provided by the chosen research methodology. The experiment with actual data, despite its importance and the amount of effort involved, became less prominent in view of the overall work required to maintain rigor of research.

This chapter has been organized to report the findings at the same proportion. A sizeable amount of space has been devoted to discuss the software development aspect of the research, in which quite a number of important findings have occurred. The remainder of this chapter is divided into three sections: the first two deal with the software design and implementation, whereas the third discusses the findings of the actual experiments.

7.1. Research Methodology

This research was conducted using the System Development Research Methodology (SDRM) (Nunamaker & Chen, 1991) which has provided a governing framework for structuring the key activities. Whilst other constructive methodologies exist, the strength of SDRM is that it explicitly requires the selection and use of an appropriate software development methodology.

During the early exploratory phases of the research a great deal of effort was spent investigating and implementing the geometric algorithms required for developing the model of the MCPO problem. The alignment between research methodology and software development methodology provided constraints on these activities that ensured the modularity of the code and the ability to develop different modules at different rates without impacting on the overall success of the project.

7.2. Design

From a number of software development models commonly practiced, the *rapid application development* (RAD) approach had been selected. RAD was deemed suitable when evaluated against the key characteristics of the project, such as the lack of detailed initial requirements from the user and the strict time line within which the project must be completed. The model proved effective in allowing incremental development of the software to take place. The incremental development in turn provided a solid base from which the problem space could be better studied and understood. Finally the better

understanding of the problem made it possible that informed decisions were made regarding its solution.

While RAD provides the outline for the development activities, the cornerstone of actual software design is the *process modeling* where the *data flow diagram* (DFD) is used. Process modeling provides a systematic way of both decomposing the problem and designing its computer-based solution. The resulting DFD identifies all the critical components of the software and how they all interact. The subsequent actual development task was essentially translating the components from their abstract form in the diagrams to the actual program code.

In conjunction with process modeling, the various forms of information flowing within the system are captured in the *data modeling*. The end result of data modeling is the *data dictionary*, in which all types of information packet are defined and standardized. While the DFD provides a template for constructing the actual objects, instructions, and routines in the program code, the data dictionary serves as the equivalent of the data structures that are manipulated by those routines.

Although they are indispensable for analyzing the problem and designing the solution, the DFD and the data dictionary are not without weaknesses. More than anything they did not in any way provide an error-proof mechanism of constructing the actual software. The RAD model indeed lacks the capability of automatically generating the program code, which is the main feature of more modern software modeling tools such as 4GL. All the intelligence built into the software code has been entirely written by hand. As a result, there can never be a guarantee for faithful translation of the abstract design to the actual program code. The information flow and transformation mechanism devised for the system, however, remains to a high degree mapped to the actual code.

7.3. Implementation

Maintaining software modularity has always been practiced throughout its development. As a defining characteristic of RAD, modularity allows various parts of the software to be added, updated, or in some cases removed, with minimum impact to other parts of the system. Such flexibility in turn allows those parts to be coded, tested, and debugged in isolation to ensure the development of the software as a whole can reliably take place at a rapid pace.

The notion of modularity can be found both in the program logic and the organization of its source code. Modularity in program logic is realized by the use of object-oriented programming (OOP) language and environment. Programming language-specific objects are used extensively to encapsulate logical entities of the software, effectively turning them into modules, to make clear separation between various data packets and the processes associated.

Furthermore, the program source code has been organized into a number of disk files to make closely related objects easy to locate. While file organization does not have any impact whatsoever to the correctness or efficiency of the executable program, it has enormous value in maintaining the efficiency of the coding effort.

In the absence of definitive requirements from the user, the input and output of the software take the form of disk files. Run-time parameters supplied through the program user interface make for additional input. Another major form of output is on-screen visualization, with which the two dimensional objects are presented to the user the way they would appear in real life.

When external files are used either as input or output, the associated data is always organized in a hierarchical structure. This approach offers the capability of packing a collection of data of dissimilar formats into a convenient single container. Such a container is physically implemented conforming to extended-markup language (XML) standard, which is a data exchange protocol widely accepted in the computing establishment.

Although the XML data packet is commonly criticized for its comparatively low information content and lack of inherent security features, it proved to be well suited for this project's requirements for several reasons. Firstly, being a text based standard it allows the data to be inspected and edited by hand using a general purpose text editor, eliminating the need of a specialized editor. Secondly, wide support of the XML standard has resulted in the easy access to third party XML-handling components that can be integrated into the software. Finally, because the actual disk files containing the data are typically small, the low information content has not been found to be an issue. Data security has never been considered relevant to this project, making it a non-factor when it comes to determining the data exchange format.

Various data structures are used to internally represent the pieces of information in the computer memory. Depending on the complexity of operations associated to each data structure, they can be divided into two types: the active and passive data objects. Active data objects are used to represent data entities associated to a wide range of activities and hence tend to be processed individually. Active data objects consist of data and methods, and exhibiting encapsulation and polymorphism, the characteristics of true objects in OOP terms. The active data structure is quite powerful, yet delicate to handle and tends to consume system resources when incorrectly managed.

In contrast, passive data objects contain only data. Moreover all their contents are fully exposed and hence become subject to external processes. Passive data objects are a key feature of procedural programming language such as ANSI C or Pascal (implemented as *struct* and *record* types respectively), and unknown in a pure OOP environment. Passive data objects are much simpler and straightforward than their active counterparts, making them suitable for batch-type processing. A two-dimensional vertex for example, can be easily represented by a passive data structure containing only X and Y values. An array of vertices makes for a polygon, which can be easily passed between procedures when a complex task is performed.

Optimization of two-dimensional problems proves to be much more complex than optimizing problems consisting of two scalar variables because of the geometric calculations involved. Two-dimensional layout optimization problems are characterized by a multitude of geometric operations to be performed to find the solutions. Consequently a considerable amount of effort has been spent in building geometric models and writing geometric calculation routines before optimization algorithms could be realized.

A number of geometric operations stand out because of their complexity. Polygon triangulation, polygon overlap detection, polygon linear cut, and polygon clipping are geometric problems whose solutions in the literature are either non-existent or excessively complex for the scale of this project. To resolve this problem, novel solutions have been devised and empirically proven to be applicable.

These solutions have been found to work correctly in all tested cases and displayed with adequate performance to justify their use. Also the program code is flexible enough to easily accommodate other solutions for those geometric problems later. While there is

no plan to implement other solutions in the immediate future, this flexibility further justifies the effort spent maintaining modularity of the software during its development.

Three search algorithms have been selected to perform the actual optimization: the greedy search, the Monte Carlo technique, and the Genetic Algorithm. It was planned to model the optimization problem in such a way that all three could be used in both stages of the solution. With the implementation of those algorithms in place, it would only be a question of determining parameters to be optimized during each stage before those optimization algorithms can be used. While the parameters for the second stage problem are quite complex, the parameters for the first stage are very simple: only the X and Y coordinate for the point of origin. Unfortunately there are two major problems that prevented the actual implementation of this plan.

The first is the greedy algorithm implementation, which only works effectively when it has been extended so it can deal with vector parameters instead of just scalar parameters it was originally designed for. The result is a heavily modified greedy algorithm that is efficient in solving nesting problem, but incredibly awkward to use with simple scalar parameters.

The second is the enormous computational cost associated with the second stage solution, particularly when simulation-based solution algorithms are used. Only the greedy algorithm is efficient enough to find a solution for the second stage problem to make it appropriate as part of fitness evaluation of the MCPO problem. The other two are slower typically by the order of two or three, requiring execution time of up to a few hours for a single moderate sized second stage problem. This level of computational expense is unacceptable for the industry partners of the project, and considerable further work will be needed to develop heuristic algorithms that increase the quality of solution in an acceptable timescale.

Because of the above problems, the optimization algorithms are used only to find the second stage solution. The parameters for the first stage solution are to be specified explicitly by the user.

The first stage task is to partition the container area both vertically and horizontally using the stock panels as a template. The result is a set of small pieces that either fit perfectly to a stock panel because of their identical size and shape, or smaller with

possibly an irregular outline. The irregular panel becomes the input for the second stage optimization.

For the second stage optimization, the greedy algorithm is relatively simple to code. The algorithm attempts to fit a piece with the largest surface area into the vacant space in the container at every turn. When such a piece is found, there is typically more than one valid way to position and orient the piece within the container. The first-fit strategy simply accepts the first valid solution, resulting in a solution found faster but with potentially sub-optimum efficiency.

The best-fit strategy is meant to seek better solution by considering all valid placement options before selecting the best. Naturally, the best placement is the one that conserves the most contiguous space. This is not possible to implement however, due to the lack of a reliable means of measuring such a quality. As an alternative, the length of shared edge is used instead. The rationale is that even though contiguous space is not conserved, more shared edge would mean less effort in cutting the pieces off the material. This is typically one of the concerns of the targeted end user, builders, who wish to simplify installation as much as possible. It was also believed that maximizing the shared edge may have the side effect of making the piece less likely to become obtrusive in the vacant space.

Modeling the second stage parameters into a binary string is a major issue in the implementation of both the Monte Carlo and the Genetic Algorithm. Unlike other layout optimization problems commonly found in the literature, the second stage problem requires a number of pieces to be nested in multiple containers. The optimization task therefore becomes twofold: to minimize the number of containers and to map the pieces to those containers in a way that minimizes the containers' vacant surface area.

No suitable model has been found to satisfactorily map all the parameters to a single binary string. Only a subset of the parameters is used in the model that prevailed. In this model, the pieces are mapped to clusters to indicate to which panel they will be nested. Greedy search is then used to actually put those pieces together within the assigned panel.

Although the cluster-based model is workable, it has an important issue of handling invalid clusters where the corresponding panels cannot accommodate all pieces assigned to them. A correction strategy of redistributing surplus pieces is used to split an invalid

cluster to multiple valid clusters. The actual software uses the append at tail strategy, where new cluster is added at the end of the list to hold excess pieces from the invalid cluster. It is acknowledged that such strategies may have an impact on the effectiveness of the GA implementation. Other strategies, such as the redesign of the penalty function, could be considered in the future if this approach does not yield adequate results.

Whilst the Monte Carlo technique was very easy to implement because it only needs to do a random walk in the search space to find the solution, the Genetic Algorithm is more complex because of the number of sub-processes involved, of which adopting different strategies for which may have significant impact on the overall performance of the algorithm. A version of a GA as implemented by Goldberg (1989) is used to avoid unnecessary complications. Again, it is acknowledged that this implementation may lack the sophistication required to solve complex problems.

7.4. Experimental Results

The MC and GA implementations have been subjected to a series of test problems to verify their correctness. The purpose of the procedure was to ascertain that the logic of the algorithms remains fully preserved in the program code. Such verification is important because unlike the deterministic greedy search where the solution for a certain problem is singular, MC and GA contain stochastic elements that make solutions vary between runs and the exact process non-repeatable.

