The University of Reading

Departments of Computer Science
and Cybernetics

A New Design and Evaluation of a Layout Determining Algorithm for the Stock-Cutting Problem

Mark Barry Jones

Project Supervisor: David Corne

September 1998

A dissertation completing the requirements for the degree of
Master of Science in Information Systems Engineering

To Sarah,

for all her love and support.

1 Abstract

A new layout determining algorithm (LDA) for the stock-cutting problem is developed and a graphically-based, Genetic Algorithms application, StockGA, (developed in Visual C++ v5) is written for investigating the LDA's use in the stock cutting problem.

The particular type of cutting problem used is the 2-D non-guillotine, rectilinear problem. The problem arises in various production process in the glass, wood, paper and textile industries. Thus, the subject is interesting both from theoretical and industrial points of view. The cutting stock problem is a well-known NP-hard combinatorial optimization problem. This results in optimal algorithms being unsuitable for large-scale use as the problem is unsolvable in polynomial-time.

Various optimal and sub-optimal algorithms have been proposed from the operations research field. Optimal methods use linear programming, whilst sub-optimal methods use heuristic-based approaches. Although LDAs are mentioned and applied in the literature, information concerning how these are designed and implemented is sparse. This report presents a viable and proven LDA which is fully documented.

This work uses a heuristic-based approach which generates good solutions using a Genetic Algorithm. The use of different genetic operators and parameters such as which operator rate adaptor to use are investigated and the question of whether or not to include the placement heuristic in the evolution process is considered.

Table of Contents

APPENDIX 2

Appendix 3 – Code

2 The Stock-Cutting Problem

2.1 Introduction to the Stock-Cutting problem

In this section, preliminary background covering the stock-cutting problem is presented. The activites cited in §2.1.1 indicate the research areas in which the problem was first studied. References which are directly related to the project are discussed in §2.2. A taxonomy of layout problems is presented in §2.1.2.

2.1.1 Historical background

Active research has taken place in stock-cutting since the 1940s, but it was not until the 1960s that major work got underway. First attempts concentrated on using exact techniques, such as linear programming to solve the cutting problem optimally, but due to the NP-completeness of the cutting problem large-scale implementations were unfeasible due to the size of the search-space. The stock cutting problem is NP-hard – no known polynomial algorithm exists to solve the problem optimally.

It is widely accepted that heuristic techniques show the most promise for progress, and in recent years the use of evolutionary algorithms has yielded significant gains. The historic background is well documented in Kendall's literature review [1]. The work which is relevant to this project may be found in my literature review, whilst [1] provides background in this section.

In 1940 a paper [2] was published discussing how a square may be broken into rectangular pieces. 1951 saw the link being made between linear programming theory and the stock-cutting problem, in [3]. Linear and dynamic programming models [21], [22], were developed throughout the 1960s that solved the one- and two-dimensional problems to optimality. In the 1970s it was suggested [4], [5], that the order of the cuts and defect avoidance should be considered along with trim loss that had been studied before. Problems were restricted to rectangles and guillotine-cutting due to the complexity of the problem. [24] presents a tree search approach that found the optimal solution, whilst previous research had used mathematical models. Heuristic solutions began to be used and in 1976, [18], presented a heuristic that arranged rectangles into strips on stock sheets. In 1980, [11], [12], presented algorithms which reduced the complexity of the problem by sorting rectangles into a pre-

defined order and packing these into the stock sheet using a bottom-left heuristic. These references and others that are relevant to this project are examined in the §2.2.

2.1.2 A Taxonomy of the Layout Problem

The major types of layout problem, as proposed in [6], are now introduced .Producing a layout requires assigning a finite stock of items to a given space, whilst preventing mutual overlapping between placed items. Historically, layout research has been carried out in two research fields: operations research and artificial intelligence. Most layout problems can be divided into two categories: minimisation of empty space within a fixed area, and the minimisation of area used for a fixed set of items.

2.1.2.1 Type 1: Fixed size, fixed contents

In this scenario, the aim is to fit a number of stock items into a fixed area and to conclude whether this is impossible to achieve. [6] states that these may be formulated as Quadratic assignment problems (QAP), if there is a one-to-one relationship between layouts and permutations of items. The QAP is NP-complete [7] and is unsuitable for 2-dimensional and 3-dimensional problems with varying-sized elements. This is due to the available positions depending on the elements have already been inserted.

2.1.2.2 Type 2: Varying size, fixed contents

In this scenario, all the stock items must be laid out, simultaneously minimising the total space, or the transportation of resources, between items. An example is the facility layout problem where a number of machine-areas are to be assigned to a factory floor-space. Areas may be defined as having fixed dimensions, or as having a range of acceptable aspect-ratio. Both definitions require a fixed area to be assigned. Traffic levels between all areas are estimated and these are used to bias the placement mechanism in favour of placing high-traffic areas in close proximity. Further taxonomy for machine layout problems has been described in [8]. Other problems include VLSI chip layout design, where wire lengths are also minimised.

A number of optimal algorithms have been developed to solve the layout problem, but due to the NP-completeness of QAP, these algorithms cannot be used to solve large-scale layout problems, e.g. problems with more than fifteen items [9]. A range of suboptimal algorithms

have been developed, including graph-theoretic algorithms, simulated annealing and genetic algorithms.

Graph-theoretic algorithms describe the problem in terms of a graph with nodes as states in the problem-solving process and edges as transitions between states, i.e. steps taken in search of a good solution. Brute force graph-searching techniques are not suitable due to the size of the search-space. Heuristic graph-searching use rules of thumb that guide the search to good solutions to a problem. Heuristics are used to select the most promising-looking branch of the search tree or graph. These branches may be examined first, before others, or may be used to cut out parts of the search tree altogether. The problem of finding heuristics that lead to good solutions limit their usefulness. Examples include: insert the items in order of decreasing size; insert an item in the smallest empty region where it fits; and attempt to fit larger items first, filling remaining space with small items.