The verification problems took the form of continuous multi-variable mathematical functions. Such functions are characterized by the existence of multiple peaks in their ranges, which present local optima the search algorithm must negotiate in its attempt to find a global optimum. There are numerous such functions available in the literature, of which the Rastrigin and Schwefel functions have been selected. The optimization task of both was to find the global minimum.

In general, both the MC and GA methods are proven able to consistently find solutions with improved function values as the search progresses. When only two variables are used ($n = 2$), both algorithms succeeded in finding the global minimum after the same number of evaluations. The GA has a clear performance advantage over MC for minimizing the Schwefel function with $n = 4$. In this case the GA was able to find the global minimum with a relatively small number of evaluations, whereas MC only managed to find a local minimum after six times as many evaluations.

Both algorithms failed to find the global optimum for the Schwefel function with $n = 10$, although the GA consistently achieved better solutions compared to that of the MC. This result is not surprising since there are 1024 possible values for each variable, making the total number of possible solutions 1024^{10}. The Schwefel function, with $n \geq 10$ is considered a complex function, of comparable difficulty to solve as many real world optimization problems. Clearly a much more advanced implementation of GA is required to effectively solve a search problem of that magnitude.

The algorithms' obvious lack of power in solving optimization problems involving multiple variables may have a profound impact on the actual software performance solving the MCPO problem. This is especially true since the second stage problem of MCPO typically deals with large numbers of pieces, each being mapped to an optimization variable. The problem is further compounded by the computational expense associated with each fitness evaluation. Nevertheless, the verification tests served their stated purpose and the correctness of the algorithms is confirmed.

The actual experiments were conducted with various MCPO problems with increasing complexity. More straightforward problems were used to provide empirical proof that the software actually performs the way it was designed and all the search algorithms apply their underlying logic to find this domain-specific solution. More complex problems were used to gain insight into comparative performance of the optimization algorithms, particularly in terms of computation time and solution quality. Solution quality is defined by the efficiency of material usage (or minimization of wasted material) and the length of shared edges (indicating the effort of cutting the pieces off the stock panels).

The greedy search proves to be very effective by consistently outperforming other algorithms on both accounts. Greedy search has a decisive advantage in computation time because the task of constructing the solution using this method is equivalent to just one fitness function evaluation on MC and GA. It also converges to a local optimum every time, of which the efficiency is always higher than that of MC and GA.

Table 7.1 shows the solution efficiency of each optimization algorithm in more detail. The figures have been averaged from experiment results given in Appendix A. As mentioned above, the key parameters of solution efficiency are the material usage and the shared edge length. The number of irregular panels is included to indicate the complexity of the second-stage optimization.

	Irregular Panels			Material Usage			Shared Edge Length		
	Greedy Search	Monte Carlo	Genetic Algorithm	Greedy Search	Monte Carlo	Genetic Algorithm	Greedy Search	Monte Carlo	Genetic Algorithm
Simple Rectangle	14	14	14	100.00%	100.00%	100.00%	2450.00	2450.00	2450.00
Single Wall	23	19	19	87.81%	84.98%	85.42%	2891.58	2397.09	2357.71
Simple Roof	24	24	24	92.29%	77.05%	76.40%	1733.93	1613.83	1636.32
Complex Roof	129	129	129	96.00%	65.95%	71.23%	9714.97	9387.27	9516.96

Table 7.1: Summary of MCPO Solution Efficiency

Multiple stock panels of varying size have been provided for each layout optimization problem. It has been observed that stock panels of smaller size would typically yield better efficiency, though in reality may suffer from higher installation costs. The single wall problem seems like an anomaly where both MC and GA had found solutions of better efficiency with a stock panel larger than that used by greedy algorithm, as indicated by the number of irregular panels. Recall that an identical point of origin at (0, 0) is used for all cases, meaning the number of irregular panels resulting from the first stage solution is identical for a given stock panel dimension. The lower efficiency of the solutions, however, indicates that the decision to use a larger panel was sub-optimal.

Similar to the validation functions, the performance difference between MC and GA is only slight when only a small number of pieces are involved. Both are able to find reasonable solution, compared to that achieved by greedy search, with only marginal difference in efficiency.

When a larger number of irregular panels are involved, such as the case with a complex roof, the GA performance advantage over MC becomes evident. Much in the same way GA obtained better solutions than MC did in solving the Schwefel function with $n = 10$. At the same time however, the superior performance of the greedy search over both becomes even more pronounced.

Such a finding naturally raises a question of why such a crude algorithm can perform so much better than its much more sophisticated counterpart. Especially when compared to a GA, which is widely accepted as a powerful tool for solving the multi-variable optimization class of problems to which the second-stage problem of MCPO belongs.

Because the optimization algorithms have been implemented as integral parts of a computer application solving real rather than hypothetical problems, there is a number of contributing factors to be considered for an answer. The first is the relatively low level of sophistication possessed by the GA implementation. As discussed in Chapter 5, the implementation has been based on a simple variant of GA once coded for the benefit of students and researchers new to the subject. Such an implementation is characterized by a single crossover point, pair selection for breeding exclusively based on fitness value alone without regard to the actual bit patterns in the chromosome, and a lack of elitism. These deficiencies alone may be directly responsible for the algorithms failure to solve complex problems such as the ten-variable Schwefel function.

The second fundamental problem with the use of a GA in solving MCPO problem is the parameter modeling, which also applies to the MC method. As the analysis in section 5.5.2.2 demonstrates, inter-dependencies exist between the number of stock panels required and the actual nesting result for a given stock panel. The latter further introduces the problem of handling the invalid cluster, of which the attempt to put together pieces allocated for a stock panel within its boundaries is unsuccessful.

The concept of clustering is used in the prevailing model to address the problem of mapping the irregular panels to an undetermined number of stock panels. Whilst the model solves this particular problem quite well, it completely disregards a host of crucial parameters to be solved in the individual nesting tasks. Relegating the actual nesting of irregular pieces from the clusters to individual stock panels to greedy search has been done as a pragmatic measure taken in the interests of generating a valid nesting layout at minimum computational cost. Special provisions were also needed to effectively deal with invalid clusters.

By only partially solving the second stage problem, the MC and GA optimization have no direct impact on the final result. It may also be the main reason why the efficiency of the MC and GA solutions is only marginally different in the majority of cases. GA solutions only become visibly better than those of MC when a large number of irregular pieces are involved. This is consistent with the result of verification procedure using the Schwefel test function, where the GA performs noticeably better when more variables were used. The most likely explanation is that the performance disparity between the two algorithms is such that relegating the rest of the nesting task to an external entity does little to hide the difference in the final result.

In any case, the current parameter modeling has been found far from ideal. It is quite possible that a better model may realize the true potential of the GA in solving MCPO problems. Constructing such a model however is beyond the scope of this thesis. Further interest in achieving better MCPO solutions using the GA may warrant future study in this area. On a smaller scale, various aspects of the current model, such as better realization of the best-fit strategy and invalid cluster handling, can also be subject to more thorough study.

Lastly, the nature of the actual data itself may contribute to the efficiency of the solution. Take the simple and complex roof problems for instance. The simple roof has smaller search space, allowing the heuristic algorithms to find the good solutions in a

relatively few evaluations. The complex roof on the other hand, presents a much larger search space, requiring increased number of evaluations while making it less likely for those algorithms to find good solutions. At the same time, the complex roof has larger surface area. In effect, the ratio of the surface area of the stock panel with the container is smaller in complex roof problem than it is in simple roof counterpart. Bearing in mind that smaller stock panels tend to produce more efficient results, it is understandable that the result obtained by the greedy search is better on complex roof problem. This effect, however, is negated by the increased search space in the case of heuristic algorithm. The increased difficulty is reflected by the lower efficiency obtained by the MC and the GA on solving the complex roof problem compared to that of simple roof.

From the user perspective, the experiment results reveal that the use of novel optimization algorithms such as MC and GA has not been justified at the current stage of the software maturity. Employing the greedy search is the most logical choice for solving MCPO problems due to its low resource requirements and high quality solutions. However, it is important to point out that all three methods are finding solutions that have better material utilization than those created by a builder.

As a commercial application, the MCPO software delivers value to the user in at least three different ways. The first is that a great amount of manual work involved in planning for a panel layout project has been automated. The automated process gives the user detailed information regarding the number of stock panels required, the nesting plan for each panel, and the layout plan for the actual sections of the physical building. This wealth of information in turn allows the user to more accurately predict the costs associated with material and labor required to undertake the project.

The second benefit to the user is the optimization capability that helps them to minimize the project cost by making the necessary selection from different types of stock panel as well as making sure that a minimum number of panels need to be allocated. The amount of computational task needed to accomplish the optimization is such that manual optimization is unlikely to yield comparable results except in very simple cases.

Finally, the software capability of solving multiple container problems means that optimization does not need to be performed on the basis of individual sections of the building. As previously mentioned, the use of a smaller stock panel typically results in better material efficiency. It follows that the material usage efficiency tends to improve when the ratio of container area to the stock panel area increases. Furthermore using

multiple containers for a single MCPO problem is a good way of improving the ratio. The important implication is that not only does solving MCPO for multiple sections of the building in a single optimization run become possible, but doing so actually generates less waste for the overall project than it would if the sections are optimized individually.

8. Conclusion

The objectives of this research were stated in Chapter 1 as;

- To develop a model of the MCPO problem into a series of parameters that can be optimized by numerical algorithms
- To develop a software application which implements MCPO automated optimization processes using general-purpose programming tools
- To demonstrate that optimization algorithms can be utilized to solve the MCPO problem effectively
- To observe the relative performance of alternative optimization algorithms

It is clear that each of these objectives have been addressed by the research undertaken, although at times with unexpected observations.

Decomposing MCPO into a two-stage optimization model provides a solid ground for constructing a well-functioning solution. The study has also proven that with the support of appropriate analysis, software application to solve complex problems such as MCPO can be successfully implemented using standard modeling and programming tools.

Various technical issues discovered throughout the development of the software have provided insight to the nature of the problem as well as the challenges of constructing a solution using contemporary programming tools. Pragmatic approaches have been taken to resolve some of those problems for the lack of access to better alternatives. The deficiencies of pragmatic solutions have manifested themselves in some sub-optimum performance in affected modules.

Nonetheless the resulting software is capable of solving actual MCPO problems, thus offering a proof for the correctness of the two-stage optimization model. Successful implementation of the software also proves the feasibility of constructing MCPO solutions automatically for commercial use with current computing technologies.

A series of experiments have demonstrated the outstanding performance of greedy search in comparison with simulation-based search algorithms represented by the Monte Carlo technique and a Genetic Algorithm. While this result is not surprising for the Monte Carlo technique given its inherent inefficiency, the unexpected lack of

performance of such a sophisticated method as a Genetic Algorithm calls for further investigation in the area.

There are a number of possible reasons for the Genetic Algorithm's relative poor performance. The first is the efficiency of the coded implementation, which has been based of an unsophisticated version of the algorithm featuring naïve strategies in accomplishing its key sub-tasks. The second possible reason is the accuracy of the parameter modeling that prevailed, in which many important parameters of the nesting problem have been omitted to be optimized by external processes. Finally, the MCPO class of problems may have certain characteristics that make Genetic Algorithms unsuitable to solve them. None of these assertions have been proved however, implying the need for further research in the area.

8.1. Suggestions for Future Work

At present the MCPO application has limitations indicating its lack of maturity as solution, both as research and commercial software. There are quite a few areas where it can be improved to expand its capabilities. There are also opportunities of exploring various theoretical aspects of MCPO problem.

A degree of uncertainty regarding the correctness of the program code exists because of the practice of manual coding and the absence of rigorous mathematical analysis in constructing various algorithms. These are the byproduct of the Rapid Application Development (RAD) model used for the project. While the use of RAD was justified by the need to construct the solution software while simultaneously identifying various issues unknown at the time the project was conceived, further development can use different development models to take advantage of knowledge of key issues discovered during the course of this project.

Many programming errors can be eliminated when automated coding is used. In this case, *4th Generation Technique* (4GT) is a good candidate as alternative development model. Similarly, *Formal Methods* can increase the efficiency of many underlying modules by its rigorous mathematical analysis. Future studies may discover the benefit of employing these two development models in improving the quality of MCPO software.

At a more technical level, there are a number of ad-hoc algorithms employed in the current MCPO software that can be replaced with more efficient substitutes.

Mathematically proven algorithms are especially valuable in increasing the efficiency of difficult geometric operations such as polygon triangulation, polygon overlap detection, and polygon clipping. Only pragmatic solutions have been implemented to solve these three problems, resulting in possibly sub-optimal performance. The use of more advanced algorithms, even in the form of third-party software components, may have direct impact on the performance of MCPO software.

There also a number of unresolved problems with the second-stage optimization. The most glaring problem is the unsatisfactory implementation of the best-fit strategy, which calls for a better model in calculating the convex vacant space. Better models resulting from study in this area will enable the correct realization of a best-fit placement strategy. Similarly, the ability to automatically identify the best point of origin is likely to lead to significantly improved results.

Another problematic aspect of the second-stage optimization is the parameter modeling into chromosome used by simulation algorithms such as the Monte Carlo technique and Genetic Algorithm. The current solution seems inadequate from a performance standpoint, with many important parameters missing from the bit string representation of the chromosome. It is used regardless because the task of constructing a better model is not trivial. Given the existence of inter-dependencies between the second-stage parameters, simple mapping of the parameters to the bit string is unlikely to be workable. A more thorough exploration of possible parameter modeling is necessary to find a better alternative.

On a smaller scale, further investigation can be made to improve the effectiveness of the Genetic Algorithm using the current chromosome structure. For all its flaws, the current parameter modeling serves a critical role of making it possible to employ such an advanced optimization technique to solve MCPO problems. A major problem encountered with utilizing Genetic Algorithms in this particular problem is the abundance of invalid chromosomes, in which stock panels are assigned with more pieces than they can accommodate. An invalid chromosome complicates the task of calculating the fitness function of the individual.

At present, invalid chromosomes are turned into their valid equivalents through a correction procedure of redistributing the surplus pieces found in the overcrowded panels. Although this approach effectively enables a fitness function to be calculated directly, its potentially adverse impact on the survival of better individuals has not been

explored. An investigation into reintroducing modified strings back into the population could lead to a significant improved in performance.

An alternative solution is to penalize the chromosome for the invalid clusters it contains. For example, the surface area of the pieces that fall outside the stock panel boundaries may be used to calculate the penalty applied to the chromosome. More solutions are likely to result from a closer study in this area.

The use of a more sophisticated implementation of the Genetic Algorithm should also be explored. Given the lack of sophistication of the current implementation, it is possible that improvement in the evolution mechanism of the algorithm will have direct impact on the result of the second-stage solution. Key aspects of the algorithms that need improvement are the selection policy, crossover mechanism, and the introduction of elitism where best individuals are carried over to the succeeding generations.

Successful use of the Monte Carlo technique and the Genetic Algorithm has proven that optimization algorithms can be used to solve the second-stage problem. The implication is that other optimization techniques could also be used in their place. Various optimization techniques such as *Swarm Intelligence*, *Simulated Annealing*, and *Tabu Search* can potentially increase the effectiveness of the search for the second-stage optimization.

From a commercial point of view, study of the impact of the software to the practices of house construction has significant value. A particular point of interest is the deployment value of the greedy algorithm, which perfectly matches the needs of an industry solution for quick optimization and high quality results. The software has been intended to streamline the process of covering sections of a building. It has been envisioned that successful deployment of the software may also lead to its usage for assisting related tasks. For instance, the availability of MCPO solutions for covering a house section with drywall can be used to design the wooden frame of that section. The resulting design can then be used by the frame manufacturer to assist the calculation of the amount of required timber as well as it cutting plan. Study in such areas will reveal which aspects of the software that gives most value to the user, which in turn will allow improvement efforts for the software to be directed towards areas that benefit the user most.

9. Cited References

Adamowicz, M., & Albano, A. (1976). Nesting Two-Dimensional Shapes In Rectangular Modules. IEEE Transactions on Systems, Man and Cybernetics, 8(1), 27-33.

Ahn, C.W. & Ramakrishna, R.S. (2003). Elitism-based compact genetic algorithms. IEEE Transactions on Evolutionary Computation, 7(4), 367-385.

Ansari, N., & Hou, E. (1997). Computational Intelligence for Optimization. Newark, New Jersey: Kluwer Academic Publishers.

Bäck, T., & Schwefel, H.-P. (1996). Evolutionary computation: an overview. Paper presented at the Evolutionary Computation, 1996., Proceedings of IEEE International Conference on.

Bellomo, D., Naso, D. & Turchiano, B. (2002) Improving Genetic Algorithms: An approach based on multi-elitism and Lamarckian mutation. Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 4, 89-94.

Bennell, J. A., & Dowsland, K. A. (1999). Tabu thresholding implementation for the irregular stock cutting problem. International Journal of Production Research, 37(18), 4259-4275.

BISCO. (2006). The BISCo Solution. Retrieved 8 June 2006, from www.bisco.com

Bolc, L., & Cytowski, J. (1992). Search Methods for Artificial Intelligence. Warsaw, Poland: Academic Press.

Bounsaythip, C., & Maouche, S. (1997). Irregular shape nesting and placing with evolutionary approach. Paper presented at the Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on, Orlando, FL, United States.

Bounsaythip, C., Maouche, S., & Neus, M. (1995). Evolutionary search techniques application in automated layout-planning optimization problem. Paper presented at the Systems, Man and Cybernetics, 1995. 'Intelligent Systems for the 21st Century'., IEEE International Conference on.

Caprara, A., & Monaci, M. (2004). On the two-dimensional Knapsack Problem. Operations Research Letters, 32(1), 5-14.

Chan, F. T. S., Au, K. C., & Chan, P. L. Y. (2005). A genetic algorithm approach to bin packing in an ion plating cell. Proceedings of the Institution of Mechanical Engineers, Part B (Journal of Engineering Manufacture), 219(B1), 1-13.

Chang, W.-D. (2006). Coefficient estimation of IIR filter by a multiple crossover genetic algorithm. Computers and Mathematics with Applications, 51(9-10), 1437-1444.

Chazelle, B. (1990, 1990//). Triangulating a simple polygon in linear time. Paper presented at the Proceedings. 31st Annual Symposium on Foundations of Computer Science (Cat. No.90CH2925-6), 22-24 Oct. 1990, St. Louis, MO, USA.

Cheng, C. Y., & Atkinson, J. (1994). Comparison of some methods for computer-aided nesting of sheet components. Journal of Materials Processing Technology, 44(3-4), 311-318.

Connor, A. (1996). The Synthesis of Hybrid Mechanism Using Genetic Algorithms. PhD Thesis, Liverpool John Moores University, Liverpool, UK.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2003). Introduction to Algorithms (Second ed.). Cambridge, Massachusetts: The MIT Press.

Crispin, A., Clay, P., Taylor, G., Bayes, T., & Reedman, D. (2005). Genetic algorithm coding methods for leather nesting. Applied Intelligence, 23(1), 9-20.

Daniels, K., & Milenkovic, V. (1995). Multiple translational containment: approximate and exact algorithms. Paper presented at the Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms, San Fracisco, California, United States.

Dean, T., Allen, J., & Aloimonos, Y. (1995). Artificial Intelligence: Theory and Practice. Menlo Park, California: Addison-Wesley Publishing.

De Jong, K. (1975). An Analysis of the Behavior of a Class of Genetic Adaptive Systems. PhD Thesis, The University of Michigan, Michigan, USA.

Djurisic, A.B. (1998). Elite genetic algorithms with adaptive mutations for solving continuous optimization problems - application to modeling of the optical constants of solids. Optics Communications, 151(1-3), 147-159.

Dolan, A. (2006). GA Playground (A Genetic Algorithm Toolkit). Downloaded from http://www.aridolan.com/ga/gaa/gaa.html on 14th August 2006.

Du, J., Alhajj, R. & Barker, K. (2006). Genetic algorithms based approach to database vertical partition. Journal of Intelligent Information Systems, 26(2), 167-183.

Dyckhoff, H. (1990). Typology of cutting and packing problems. European Journal of Operational Research, 44(2), 145-159.

Faina, L. (1999). Application of simulated annealing to the cutting stock problem. European Journal of Operational Research, 114(3), 542-556.

Falkenauer, E., & Delchambre, A. (1992). A genetic algorithm for bin packing and line balancing, Paper presented at the 1992 International Conference on Robotics and Automation, Nice, France

Fortnow, L., Homer, S. (2002). A Short History of Computational Complexity. In D. van Dalen, J. Dawson, and A. Kanamori, editors, The History of Mathematical Logic. North-Holland, Amsterdam.

Fournier, A., & Montuno, D. Y. (1984). Triangulation of Simple Polygons and Equivalent Problems. ACM Transactions on Graphics, 3(2), 153-174.

Garey, M., Johnson, D., Preparata, F., & Tarjan, R. (1978). Triangulating a simple polygon. Information Processing Letters, 7(4), 175-179.

Georgis, N., Petrou, M., & Kittler, J. (2000). On the generalised stock-cutting problem. Machine Vision and Applications, 11(5), 231-241.

Goldberg, D. E. (1989). Genetic Algorithm in Search, Optimization, and Machine Learning. Reading, Massachusetts: Addison-Wesley Publishing.

Gordon, V., Mathias, K., & Whitley, D. (1994). Cellular Genetic Algorithms as Function Optimizers: Locality Effects. Paper presented at the Proceedings of the 1994 ACM symposium on Applied computing, Phoenix, Arizona, United States.

Hakimi, S. L. (1988). Problem on rectangular floorplans. Paper presented at the 1988 IEEE International Symposium on Circuits and Systems, Proceedings, Jun 7-9 1988, Espoo, Finl.