2.1.2.3 Type 3: Fixed size, varying contents

In this scenario, the space to be filled is fixed and must be used as efficiently as possible by items from the given stock. Applications include cutting required shapes and sizes from textiles or glass. Additional constraints may be defined, such as the cutting machine may only make ‘guillotine cuts’, where a cut must be a straight line from one side of a piece to its opposing side with no corners on the way. The goal is to minimise wasted space arising from the layout. It is not necessary to place all the items from the stock in the layout, as multiple stock-sheets may be used. Solutions which minimise the number of sheets required are preferred.

These problems can be formulated as a variant of the knapsack problem, where the goal is to fill a sack as full as possible from the given stock, or fill the given stock in as few sacks as possible. For these problems the sacks have fixed size, but the stock items may vary in size.

The major difficulty, as reported by [10], has been encountered in developing a representation of the bin-packing problem that is suitable for a GA and is not limited to cuttings representable by slicing trees. The same authors reported using GAs successfully for improving solutions found by heuristic algorithms.

2.1.2.4 Guillotine versus Non-Guillotine Cutting Problems

Guillotine problems include all those regarding the flat-glass industry. Due to the scoring and break-out bar methods used to cut the glass, it is necessary to make all cuts from one side of a glass piece to the other. This restriction reduces the number of possible layouts and facilitates the use of algorithms such as branch-and-bound and tree-search. Single stage guillotine cutting restricts cuts to those that span the entire length of one dimension of the stock sheet. Multiple stage guillotine cuts allow recursive cuts to be made. In these cases, cuts are made end-to-end in pieces already cut.

Non-guillotine problems do not impose cutting restrictions and may be cut using a flame, for example, as in the sheet-metal industry. Figure 1 illustrates the different guillotine styles in use.

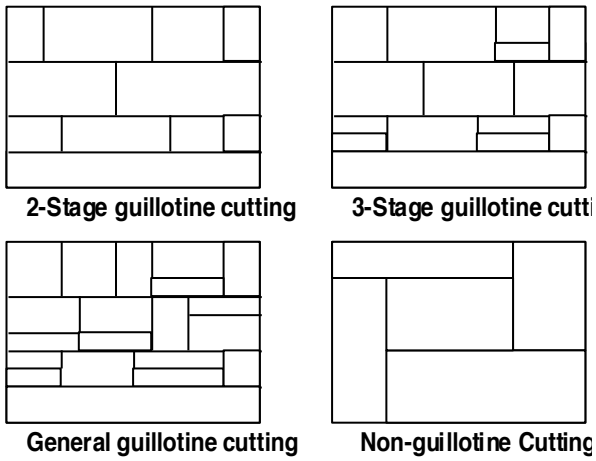

Figure 1. Different styles of guillotine cutting patterns.

This project is concerned with Type 3, non-guillotine layout problems which require near-optimal rectangular-piece placement on a stock sheet.

2.2 Literature Review of the Stock-Cutting Problem

The two-dimensional packing problem investigated in this project was first proposed in [12] in 1980. Given a collection of rectangles and a bin with fixed width and unbounded height, the rectangles are to be packed into the bin so that no two rectangles overlap and the height to which the bin is filled is as small as possible. This problem is a generalised form of the one-dimensional bin-packing problem, studied in [13]. If all rectangles have the same height then the two problems are equivalent. Alternatively, if all rectangles have the same width, the problem is equivalent to the makespan minimisation problem of combinatorial scheduling theory [14]. Both of these restricted problems are known to be NP-complete [14], [15], meaning that algorithms to solve them in polynomial-time have not been found. It follows that the two-dimensional packing problem is also NP-complete. It is for this reason why much research focusses on fast heuristic algorithms for solving this problem.

The first research in this area of stock-cutting [11], [12], devised efficient approximation algorithms for the non-guillotine cutting problem and derived the worst-case bounds on the performance of packings they produced. Both restrict their treatment to packings which are both orthogonal and oriented. An orthogonal packing is one in which every edge of every rectangle is parallel to either the bottom edge or the vertical edges of the bin. An orthogonal packing is also oriented if the rectangles are regarded strictly as ordered pairs, ie the rectangles are not allowed to rotate. Thus 90 degree rotations which preserve orthogonality are not allowed. The authors note that little published work has a bearing on the packing problem they have defined. [16] has shown that orthogonal packings of rectangles are not always optimum if rotation is disallowed.

[12] defines a class of packing algorithms, called *bottom-up left-justified* (BL) algorithms. Each algorithm packs the pieces one at a time as they are drawn in sequence from a list, *L*. When a piece is packed into the bin it is first placed into the lowest possible location and then it is left-justified at this vertical position in the bin. This is illustrated in .

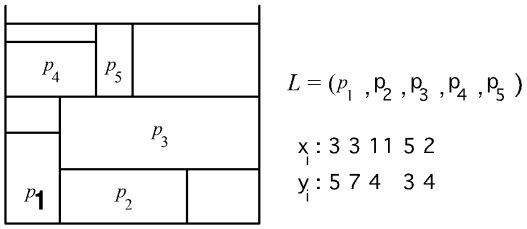

Figure 2. A layout obtained by using the 'bottom-up left-justified' layout algorithm. Bin has fixed width, layout grows from bottom of bin towards top.

The authors of [12] show that an upper-bound can be placed on bin height if the rectangles in list *L* are in decreasing-width order. The height of the packing is no more than three times the optimal bin height for rectangle pieces, and no more than twice as high for square pieces. Significantly, it is also shown that there are sets of pieces for which no BL algorithm can produce an optimal packing. That is to say, all possible list orderings yield a sub-optimal packing.

Figure 3a illustrates a layout presented in [12] which is optimally packed. Each piece is a square, labelled with its size. The authors prove that this is the only optimal packing (save for reflections around the axes) for this set of pieces. The authors make two modifications so that (i) the 3x3 square is now (3+ε)x(3+ε) and (ii) the bin now has width 15+ε. Figure 3b shows the resulting layout, which (to within ε) is optimal, according to the proof. However, in this layout there must be gaps between the 5x5 and 4x4 and between the 4x4 and 6x6 pieces which add up to ε; otherwise the 1x1 and (3+ε)x(3+ε) pieces cannot fit on top of the 4x4 piece. The authors of [12] state that no BL rule can produce these gaps, so the optimum packing of Figure 3b is unachievable with BL heuristics.

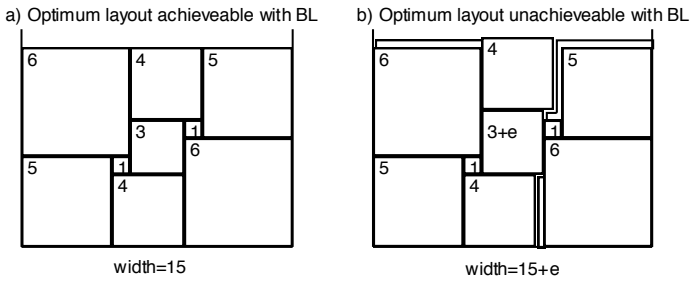

Figure 3. a) An optimum packing which is achievable using a BL algorithm. b) A slight increase in size of shape 3 of (a) generates gaps in the optimum layout. These are not achievable using a BL algorithm.

The authors of [12] raise a number of questions for future research. These concern the implementation of the BL algorithm, and include how to efficiently maintain the structure of available space as the packing sequence progresses. These questions are addressed in the algorithms developed in this report and the 'unachievable' layout of Figure 3b is obtainable.

[11] discusses the *Next-Fit Decreasing-Height*(NFDH) and *First-Fit Decreasing-Height* (FFDH) packing heuristics for level-oriented two-dimensional packing algorithms. In these algorithms, rectangles in the list L are ordered by non-increasing height and the rectangles are packed in the order given by L so as to form a sequence of levels. The first level (B_1) is the bottom of the bin and each subsequent level is defined by a horizontal line drawn from the top of the first (and hence maximum height) rectangle placed on the previous level. This corresponds with one-dimensional bin-packing; the horizontal slice determined by two adjacent levels can be regarded as a bin (lying on its side) whose width is determined by the maximum height rectangle placed in that bin.

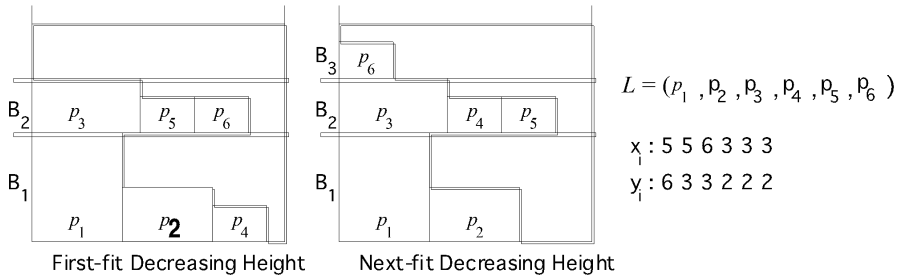

Figure 4. NFDH and FFDH algorithms applied to the list L=(r1,r2,r3,r4,r5,r6).

The use of NFDH or FFDH determines the order with which rectangles are packed. In NFDH, rectangles are packed left-justified on a level until there is insufficient space at the right to accommodate the next rectangle. At that point, the next level is defined, packing on the current level is discontinued and packing proceeds on the new level. In FFDH, each rectangle is packed left-justified on the first (ie lowest) level on which it will fit. If none of the current levels will accommodate this rectangle, a new level is started as in NFDH.

The essential difference between these algorithms is that whereas FFDH can always return to a previous level for packing the next rectangle, NFDH always places subsequent rectangles at or above the current level. The authors of [11] argue that BL methods turn out to be substantially worse than the level methods. They state that BL methods cannot improve on bin heights less than twice the optimal height, based on ordering list L by increasing or decreasing height or width. In contrast, they prove that FFDH can achieve close to 1.7 times the optimal height, when the list L is ordered by non-increasing height. Additionally, a split fit algorithm is presented that has a performance ratio of 1.5.

Also in 1980, [19] proposes an approach that combines heuristic and exact techniques to tackle the guillotine-cut problem. The solution is found by a sequential decision making process based on heuristics that first decides whether to fill a sheet or to replace the original problem by two subproblems, obtained by a vertical guillotine cut. The cut is made in such a way that the two solutions will imply a ‘good’ solution of the original problem; then the current problem is approximately solved by allocating groupings of pieces called strips. If the solution is not satisfactory, either a different subproblem generation is decided upon or an approximate solution will be produced, by applying a simple sequential placement procedure. During the process on the current sheet, at most three parts can be distinguished: the part that is already filled, the part currently under examination and the part still to be used. A new subdivision does not affect the first part, but does reduce the size of the second with respect to the third one. This process continues as long as there are pieces to be allocated.

The heuristics are tested on the same data as used in [18] and are shown to improve on their results. The tests use a large number of rectangular pieces (e.g. 398), but these are of a limited number of sizes and shapes (e.g. 8 varieties of piece). The algorithm exploits this to form large homogenous arrays of pieces of a single variety, which can then be arranged as a larger rectangular meta-piece in the layout. It could be argued that this makes layout-

generation easier. Using random data, a total percentage usage of the sheet ranges from 96.56% to 98.90%.

In 1982, [17] proposes a heuristic approach which restricts layouts to those attainable with one or more guillotine cuts, which is said to drastically reduce the number of layout combinations. Pieces are allowed to be rotated, but only by 90 degrees. The following questions are put forward: how should the huge set of possible layouts be reduced using a heuristic; how can the layouts belonging to this sub-set be enumerated, imposing some implicit or explicit ordering of the layouts to enable them to be scanned; and how is it determined whether a layout being formed has any chance of beating the best one found so far.

The chosen heuristic selects layouts where the pieces form 'piles without spurs'. Figure 5 illustrates such an arrangement. The pieces 1, 2, 3 and 4 form such a pile without overhanging parts or spurs, as the horizontal lengths decrease. After the addition of pieces 5, 6 and 7 to the pile, this condition is still fulfilled although these pieces are of increasing horizontal length. The sheet is filled from left to right in sections. This fulfils the first question asked and addressing the second question, a number of combinations are compared by starting with different pieces at the bottom of each section.