Hedar, A. (2006). Global Optimization Test Problems. Retrieved 18 July 2006, 2006, from http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO.htm

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. MIS Quarterly, 28(1), 75-105.

Hifi, M. (1998). Exact algorithms for the guillotine strip cutting/packing problem. Computers & Operations Research, 25(11), 925-940.

Holland, J. Adaptation in Natural and Artificial Systems. University of Michigan Press, 1975.

Horn, J. (2005). Shape nesting by coevolving species. Paper presented at the Genetic And Evolutionary Computation Conference, Washington DC, USA.

Hsieh, S.-T., Lin, C.-W., & Sun, T.-Y. (2005). Particle swarm optimization for macrocell overlap removal and placement, Proceedings of the 2005 IEEE International Symposium on Swarm Intelligence, Pasadena, CA, United States.

Hsu, Y. C., & Kubitz, W. J. (1988). ALSO: A System for Chip Floorplan Design. Integration, the VLSI Journal, 6(2), 127-146.

Hwang, S-F. & He, R-S. (2006) Improving real-parameter genetic algorithm with simulated annealing for engineering problems. Advances in Engineering Software, 37(6), 406-418.

Imahori, S., Yagiura, M., & Ibaraki, T. (2005). Improved local search algorithms for the rectangle packing problem with general spatial costs. European Journal of Operational Research, 167(1), 48-67.

Jiang, J. Q., Xing, X. L., Yang, X. W., & Liang, Y. C. (2004). A hybrid algorithm based on PSO and genetic operation and its applications for cutting stock problem, Paper presented at the Proceedings of 2004 IEEE International Conference on Machine Learning and Cybernetics, Shanghai, China.

Kiyota, K., & Fujiyoshi, K. (2000). Simulated annealing search through general structure floorplans using sequence-pair. Paper presented at the Proceedings of the IEEE 2000 Internaitonal Symposium on Circuits and Systems, May 29-May 31 2000, Geneva, Switzerland.

Lamousin, H., & Waggenspack Jr., W. N. (1997). Nesting of two-dimensional irregular parts using a shape reasoning heuristic. CAD Computer Aided Design, 29(3), 221-238.

Laurent, D. G., & Iyengar, S. S. (1982). Heuristic Algorithm for Optimal Placement of Rectangular Objects. Information Sciences, 26(2), 127-139.

Levenick, J.R. (1991). Inserting introns improves genetic algorithm success rate: Taking a cue from biology. Proceedings of the International Conference on Genetic Algorithms, 123.

Levitin, G. (2006). Genetic algorithms in reliability engineering. Reliability Engineering and System Safety, 91(9), 975-976.

Lewis, J. E., Ragade, R. K., Kumar, A., & Biles, W. E. (2005). A distributed chromosome genetic algorithm for bin-packing. Robotics and Computer-Integrated Manufacturing, 21(4-5), 486-495.

Liang, Y.-D., & Barsky, B. A. (1983). An analysis and algorithm for polygon clipping. Communications of the ACM, 26(11), 868-877.

Lis, J. (1996). Parallel genetic algorithm with the dynamic control parameter. Proceedings of the IEEE Conference on Evolutionary Computation, 324-329.

Liu, D., & Teng, H. (1999). Improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles. European Journal of Operational Research, 112(2), 413-420.

Lodi, A., Martello, S., & Monaci, M. (2002). Two-dimensional packing problems: A survey. European Journal of Operational Research, 141(2), 241-252.

Maillot, P. G. (1992). A New, Fast Method For 2D Polygon Clipping: Analysis and Software Implementation. ACM Transactions on Graphics, 11(3), 276-290.

Mak, B., Blanning, R. & Ho, S. (2006). Genetic algorithms in logic tree decision modelling. European Journal of Operational Research, 170(2), 597-612.

Mount, D. (1992). Intersection detection and separators for simple polygons. Paper presented at the Proceedings of the eighth annual symposium on Computational geometry, Berlin, Germany.

Murata, H., Fujiyoshi, K., Nakatake, S., & Kajitani, Y. (1995). Rectangle-packing-based module placement. Paper presented at the Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, Nov 5-9 1995, San Jose, CA, USA.

Nagao, T. (1996, 20-22 May 1996). Homogeneous Coding for Genetic Algorithm Based Parameter Optimization. Paper presented at the Proceedings of the IEEE Conference on Evolutionary Computation, Nagoya, Japan.

Ndiritu, J.G. & Daniell, T.M. (1999). Improved genetic algorithm for continuous and mixed discrete-continuous optimization. Engineering Optimization, 31(5), 589-614.

Nunamaker, J., Chen, M. (1991). Systems Development in Information Systems Research. Paper presented at the Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Kailua-Kona, Hawaii, USA.

Nye, T. J. (2001). Optimal nesting of irregular convex blanks in strips via an exact algorithm. International Journal of Machine Tools and Manufacture, 41(7), 991-1002.

Oloffson, P. (2005). Probability, Statistics, and Stochastic Processes. Houston, Texas: Wiley-Interscience.

Pai, P.-F., Deng, S., Lai, C.-C. and Wu, P.-S. (2006). Genetic algorithms in simulating optimal stacking sequence of a composite laminate plate with constant thickness. International Journal of Modelling and Simulation, 26(1), 61-67.

Peffers, K., Tuunanen, T., Gengler, C., Rossi, M., Hui, W., Virtanen, V., Bragge, J. (2006). The Design Science Research Process: A Model for Producing and Presenting Information Systems Research. Proceedings of the First International Conference on Design Science Research in Information Systems and Technology (DESRIST 2006), Claremont, CA. Retrieved 17/05/2006 from http://ncl.cgu.edu/designconference/DESRIST%202006%20Proceedings/4A_2.pdf.

Prasad, Y. K. D. V. (1994). Set of heuristic algorithms for optimal nesting of two-dimensional irregularly shaped sheet-metal blanks. Computers in Industry, 24(1), 55-70.

Prasad, Y. K. D. V., & Somasundaram, S. (1991). CASNS- a heuristic algorithm for the nesting of irregular-shaped sheet-metal blanks. Computer-Aided Engineering Journal, 8(2), 69-73.

Pressman, R. (2004). Software Engineering: A Practitioner's Approach (Sixth ed.). New York: McGraw-Hill Higher Education.

Reca, J. & Martinex, J. (2006). Genetic algorithms for the design of looped irrigation water distribution networks. Water Resources Research, 42(5), W05416.

Schneider, B.-O., & Van Welzen, J. (1998). Efficient polygon clipping for an SIMD graphics pipeline. IEEE Transactions on Visualization and Computer Graphics, 4(3), 272-285.

Shi, X.H., Wan, L.M., Lee, H.P., Yang, X.W., Wang, L.M. & Liang, Y.C. (2003). An improved genetic algorithm with variable population-size and a PSO-GA based hybrid evolutionary algorithm. International Conference on Machine Learning and Cybernetics, 3, 1735-1740.

Shian-Miin, H., Cheng-Yan, K., & Jorng-Tzong, H. (1994). On solving rectangle bin packing problems using genetic algorithms, Paper presented at the 1994 IEEE International Conference on Systems, Man, and Cybernetics, San Antonio, TX, USA.

Shigehiro, Y., Koshiyama, S., & Masuda, T. (2001). Stochastic tabu search for rectangle packing. Paper presented at the 2001 IEEE International Conference on Systems, Man and Cybernetics, Oct 7-10 2001, Tucson, AZ.

Sibley-Punnett, L., & Bossomaier, T. (2001). Optimisation techniques for roof layout. Paper presented at the Electrical and Electronic Technology, 2001. TENCON. Proceedings of IEEE Region 10 International Conference on.

Sun, Z.-G., & Teng, H.-F. (2002). An ant colony optimization based layout optimization algorithm, Paper presented at 2002 IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering, Beijing, China.

Tanenbaum, A. (2001). Modern Operating Systems (Second ed.). New Jersey: Prentice-Hall.

Tokuyama, H., & Ueno, N. (1985). Cutting Stock Problem For Large Sections In The Iron And Steel Industries. European Journal of Operational Research, 22(3), 280-292.

Vassiliadis, V. S. (2005). Two-dimensional stock cutting and rectangle packing: Binary tree model representation for local search optimization methods. Journal of Food Engineering: Operational Research and Food Logistics, 70(3), 257-268.

Vatti, B. R. (1992). A Generic Solution to Polygon Clipping. Communications of The ACM, 35(7), 56-63.

Watkins, A., Hufnagel, E.M., Berndt, D. & Johnson, L. (2006). Using genetic algorithms and decision tree induction to classify software failures. International Journal of Software Engineering and Knowledge Engineering, 16(2), 269-291.

Weisstein, E. W. (1999). Monte Carlo Method. Retrieved June, 2006, from http://mathworld.wolfram.com/MonteCarloMethod.html

Yoon, H.-S., & Moon, B.-R. (2002). An empirical study on the synergy of multiple crossover operators. IEEE Transactions on Evolutionary Computation, 6(2), 212-223.

Yourdon, E. (1989). Modern structured analysis. Englewood Cliffs, N.J.: Prentice-Hall.

Yourdon, E. (2006). Just Enough Structured Analysis. Retrieved 5 July 2006, from http://www.yourdon.com/strucanalysis/

Yuping, Z., Shouwei, J., & Chunli, Z. (2005). A very fast simulated re-annealing algorithm for the leather nesting problem. International Journal of Advanced Manufacturing Technology, 25(11-12), 1113-1118.

Zhang, M., & Sabharwal, C. L. (2002). An efficient implementation of parametric line and polygon clipping algorithm. Paper presented at the Applied Computing 2002: Proceeedings of the 2002 ACM Symposium on Applied Computing, Mar 11-14 2002, Madrid, Spain.