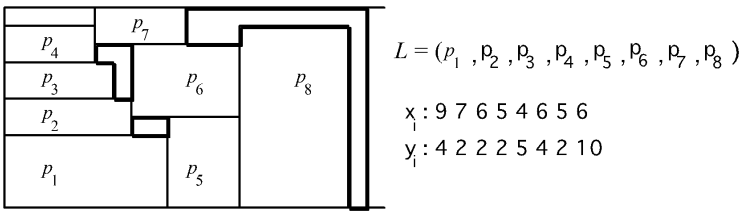

Figure 5. A possible arrangement of eight pieces, where pieces form 'piles without spurs'.

The author's test data was randomly created such that each sheet of raw material could hold a small number of pieces only, say between 5 and 20. A production plan may cover a few hundred of pieces of different sizes and shapes. The author notes good results and remarks that direct comparisons cannot be made to other algorithms as no previous papers deal with the precise problem. The main advantage of heuristic approaches is simplicity and small memory demands. Packing of irregular pieces may be achieved using a two-stage process, as suggested in [18]. Firstly, the pieces are encased in rectangular modules either singly or in

combination with other pieces. Then these modules are used in the second stage to produce layouts on the rectangular sheets.

In 1985 [20] developed a Lagrangean relaxation of a zero-one integer programming formulation of the non-guillotine two-dimensional cutting problem and used it as a bound in a tree search procedure. The author states that non-guillotine cutting problems had been considered by relatively few authors in the literature at that time, and indeed no other exact solution procedure for this problem existed in the literature. Results indicated that the Lagrangian-based tree search algorithm is capable of solving moderately sized non-guillotine cutting problems.

Guillotine cutting problems had been considered by a number of authors. The unconstrained problem (where there is no constraint imposed upon the number of cut pieces of the same size that are produced) has been solved by dynamic programming [21], [22], and by the use of a recursive (tree search) procedure [23]. [24] presented a tree search procedure for the constrained guillotine cutting problem. [25] lists references covering the problem.

[28] describes a 'usually adopted tactic' for tackling the stock-cutting problem using a genetic algorithm(GA, see §3.1). It is usual for the GA to use a gene coding which represents the two-dimensional position of patterns and the search is done two dimensionally. It is difficult to get proper results due to the size of the search space using this method [28]. Assuming a binary gene representation consisting of both the positions in the width and those in length to all of the patterns, the length of the genes is $N(\log_2 W+\log_2 L) = N\log_2(WL)$ and thus the size of the search space becomes $2\text{^}(N\log_2(WL)) = W^N L^N$, where N is the number of patterns, and W and L are the width and length of the sheet respectively.

[28] goes on to discuss a method for representing the stock-cutting problem as a (one-dimensional) permutation problem by incorporating layout determining algorithms (LDAs) – using a heuristic such as Bottom-Left – into GAs to reduce the dimension of the search space. The GA's task is to sample the search space and the LDA is designed to help the GA find the best solution as quickly as possible. Solving the problem as an ordering one, the size of search space becomes $N!$, that is, the permutation of N. It is the job of the LDA to process an ordering of shapes and to generate a layout which has no mutual overlapping and which also maximises sheet usage.

The GA determines the order of the shapes to be arranged on the sheet and the LDA fixes the position of each shape. Figure 6 illustrates the process. After positioning all shapes, the LDA returns the required sheet length (or the number of fixed-size shapes required) required to arrange the shapes. An efficiency usage for each sheet can be used to determine how fit a particular ordering is. The GA uses the fitness value to make the object value(the total sheet wastage) minimal, by genetic operations such as selection, crossover and mutation.

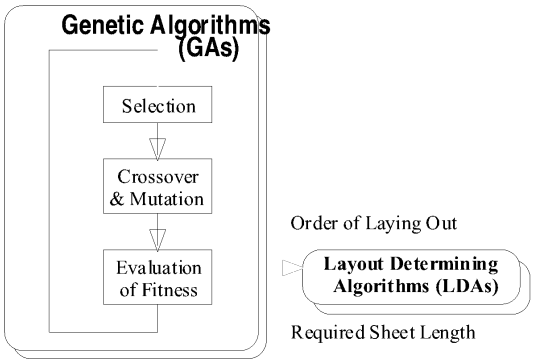

Figure 6. Configuration of a Genetic Algorithm system using a Layout Determining Algorithm to place shapes.

[28]concludes that combining GAs with layout determining algorithms to solve a two-dimensional problem as an one-dimensional order one, considerably reduces the required calculation time.

[26] describes the GGOAL Genetic Algorithm for the minimisation of glass loss in the guillotine cutting problem. Industial constraints stemming from both the glass cutting technology and from the requirement that the optimisation should run in real-time, concurrently with the cutting operation, are accommodated. These include minimum and maximum distances between adjacent cuts, minimum distance of cut from sheet borders and the capability to avoid specific internal points which are damaged.

A greedy placement algorithm is used, so that once a piece has been placed it is never reconsidered. The placement algorithm is only briefly mentioned. All scraps in the current sheet are considered and the one which ‘best contains’ the piece to be placed is chosen. All technological constraints are accounted for in the placement algorithm, freeing the GA from having to account for them in the fitness function.

An individual in the population of solutions consists of a permutation of genes, each containing the geometrical characteristics of a piece to be placed, an orientation flag and a placement criterion flag which links the piece with the previous one, if this is possible. An individual's fitness takes into account the total area of glass loss. For all sheets except the last, the ratio between the utilized area of the sheet and the total sheet area is evaluated. For the last sheet, the ratio between the area of the rightmost scrap (that can be re-used) and the total sheet area is evaluated. The fitness function is derived from the relevant evaluation function via linearization.

The crossover operator used is a uniform, order-based one [27]: a set of genes is extracted from the first parent, and copied to the newly generated individual in the same positions. Remaining genes fill the gaps, and are taken in the order in which they appear in the second parent. Mutation operators used include a single-swap mutator and mutators which flip either orientation or placement criterion flag.

The authors cite the following advantages of Genetic Algorithms: multiple constraints can be handled easily; an acceptable solution can be provided in the event of premature stoppage; and due to the iterative nature of the Genetic Algorithm, the code can be multithreaded easily.

In 1996 [33] described a set of algorithms for two-dimensional packing which uses no heuristics or layout determining algorithm. Permutation codings are argued to be limiting in the amount of meaningful information passed from one generation to the next. An alternative coding which contains all the information needed to produce the packing it represents is used. A two-dimensional matrix representation is presented, which preserves the notion of closeness between positions on the sheet. The solution space includes both feasible and infeasible solutions, so a number of different fitness functions which penalise infeasibility in different ways are developed. There results suggest that for small to moderately sized problems, simple one-dimensional crossovers are at least as good as, if not better than, more complex two-dimensional versions. It is also found that their genetic algorithm is not as good as traditional heuristic methods. It is found that their algorithm, although relatively efficient in terms of producing and decoding individuals, this is at the expense of larger populations and more generations.

Recent developments have included the work of [29] (1997), where an algorithm is designed whose performance ratio is constant for cases where rotation is allowed. In contrast, it is also shown that where rotations are not allowed there is no algorithm of constant ratio that exists. For this case an algorithm is designed with performance ratio of $O(\log(1/\varepsilon))$, where ε is the minimum width of any rectangle. It is also shown that no algorithm can achieve a better ratio than $O\left(\sqrt{\log(1/\varepsilon)}\right)$ for this case.

In [30] (1997) an efficient simulated annealing (see §3.5) search technique to solve non-guillotine, two- or three-dimensional cutting stock problems is presented. Simulated annealing was first proposed in [32] in 1983. Rectangles are placed in spaces which are large enough to accommodate them, and have shortest 'anchor' vector. The anchor vector is defined as a vector that points from the origin to the bottom-left hand corner of the piece. The system developed is tested on different sets of perfect non-guillotine cutting patterns, which is used to reassemble the cutting pattern. Perfect cutting describes cutting pieces on the stock sheet with zero trim loss. The algorithm is found to be good for small sized cutting problems (where there are less than 15 pieces to be cut). The algorithm does not guarantee to generate the optimal cutting pattern, but it can obtain an efficient and acceptable near-optimal solution.

In [31] the integration of the evolutionary algorithm technique with two-dimensional, non-guillotine cutting stock problems is examined. Crossover operations are not used, so the algorithm relies on mutation and selection of fit candidates to evolve a good solution. The test data is derived from making a perfect cutting pattern and then allowing the evolutionary algorithm to reassemble the pattern. The algorithm is found to be effective for cutting small pieces. For 10 pieces of paper to be cut from the stock sheet, the average trim loss was found to be 14.67%. For 20 pieces the average loss decreased from 14.67% to 7.93% but the computation time increased ten-fold. For 30 pieces the average loss decreased from 7.93% to 6.61% but the computation time increased a further three-fold. The authors argue that this techniique is considerably more effective than branch and bound techniques, and does not require a large amount of memory.

3 Introduction to Evolutionary Techniques

3.1 Genetic Algorithms

3.1.1 Introduction

Genetic Algorithms (GAs) are adaptive methods which may be used to solve optimisation problems. They are based on the genetic processes of biological organisms. Natural populations evolve over many generations according to the principles of natural selection and 'survival of the fittest', first clearly stated by Charles Darwin in *The Origin of Species*. By simulating this evolutionary process, GAs are able to 'evolve' solutions to real world problems, if the problems have been suitably encoded. The basic principles of GAs were first laid down rigourously by Holland, [34], initially for studying adaption in natural and artificial systems. Nowadays, the term GA is applied to any population-based search algorithm using selection and recombination operators to generate solutions in the problem space.

In order to understand GAs, it is necessary to have some appreciation of the biological processes on which they are based.

3.1.2 Biological Background

Whether in nature or anywhere else, evolution is not a directed process. For example, there is no evidence to support the assertion that the goal of evolution is to produce Mankind. The processes of nature appear to result in a haphazard generation of biologically diverse organisms. Some evolution is determined by natural selection, where different individuals are competing for resources in the environment. Some individuals are better, or fitter, than others and hence are more likely to survive and propagate their genetic material.

An *individual* (or *candidate*) in a *population* has its genetic material contained within a *chromosome.* The chromosome consists of a finite string of *genes*, which encodes a possible solution in a given problem space. Each gene carries information pertaining to a particular characteristic for the individual. The information-content is referred to as the *allele* for the gene. The *search space* comprises all possible solutions to the problem being investigated. GAs are applied to search spaces which are too large to be exhaustively searched. The

representation used for the chromosome is vital to the success of the GA as it must encompass the search space without being computationally cumbersome.

3.1.3 Mutation

In nature, asexual reproduction typically results in offspring that are genetically identical to the parent. Organisms, such as bacteria, which reproduce asexually evolve solely by *mutation.* Mutation is a sparse, random event that introduces small errors into alleles. Mutation may occur whenever chromosome material is copied (called the process of *transcription*) from parent to offspring. Mutation is used in GAs to prevent premature convergence to local optima by randomly sampling new points in the search space. This process introduces new genetic material into the gene-pool supporting greater diversity.

3.1.4 Crossover

Sexual reproduction allows some shuffling of chromosomes, producing offspring that contain a combination of genetic information from each parent. Typically, two parents' chromosomes are recombined to form an offspring by some kind of *crossover* operation. Crossover enables the evolutionary process to move toward 'promising' regions of the search space. Recombination occurs in an environment where the selection of who gets to mate is largely a function of the *fitness* of the individual, along with an element of chance.

The simplest crossover operator is the one-point crossover. Two parents are selected, and a single crossing point is chosen along the length of the chromosome. The two resulting segments of the chromosome are copied from parent one and parent two, respectively. For example, the child may be generated by copying all genes to the left of the cross-point from parent one, and all genes to the right of the cross-point from parent two, as shown in Figure 7.