Appendix A: Experiment Results

Rotation	None	180 Degree	90 Degree	
Placement	First Fit	First Fit	First Fit	Best Fit
Experiment #	1	1	1	1
Regular Panels	10	10	10	10
Partial Panels	14	14	14	14
Total Panels	24	24	24	24
Stock Panels	17	17	17	17
Shared Edge	2450.00	2450.00	2450.00	2450.00
Covered Area	85000.00	85000.00	85000.00	85000.00
Stock Panel Area	85000.00	85000.00	85000.00	85000.00
Wasted Material	0.00	0.00	0.00	0.00
Solution Efficiency	100.00%	100.00%	100.00%	100.00%
Search Duration	0:00:01	0:00:01	0:00:01	0:00:01

Table A.1: Greedy Search Optimization on Simple Rectangular Container

Rotation	None	180 Degree	90 Degree	
Placement	First Fit	First Fit	First Fit	Best Fit
Experiment #	1	1	1	1
Regular Panels	11	11	11	11
Partial Panels	24	19	24	24
Total Panels	35	30	35	35
Stock Panels	25	22	25	25
Shared Edge	3013.00	2523.33	3013.00	3017.00
Covered Area	94300.00	94300.00	94300.00	94300.00
Stock Panel Area	108000.00	105600.00	108000.00	108000.00
Wasted Material	13700.00	11300.00	13700.00	13700.00
Solution Efficiency	87.31%	89.30%	87.31%	87.31%
Search Duration	0:00:01	0:00:01	0:00:01	0:00:01

Table A.2: Greedy Search Optimization on Single Wall

Rotation	None	180 Degree	
Placement	First Fit	First Fit	Best Fit
Experiment #	1	1	1
Regular Panels	8	8	8
Partial Panels	24	24	24
Total Panels	32	32	32
Stock Panels	18	17	18
Shared Edge	1698.24	1722.34	1781.21
Covered Area	26067.20	26067.20	26067.20
Stock Panel Area	28800.00	27200.00	28800.00
Wasted Material	2732.80	1132.80	2732.80
Solution Efficiency	90.51%	95.84%	90.51%
Search Duration	0:00:01	0:00:01	0:00:01

Table A.3: Greedy Search Optimization on Simple Roof

Rotation	None	180 Degree	
Placement	First Fit	First Fit	Best Fit
Experiment #	1	1	1
Regular Panels	25	25	25
Partial Panels	129	129	129
Total Panels	154	154	154
Stock Panels	65	65	64
Shared Edge	9629.28	9586.05	9929.57
Covered Area	124153.13	124153.13	124153.13
Stock Panel Area	130000.00	130000.00	128000.00
Wasted Material	5846.87	5846.87	3846.87
Solution Efficiency	95.50%	95.50%	96.99%
Search Duration	0:00:02	0:00:04	0:00:03

Table A.4: Greedy Search Optimization on Complex Roof

Rotation	None		180 Degree		90 Degree			
Placement	First Fit		First Fit		First Fit		Best Fit	
Iterations	10000		10000		10000		10000	
Experiment #	1	2	1	2	1	2	1	2
Regular Panels	10	10	10	10	10	10	10	10
Partial Panels	14	14	14	14	14	14	14	14
Total Panels	24	24	24	24	24	24	24	24
Stock Panels	17	17	17	17	17	17	17	17
Shared Edge	2450.00	2450.00	2450.00	2450.00	2450.00	2450.00	2450.00	2450.00
Covered Area	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00
Stock Panel Area	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00
Wasted Material	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Solution Efficiency	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Search Duration	00:00:32.55	0:00:32	0:00:40	0:00:32	0:00:31	0:00:31	0:00:40	0:00:41

Table A.5: Monte Carlo Optimization on Simple Rectangular Container

Rotation	None		180 Degree		90 Degree			
Placement	First Fit		First Fit		First Fit		Best Fit	
Iterations	10000		10000		10000		10000	
Experiment #	1	2	1	2	1	2	1	2
Regular Panels	11	11	11	11	11	11	11	11
Partial Panels	19	19	19	19	19	19	19	19
Total Panels	30	30	30	30	30	30	30	30
Stock Panels	23	24	23	23	23	23	23	23
Shared Edge	2471.67	2416.67	2331.67	2338.33	2491.67	2283.33	2416.67	2426.67
Covered Area	94300.00	94300.00	94300.00	94300.00	94300.00	94300.00	94300.00	94300.00
Stock Panel Area	110400.00	115200.00	110400.00	110400.00	110400.00	110400.00	110400.00	110400.00
Wasted Material	16100.00	20900.00	16100.00	16100.00	16100.00	16100.00	16100.00	16100.00
Solution Efficiency	85.42%	81.86%	85.42%	85.42%	85.42%	85.42%	85.42%	85.42%
Search Duration	0:03:31	0:03:29	0:05:41	0:05:45	0:05:41	0:03:29	0:04:40	0:04:38

Table A.6: Monte Carlo Optimization on Single Wall

Rotation	None		180 Degree			
Placement	First Fit		First Fit		Best Fit	
Iterations	10000		10000		10000	
Experiment #	1	2	1	2	1	2
Regular Panels	8	8	8	8	8	8
Partial Panels	24	24	24	24	24	24
Total Panels	32	32	32	32	32	32
Stock Panels	21	22	22	20	21	21
Shared Edge	1481.72	1523.09	1553.51	1627.01	1761.80	1735.84
Covered Area	26067.20	26067.20	26067.20	26067.20	26067.20	26067.20
Stock Panel Area	33600.00	35200.00	35200.00	32000.00	33600.00	33600.00
Wasted Material	7532.80	9132.80	9132.80	5932.80	7532.80	7532.80
Solution Efficiency	77.58%	74.05%	74.05%	81.46%	77.58%	77.58%
Search Duration	0:02:12	0:02:27	0:03:08	0:03:05	0:06:10	0:06:10

Table A.7: Monte Carlo Optimization on Simple Roof

Rotation	None		180 Degree			
Placement	First Fit		First Fit		Best Fit	
Iterations	10000		10000		10000	
Experiment #	1	2	1	2	1	2
Regular Panels	25	25	25	25	25	25
Partial Panels	129	129	129	129	129	129
Total Panels	154	154	154	154	154	154
Stock Panels	94	99	98	96	86	93
Shared Edge	9169.97	9312.40	9297.72	9197.89	9683.72	9661.90
Covered Area	124153.13	124153.13	124153.13	124153.13	124153.13	124153.13
Stock Panel Area	188000.00	198000.00	196000.00	192000.00	172000.00	186000.00
Wasted Material	63846.87	73846.87	71846.87	67846.87	47846.87	61846.87
Solution Efficiency	66.04%	62.70%	63.34%	64.66%	72.18%	66.75%
Search Duration	0:18:29	0:18:23	0:30:04	0:30:15	0:47:14	0:48:05

Table A.8: Monte Carlo Optimization on Complex Roof

Rotation	None		180 Degree		90 Degree			
Placement	First Fit		First Fit		First Fit		Best Fit	
Population | Generation	100 | 100		100 | 100		100 | 100		100 | 100	
Experiment #	1	2	1	2	1	2	1	2
Regular Panels	10	10	10	10	10	10	10	10
Partial Panels	14	14	14	14	14	14	14	14
Total Panels	24	24	24	24	24	24	24	24
Stock Panels	17	17	17	17	17	17	17	17
Shared Edge	2450.00	2450.00	2450.00	2450.00	2450.00	2450.00	2450.00	2450.00
Covered Area	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00
Stock Panel Area	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00	85000.00
Wasted Material	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Solution Efficiency	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Search Duration	0:00:32	0:00:31	0:00:38	0:00:38	0:00:30	0:00:30	0:00:40	0:00:40

Table A.9: Genetic Algorithm Optimization on Simple Rectangular Container

Rotation	None		180 Degree		90 Degree			
Placement	First Fit		First Fit		First Fit		Best Fit	
Population | Generation	100 | 100		100 | 100		100 | 100		100 | 100	
Experiment #	1	2	1	2	1	2	1	2
Regular Panels	11	11	11	11	11	11	11	11
Partial Panels	19	19	19	19	19	19	19	19
Total Panels	30	30	30	30	30	30	30	30
Stock Panels	23	23	23	23	23	23	23	23
Shared Edge	2336.67	2233.33	2341.67	2378.33	2346.67	2411.67	2456.67	2356.67
Covered Area	94300.00	94300.00	94300.00	94300.00	94300.00	94300.00	94300.00	94300.00
Stock Panel Area	110400.00	110400.00	110400.00	110400.00	110400.00	110400.00	110400.00	110400.00
Wasted Material	16100.00	16100.00	16100.00	16100.00	16100.00	16100.00	16100.00	16100.00
Solution Efficiency	85.42%	85.42%	85.42%	85.42%	85.42%	85.42%	85.42%	85.42%
Search Duration	0:03:32	0:03:33	0:05:47	0:05:38	0:03:33	0:03:32	0:04:35	0:04:37

Table A.10: Genetic Algorithm Optimization on Single Wall

Rotation	None		180 Degree			
Placement	First Fit		First Fit		Best	
Population | Generation	100 | 100		100 | 100		100 | 100	
Experiment #	1	2	1	2	1	2
Regular Panels	8	8	8	8	8	8
Partial Panels	24	24	24	24	24	24
Total Panels	32	32	32	32	32	32
Stock Panels	21	22	22	21	21	21
Shared Edge	1567.91	1590.51	1567.77	1574.51	1775.37	1741.84
Covered Area	26067.20	26067.20	26067.20	26067.20	26067.20	26067.20
Stock Panel Area	33600.00	35200.00	33600.00	33600.00	33600.00	33600.00
Wasted Material	7532.80	9132.80	9132.80	7532.80	7532.80	7532.80
Solution Efficiency	77.58%	74.05%	74.05%	77.58%	77.58%	77.58%
Search Duration	0:02:04	0:02:09	0:03:11	0:03:04	0:06:03	0:06:08

Table A.11: Genetic Algorithm Optimization on Simple Roof

Rotation	None		180 Degree			
Placement	First Fit		First Fit		Best	
Population | Generation	100 | 100		100 | 100		100 | 100	
Experiment #	1	2	1	2	1	2
Regular Panels	25	25	25	25	25	25
Partial Panels	129	129	129	129	129	129
Total Panels	154	154	154	154	154	154
Stock Panels	87	88	88	87	88	85
Shared Edge	9360.42	9353.13	9395.97	9366.58	9811.22	9814.43
Covered Area	124153.13	124153.13	124153.13	124153.13	124153.13	124153.13
Stock Panel Area	174000.00	176000.00	176000.00	174000.00	176000.00	170000.00
Wasted Material	49846.87	51846.87	51846.87	49846.87	51846.87	45846.87
Solution Efficiency	71.35%	70.54%	70.54%	71.35%	70.54%	73.03%
Search Duration	0:17:53	0:18:04	0:29:28	0:28:59	0:46:11	0:45:35

A.12: Genetic Algorithm Optimization on Complex Roof

Appendix B: Data Flow Diagrams

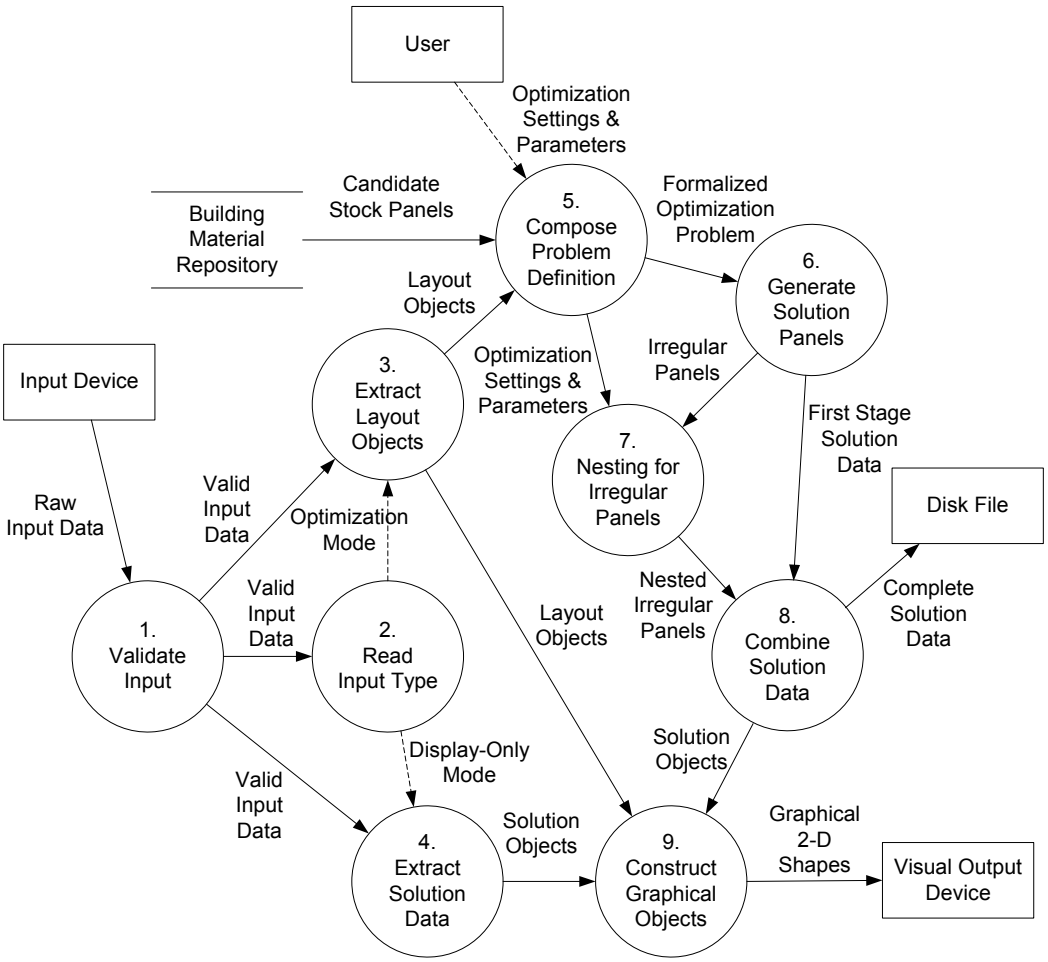


Figure B.1: Overall System Information Flow

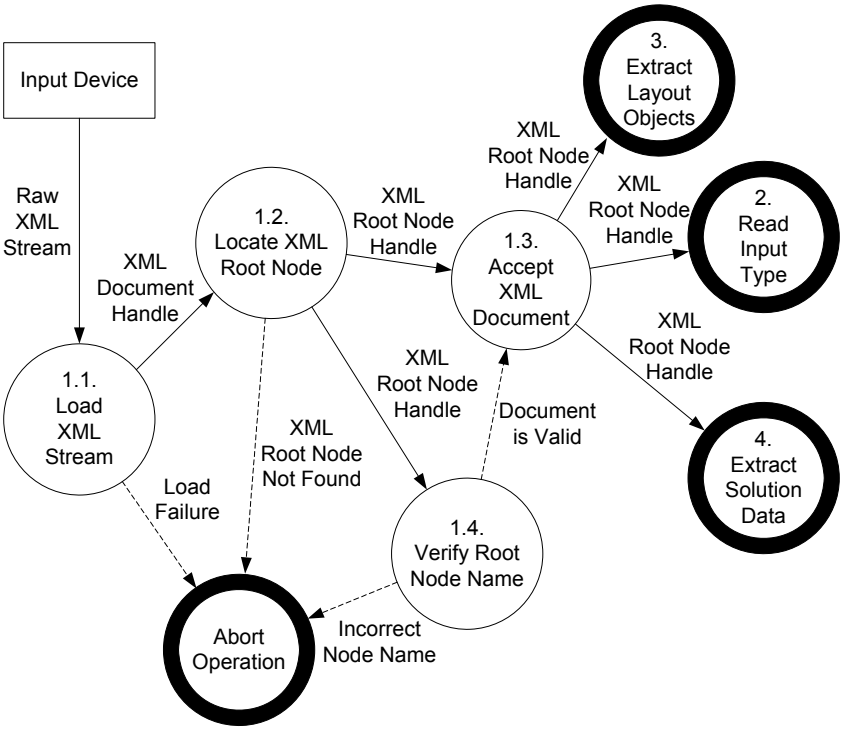


Figure B.2: Validating Input

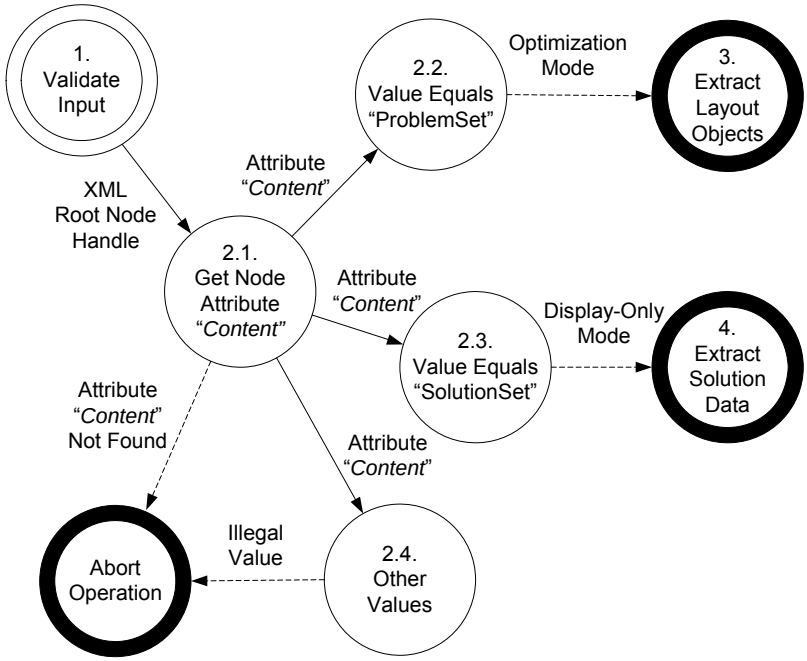


Figure B.3: Reading Input Type

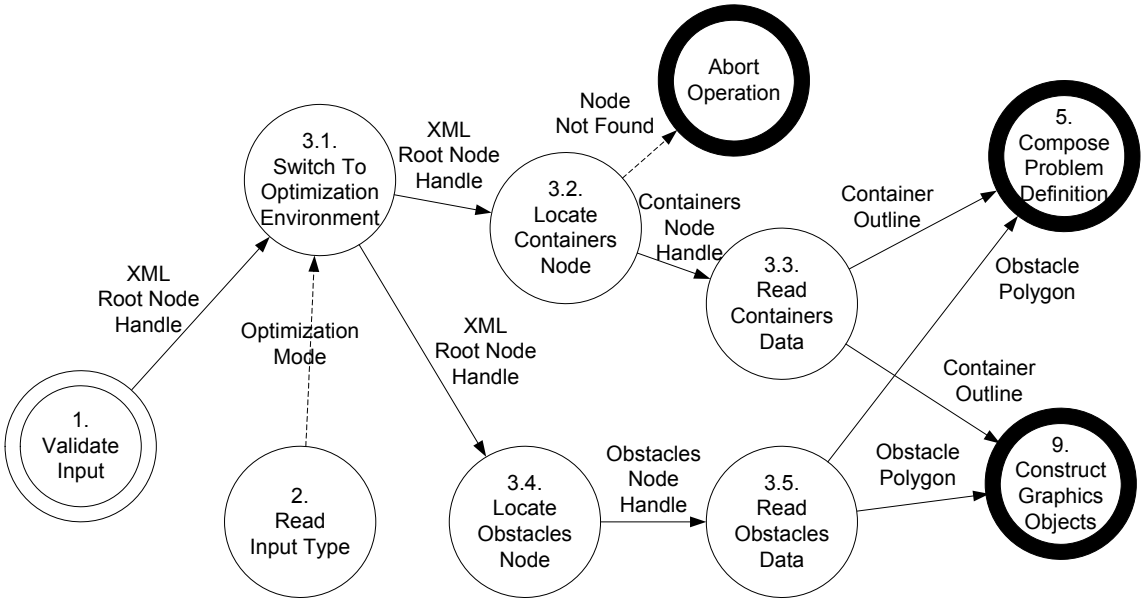


Figure B.4: Extracting Layout Objects

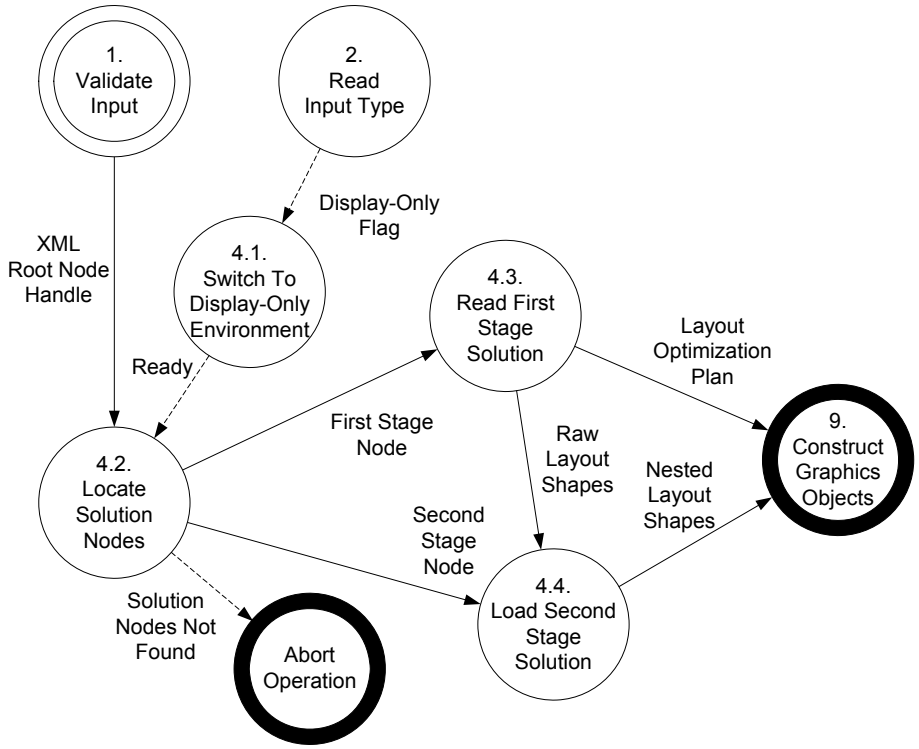


Figure B.5: Extracting Solution Data

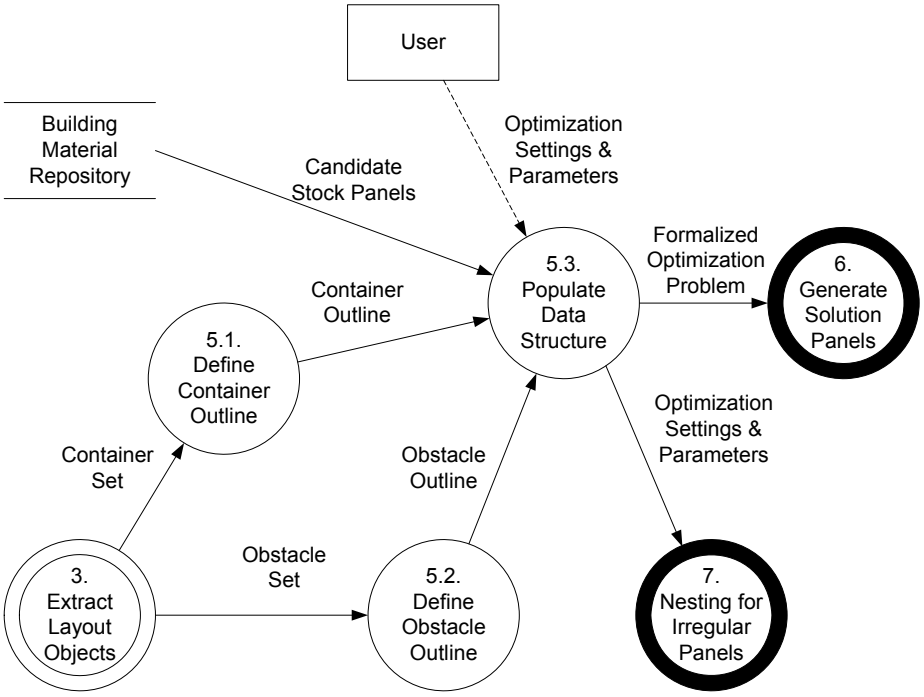


Figure B.6: Composing Problem Definition

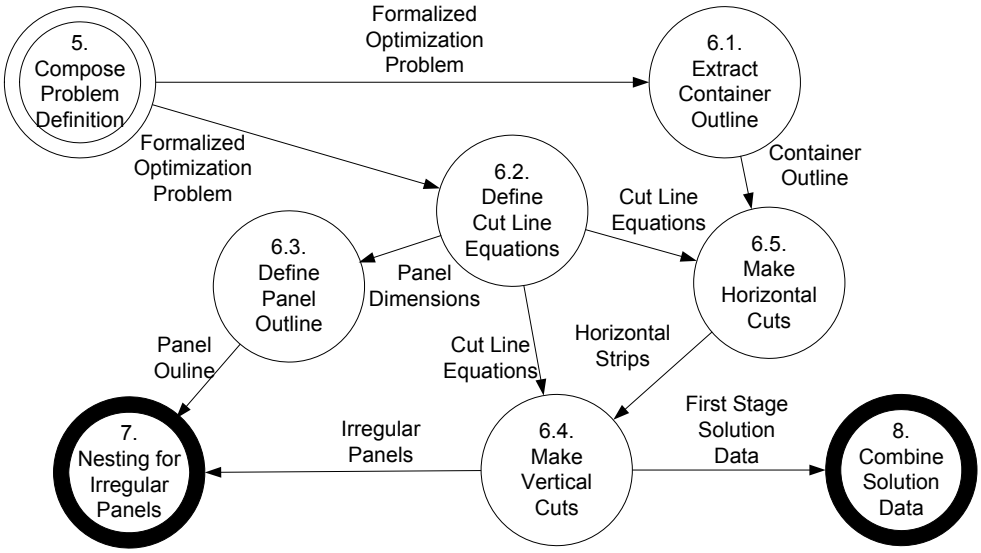


Figure B.7: Generating Solution Panels

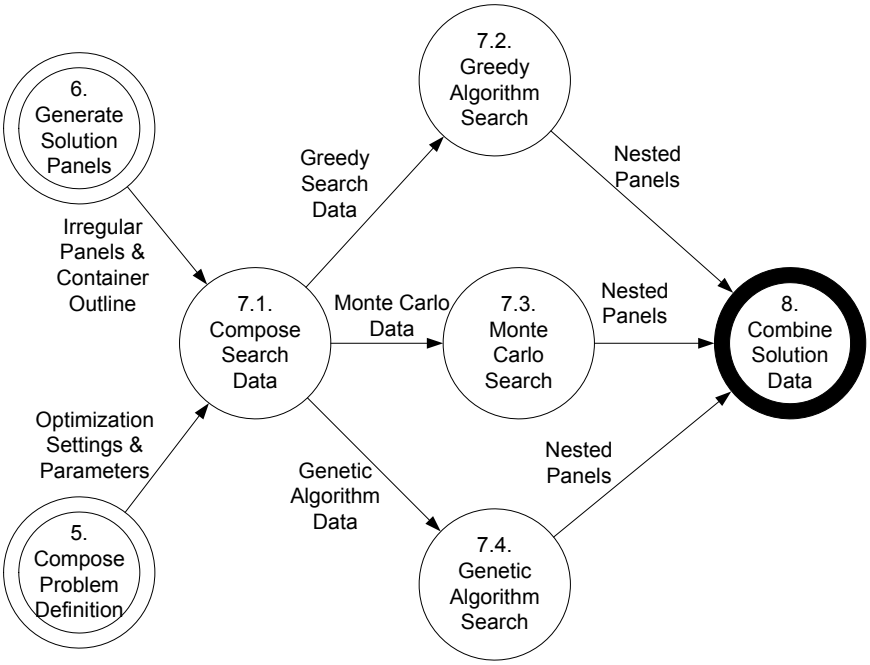


Figure B.8: Nesting for Irregular Panels

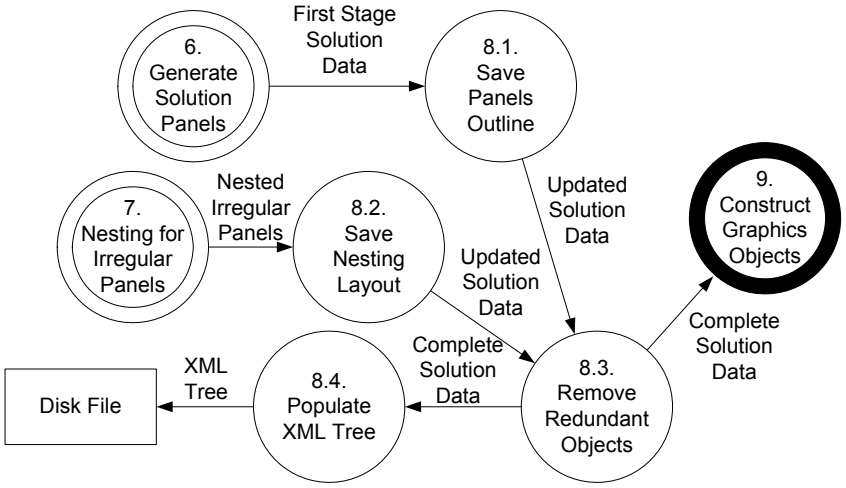


Figure B.9: Combining the Solution Data

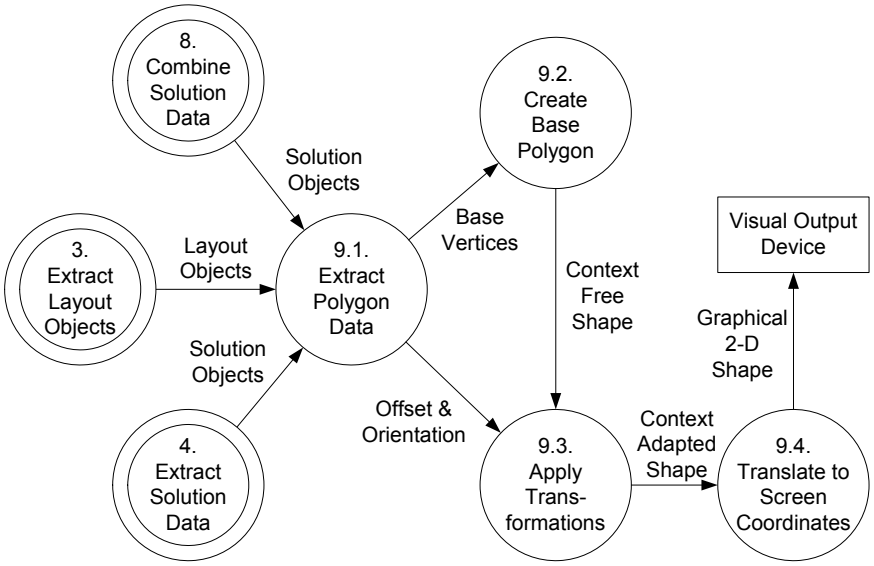


Figure B.10: Constructing Graphical Objects

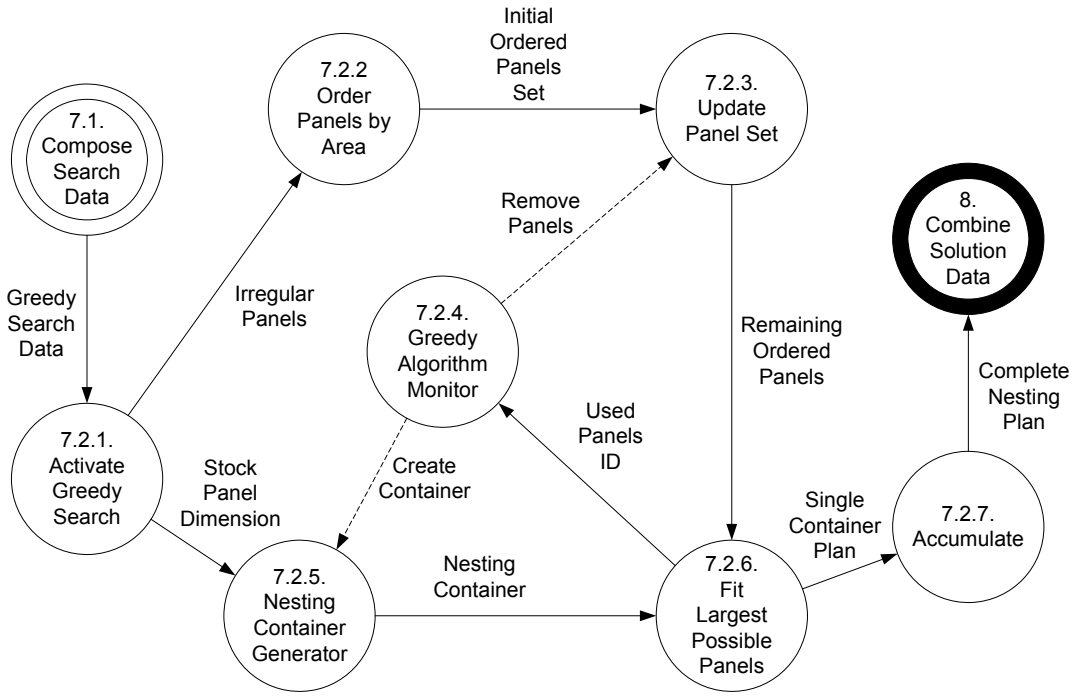


Figure B.11: Greedy Search

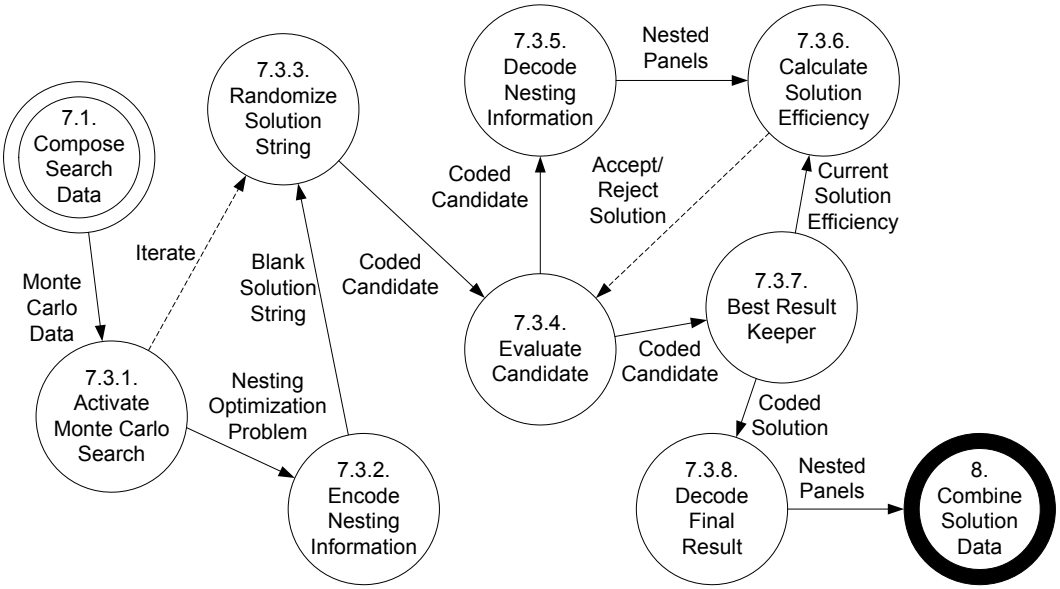


Figure B.12: Monte Carlo Search

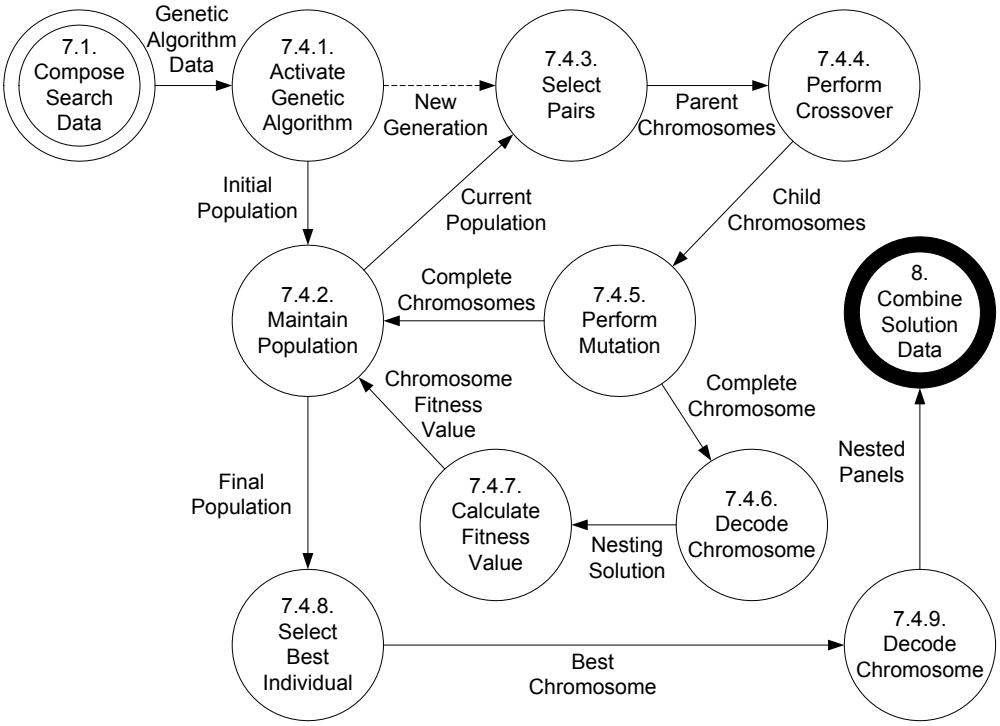


Figure B.12: Genetic Algorithm Search

Appendix C: Data Dictionary

name:	**XY real coordinates**
aliases:	vertex, offset, pivot
where used/ how used:	point definition
	polygon definition
description:	
	XY real coordinates = X value + Y value
	X value = *any real number*
	Y value = *any real number*

Figure C.1: 2-D Vertex Representation

name:	**vertices**
aliases:	raw polygon, primitive polygon
where used/ how used:	polygon definition
description:	
	vertices = {vertex}
	vertex = XY real coordinates

Figure C.2: Basic Polygon Representation

name:	**XY screen coordinates**
aliases:	none
where used/ how used:	screen pixel coordinates (output)
description:	
	XY screen coordinates = X value + Y value
	X value = *any integer number*
	Y value = *any integer number*

Figure C.3: Two-Dimensional Coordinates for Screen Output

name:	**panel ID**
aliases:	none
where used/ how used:	solution definition (input)
	solution definition (output)
description:	
	panel ID = *unique integer ≥ 0*

Figure C.4: Solution Panel Identifier

name:	**orient**
aliases:	rotation
where used/ how used:	generic
description:	
	orient = *real number *r* | -2π ≤ *r* ≤ 2π*

Figure C.5: Object Rotation Constant

name:	**rectangle definition**
aliases:	stock shape, stock panel
where used/ how used:	stock panel definition
description:	
	rectangle = origin + pivot + orient + top + bottom + left + right
	origin = *XY base coordinates*
	pivot = *XY coordinates relative to origin*
	top = XY real coordinates
	bottom = XY real coordinates
	left = XY real coordinates
	right = XY real coordinates

Figure C.6: Rectangle Shape Definition