Parent 1:	1 0 1 0 1 1 0 0 1
Parent 2:	1 1 1 0 0 1 0 0 0
Child:	1 0 1 0 0 1 0 0 0

Figure 7. One-point crossover where the chromosomes have a binary representation. The inverted segments are selected from each parent and are combined to form the child. A second child may be generated by combining the non-inverted segments from each parent.

The advantage of one-point crossover is that gene-sequences are minimally disrupted. However, the operator does not select all genes with equal probability. For example, the leftmost gene in the chromosome is far more likely to be selected from parent one than from parent two. This is because there is only one crossing point that prevents this scenario, and that is where the child is built entirely from parent two. A cross-point at any other location guarantees inclusion of the leftmost gene from parent one.

Two-point crossover is generally regarded as superior to one-point crossover. Two cross-points are selected, resulting in three segments. The leftmost segment may be obtained from parent one, the middle segment from parent two and the rightmost segment also from parent one. This may be viewed as a circular one-pt crossover which has the advantage of making all genes equally likely to be copied from either parent.

Parent 1:	1 0 1 0 1 1 0 0 1
Parent 2:	1 1 1 0 0 1 0 0 0
Child:	1 0 1 0 0 1 0 0 1

Figure 8. An illustration of two point crossover.

Uniform crossover, proposed in [47], randomly chooses which parent to copy from, on a per-gene basis. This method is not biassed to preserving short gene-sequences but this is at the cost of large disruption to most chromosomes.

Parent 1:	1 0 1 0 1 1 0 0 1
Parent 2:	1 1 1 0 0 1 0 0 0
Child:	1 0 1 0 0 1 0 0 1

Figure 9. An illustration of uniform crossover.

The edge-recombination operator in [48] is a special crossover operator used in permutation-based problems, such as the Travelling Salesman Problem. It is a suitable operator for experimentation with the stock-cutting problem.

3.1.5 Selection

The selection process should ensure that the fitter members of the population have more chance of reproduction. A popular selection method in GAs is to use a simple function that probabilistically selects individuals according to fitness. This is called *fitness-proportionate*

selection, and was first proposed in [34]. The probability that a chromosome is selected is determined by the ratio of chromosome fitness to total fitness present in the population. This method can encourage the production of super-fit individuals which tend to breed amongst themselves, limiting genetic diversity. In this situation, selection pressure is too high, and the population is effectively limited to the super-individuals. Problems also arise later in the evolutionary process when the population is converging. Many individuals become similarly fit so the selection amongst these is almost random. In this situation the selection pressure is too low.

A refinement of fitness-proportionate selection is proposed in [44] and is termed *windowing*. The fitness of the least-fit individual is found, a small threshold is subtracted, and the resulting value is subtracted from the fitnesses of each individual. This has the effect of exaggerating selection pressure and so overcomes the selection problems when the population is converging. However, super-individuals can still dominate with this method.

A selection method which attempts to overcome some of these limitations is *Linear Normalisation*, or *Rank Selection*, proposed in [44]. To make a selection, firstly the population is ordered according to decreasing fitness value and each individual is assigned a corresponding rank. The fittest chromosome in a population of x individuals is assigned a rank of x, the next fittest, a rank of x-1, and so on until the least fit individual is assigned a rank of 1. The probability that a certain individual is selected is proportional to the ratio of chromosome rank to total rank present in the population. For example, in a population of 100 chromosomes, the fittest individual is twice as likely to be picked than the median-fit individual and 100 times more likely to be picked than the least-fit individual. This method provides selection pressure no matter how close fitnesses are, preventing super-individuals from dominating.

The requirement for ordering individuals by fitness-value in rank selection prompted [46] to develop *tournament selection*. This results in the same distribution of chromosomes being selected when pairing off an entire population for reproduction, as for the rank selection method. Tournament selection requires a small subset of individuals (typically two or three) to be randomly selected from the population, their fitnesses are compared, and the winner is the fittest individual in the subset. This is easier to implement than rank selection and lends itself well to implementation on parallel processors.

3.1.6 Choosing when to apply crossover and mutation

The features that contibute most to evolution in GAs are the crossover operators used and the chromosome encoding and fitness function employed. Mutation is used to gradually introduce new genetic material into the population, and to generate a very diverse initial population by randomising genes. Crossover is used to rearrange existing genetic material into new configurations. The decision to propagate via mutation or crossover may depend upon the genetic diversity of selected parents. Crossover is more useful than mutation where selected parents are genetically diverse, and vice versa. This is because we would like each individual in the new generation to have a fair chance of containing good genetic material, whilst still allowing for genetic variation.

Grefenstette, [35], investigated whether there is an ideal set of parameters (in terms of crossover and mutation probabilities, population size, etc.) for a GA, but concluded that the basic mechanism of a GA was so robust that, within fairly wide margins, parameter settings were not critical. It was noted, however, that what is critical in the performance of a GA is the fitness function, and the coding scheme used. Holland, [34], first showed that the ideal scheme is to use a binary alphabet encoding. This has been extended in recent years to include character-based encodings, real-value encodings and tree representations [41].

3.1.7 Chromosome encoding – Holland's canonical GA

Holland's canonical GA was based on the theory of hyperplane sampling in order to find good, approximate solutions to optimisation problems. Chromosomes are encoded in binary and are decoded in order to evaluate the objective function. The result of this is processed to provide an overall fitness for the chromosome.

The schema theorem, devised by Holland, was the first rigorous explanation of how GAs work and why the crossover operator is good. A chromosome represents a sample of many different building blocks – called *schemata*. The *schema* is presented as a way of breaking-down a solution into its consituent schemata.

A schema consists of a string of symbols, representing each of the genes in a chromosome. The symbol alphabet consists of : { 0, 1, # }, where '0' and '1' represent genes with fixed allele, and the '#' representing a "don't care" situation. The schema *order* is defined as the number of bits that are of fixed value. Holland uses the schema theorem to conclude that a

binary-encoding provides the greatest amount of implicit parallelism when processing chromosomes. Holland favours binary representation over any other.