name:	**polygon definition**
aliases:	single container, single obstacle, shape
where used/ how used:	generic shape definition (input & output)
description:	
	polygon definition = origin + pivot + orient + vertices
	origin = *XY base coordinates*
	pivot = *XY coordinates relative to origin*
	vertices = {vertex}
	vertex = *XY coordinates relative to origin*

Figure C.7: Standard Polygon Definition

name:	**raw input data**
aliases:	complete solution data
where used/ how used:	layout problem definition (input) resolved layout problem data (input)
description:	
	raw input data = signature + document body signature = "PolyWorkSpace" + content type content type = ["ProblemSet" | "SolutionSet"] document body = generic part [+ solution set] generic part = view parameters + container set + obstacle set view parameters = zoom factor + viewing offset zoom factor = *real number > 0* viewing offset = XY screen coordinates

Figure C.8: Program Main Input

name:	**container set**
aliases:	none
where used/ how used:	layout problem definition (input & output) resolved layout problem data (input & output)
description:	
	containers set = {single container} single container = polygon definition

Figure C.9: Container Areas Defined in Layout Problem

name:	**obstacle set**
aliases:	none
where used/ how used:	layout problem definition (input & output) resolved layout problem data (input & output)
description:	
	obstacles set = {single obstacle} single obstacle = polygon definition

Figure C.10: Illegal Areas Defined in Layout Problem

name:	**solution set**
aliases:	none
where used/ how used:	resolved layout problem data (input & output)
description:	
	solution set = {single solution} single solution = stock panel + solution panels + nested layouts stock panel = rectangle definition

Figure C.11: Complete Layout Optimization Solution

name:	**solution panels**
aliases:	none
where used/ how used:	resolved layout problem data (input & output)
description:	
	solution panels = {solution panel} solution panel = panel ID + polygon definition

Figure C.12: Layout Solution Shapes Definition

name:	**nested layouts**
aliases:	nested irregular panels
where used/ how used:	layout solution definition (input & output)
description:	
	nested layouts = nested pack + nested layouts

Figure C.13: Nesting Plans Collection

name:	**nested pack**
aliases:	single layout
where used/ how used:	layout solution definition (input & output)
description:	
	nested pack = {shape layout definition} shape layout definition = panel ID + flip + offset + pivot + orient flip = [TRUE | FALSE] offset = XY real coordinates pivot = XY real coordinates

Figure C.14: Single Nesting Plan

name:	**layout objects**
aliases:	none
where used/ how used:	problem definition (input) extract layout objects (output)
description:	
	layout objects = container set + obstacle set

Figure C.15: Layout Problem's Containers and Obstacles

name:	**candidate stock panels**
aliases:	none
where used/ how used:	problem definition (input) formalized optimization problem (output)
description:	
	candidate stock panels = {rectangle}

Figure C.16: Available Stock Panels

name:	**formalized optimization problem**
aliases:	none
where used/ how used:	generate solution panels (input)
description:	
	formalized optimization problem = container set + obstacle set + candidate stock panels

Figure C.17: Memory Representation of Layout Optimization Problem

name:	**optimization parameters**
aliases:	none
where used/ how used:	nesting for irregular panels (input)
description:	
	optimization parameters = search algorithm + shape orientations + search parameters search algorithm = ["greedy search" | "monte carlo" | "genetic algorithm"] shape orientations = shape orientation [+ shape orientations] shape orientation = ["allow none" | "allow flip" | "allow 180" | "allow 90" | "allow 3"]

Figure C.18: Second Stage Layout Optimization Parameters

name:	**search parameters**
aliases:	none
where used/ how used:	nesting for irregular panels (input)
description:	
	search parameters = [Greedy Search parameters | Monte Carlo parameters | genetic algorithm parameters] Greedy Search parameters = ["first fit" | "best fit"] Monte Carlo parameters = flip probability + iterations flip probability = *real number *r* | 0 ≤ *r* ≤ 1" iterations = *any non-zero integer" Genetic Algorithm parameters = population size + maximum generations + crossover probability + mutation probability

Figure C.19: Optimization Search Parameters

name:	**solution objects**
aliases:	none
where used/ how used:	construct graphics objects
description:	
	solution objects = layout objects + solution panels + nested layouts

Figure C.20: Complete Optimization Solution Package

name:	**graphical 2D shapes**
aliases:	screen shapes
where used/ how used:	screen & printer devices (output)
description:	
	graphical 2D shapes = {graphical 2D shape}

Figure C.21: Collection of 2-D Screen Shapes

name:	**graphical 2D shape**
aliases:	screen shape
where used/ how used:	screen & printer devices (output)
description:	
	graphical 2D shape = name + centre of mass + points + outline color + fill color name = *any literal string* centre of mass = XY screen coordinates points = XY screen coordinates [+ points] outline color = *any available color in the palette* fill color = *any available color in the palette*

Figure C.22: 2-D Screen Shape Definition