In [43] an argument is made for coding schemes of higher cardinality. They argue that such a scheme does not explicitly lead to the manipulation of so many schemata as a binary representation, but does implicitly represent more partitions of the search space.

In [45], real-valued chromosome encodings are favoured. These offset the loss of parallelism by the use of tailored recombination and mutation operators that would not be possible with a binary encoding.

3.1.8 Convergence

Convergence is the progression of members of a population towards increasing uniformity. A gene is said to have converged when 95% of the population share the same value, [36]. The population is said to have converged when all of the genes have converged. If a population has converged, or *stagnated*, then its genetic diversity has become limited. *Cataclysmic mutation* may be applied in this situation to rekindle evolutionary progress. This involves preserving a small number of the fittest individuals, and heavily mutating (by about 35%) the rest in the population.

3.1.9 What makes a GA good?

A GA is successful if it can propagate good *building-blocks* of genetic material from one generation to the next [40, p.41]. A successful coding scheme (or chromosome representation) is one which encourages the formation of building blocks by ensuring that:

1. related genes are close together on the chromosome, while
2. there is little interaction between genes.

As selection is biased towards fitter individuals, the GA should improve upon the initial population with each generation.

3.1.10 Generation Models

Creating a new generation may involve replacing the entire population, by pairing-off all individuals in the population, as in the *Generational Evolution Model*, supported by the

investigations in [35]. However, a more recent trend has favoured the *Steady-State Evolution Model* [42], where only one or two individuals are introduced into the original population, as in the. There is the advantage that the GA has the opportunity to exploit a promising individual as soon as it is created. This model requires a number of less fit (or less diverse) individuals to be removed at each generational step to keep the size of the population constant.

3.1.11 GA Evaluation

When the GA is implemented it is usually done in a manner that involves the following cycle:

- Evaluate the fitness of all the individuals in the population
- Create a new population by performing operations such as crossover and mutation, using fitness-proportionate selection to pick parents.
- Discard either all or a subset of the old population and iterate using the new population.

The first generation of this process operates on a population of randomly generated individuals.

Reproduction focusses attention on high fitness individuals, thus exploiting the available fitness information. Crossover and mutation operators perturb those individuals, providing general heuristics for exploration of the search-space. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust and powerful adaptive search mechanisms.

A GA is not a random search for a solution to a problem. Although GAs use stochastic processes, the result is distinctly non-random (and better than random). GAs are not guaranteed to converge but they are generally good at finding 'acceptably good' solutions to problems 'acceptably quickly'; the termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level.

Another optimisation technique, which maybe compared to GAs, is dynamic programming [37]. This is a method for solving multi-step control problems which is only applicable where the overall fitness function is the sum of the fitness functions for each stage of the problem,

and there is no interaction between stages [38]. The following sections describe some of the more general techniques.

3.2 Random Search

This is where points in the search space are selected randomly, or in some systematic way, and their fitness is evaluated. This method is rarely used by itself.

3.3 Hill Climbing

This technique relies on using information about the gradient of the fitness function to guide the direction of the search. It uses a population of one and uses a mutation operator to explore the search-space in the immediate vicinity of the current solution. It performs well on functions with only one peak (unimodal functions). On functions with many peaks (multimodal functions), the technique suffers from climbing only the first peak found – which may not be the global peak. The procedure for hill climbing is:

1) Initialise: generate a random solution *s*. This is the current solution. Evaluate it and let its fitness be *F*.
2) Mutate *s* to produce *s'*; evaluate *s'* and let its fitness be *f'*. If *f'* is good enough, STOP.
3) If *f'* >= *f* then copy *s'* to *s* (make *s'* the current solution)
4) Until termination condition reached goto (2)

3.4 Steepest Ascent Hill Climbing

This technique improves upon hill climbing, as the *neighbourhood* of the current solution is sampled and the best solution in this sample is moved to. The neighbourhood of a point *s* is the collection of points which can be reached via one application of the operator.

Iterated hillclimbing allows the hillclimbing algorithm to randomly choose a new starting point if a local maxima has been reached.

3.5 Simulated Annealing

The simulated annealing algorithm, proposed by Kirkpatrick [32], is based on the analogy between the process of finding an optimal solution of a combinatorial optimisation problem and the process of annealing a solid to its minimum energy state in statistical physics. Annealing is a process which finds the low energy state of a metal by melting it and then

cooling it slowly. Temperature is the controlling variable in the annealing process and determines the randomness and the energy state.

Simulated annealing is a local search algorithm. The searching process starts with an initial solution perhaps chosen at random. A neighbour of this solution is then generated by some mechanism and the change in cost is calculated. For a general local search process, if a reduction in cost is found the current solution is replaced by the generated neighhbour, otherwise the current solution is retained. The process is repeated until no further improvement can be found in the neighbourhood of the current solution, so the local search algorithm terminates at a local minimum. The simulated annealing searching process attempts to find a near-optimal minimum by probabilistic and deterministic acceptance strategies.

This is essentially a modified version of hill climbing, which addresses the problem of the solution getting stuck in local optima. Starting from a random point in the search space, a random move is made. If this moves takes us to a higher point, it is accepted. If it takes us to a lower point (within a tolerance, so the solution doesn't fallback too far), it is accepted only with probability *p(t)*, where *t* is time. The *acceptance function p(t)* begins close to 1, but gradually reduces towards zero – the analogy being with the cooling of a solid. The SA algorithm is:

1. Let the current solution, *s*, be a random solution, and set *temperature*, $T=T_{start}$
2. Iterate:
    - Mutate *s* to *new_s*, and evaluate *f(new_s)*
    - *D = f(new_s) - f(s)*
    - Accept or discard *new_s* depending on result of ***acceptance function***
    - Amend the ***cooling schedule***

The Acceptance Function: probability depends upon:

1. *d – difference* in fitness f-f' (bigger difference, less chance it will be accepted)
2. *T* – the *temperature* reduces with time, so less chance less-fit mutant being accepted as time goes on.
    - $p = e^{(-d/T)}$
    - Generate a random number between 0 and 1; accept the mutant if number is smaller than p

The Cooling Schedule: T starts high, e.g. 10000, and reduces with time, e.g. 0.001 final value, reducing according to the cooling schedule:

e.g. *cooling_interval=1000*, *cooling_ratio = 0.9*, T_{start} *= 10000,* T_{final} *= 0.001*

1. $T=T_{start}$
2. After each cooling interval: *T = cooling_ratio*T*
3. If $T<=T_{final}$ then STOP, else goto (2)

Initially, any moves are acceptable, but as the 'temperature' reduces, the probability of accepting a negative move is lowered. Negative moves are essential sometimes if local maxima are to be escaped, but too many negative moves will lead us away from the maximum.

Simulated Annealing is good as it is guaranteed to find the optimal solution, given one constraint. Unfortunately, this constraint requires the cooling schedule to be infinitely slow. The slower we cool (i.e. the larger the cooling ratio) the better, as it allows the genes to settle down gradually to achieve an optimal fitness.

Simulated annealing deals with only one candidate solution at a time, and so does not build up an overall picture of the search space. No information is saved from previous moves to guide the selection of new moves.

Simulated Annealing is generally better than Hill Climbing at finding good solutions, but the speed/quality tradeoff has to be considered. In many real-world situations, SA may not be fast enough.

3.6 Multiple Restart Hill Climbing

This provides a fairer comparison between Hill Climbing and Simulated Annealing, as it allows the Hill Climbing algorithm to restart if, after *i* iterations (say 1000), there has been no change in the best-fitness found so far:

1. Generate random initial solution *s*
2. Hillclimb from *s* until there have been restart_threshold successive iterations without any change in best-fitness so far.
3. Goto (1)

However, since each random trial is carried out in isolation, no overall picture of the 'shape' of the search-space is obtained. This means that trials are restarted equally in both low and high fitness regions. In contrast to this, a GA starts with a random population and increasingly conducts trials in regions of high fitness. In problems where the maximum is in a small region and is surrounded on all sides by regions of low fitness, this is a disadvantage. [39] finds such functions to be difficult to optimise by any method, preferring the simplicity of the iterated search.

4 Chromosome-Related Design

This section describes how the stock cutting problem is represented in terms of shapes, genes and chromosomes. Following this genetic operators, which manipulate existing chromosomes to generate new chromosomes, are discussed.

4.1 Shape Storage

There is exactly one *n* element array, called the *shape group*, which stores all shape specifications for cutting.

4.1.1 Shape Class

One element in this array is called a *shape* and holds the following specifications:

- integer width (*x_size*)
- integer height (*y_size*)
- shape identifier (*id* – integer in range 0 to *n*-1)
- boolean rotation flag (*rotation_ok*)

It is a requirement that all shapes have x_size and y_size smaller than the stock-sheet width and height, respectively. The id uniquely labels each shape in the shape group. The first shape in the shape group is assigned zero id. Ids increase incrementally along the shape group array. Rotation_ok is true if the shape, when rotated by 90 degrees, fits on an empty stock sheet. The shape class also provides a stream-input function which allows shape dimensions to be read from a file. Streamed input should be in the format: <x_size> <y_size>, with each field separated by white space.

4.1.2 Shape Group Class

The shape group offers the following services:

- can provide the length of the shape array
- a single shape array element can be accessed using the [] operator
- set_rotation_validities function – provided with the width and height of the stock-sheet, this function updates the rotation_ok flags of all its shape in the shape group.

4.2 Gene and Chromosome Representation

In general. a *chromosome* has *n genes*. The *chromosome* class is a base for creating the specialised derivative, *LDA_chromosome* class. The *gene* class is a base for creating the specialised derivative, *shape_gene* class. The base classes contains pure virtual functions which are pertinent to any specialised chromosome or gene that the programmer may wish to write. Derived classes override functions in the base class with code specific to the problem domain. This allows other parts of the application to interact with all types of chromosome in an identical way.

Most of the functions in the base classes are pure virtual which means they contain no code. These require a derivative class to provide code for these functions. The role of the base class is to define the interface for all its derivative classes. Some functions in the chromosome class are non-pure virtual and so do not have to be overriden. As the chromosome and gene classes contain pure virtual functions, objects of type chromosome and gene cannot be instantiated.

4.2.1 Gene Class (base)

The following services are defined in the gene class interface and all derivatives of this class must supply code to implement these:

- *create* – returns a pointer to a newly allocated gene, which has identical type to the gene used to call this function. The data in the new gene is not set
- *clone* – same as create, except the data in the new gene is an exact copy of the data in the gene used to call this function
- *get_gene_id* – returns the id of the gene
- *gene copy* – given a pointer to a source gene, the contents of the source gene is duplicated in the gene used to call this function
- *copy feature 1* – same as gene copy, except only feature 1 is copied
- *copy feature 2* – same as gene copy, except only feature 2 is copied
- *copy all features* – same as gene copy, except only features 1 and 2 are copied
- *randomize feature 1* – change feature 1 in a random way
- *randomize feature 2* – change feature 2 in a random way
- *randomize all features* – change features 1 and 2 in a random way

The copy feature operations are used by crossover operators, and the randomize feature operations are used by mutation operators. The create and clone operations are used to generate new genes with identical type to the calling gene.

4.2.2 Shape_gene Class

A single shape_gene contains the following data items:

1. *a shape pointer* (*p_piece*)– points to a single shape in the *shape group* to be laid out
2. *Feature 1*: an *orientation* boolean – signifies whether the shape is rotated 0 or 90 degrees
3. *Feature 2*: a *heuristic* setting – enumerated type which instructs the layout determining algorithm to place the shape using a particular heuristic

All the functions in the gene class are overriden appropriately. The get_gene_id function returns the id of the shape being pointed to by p_piece. Randomizing feature 1 involves setting the orientation to either 0 or 90 degrees randomly. Randomizing feature 2 involves setting the heuristic to : LEFTMOST, TOPMOST, ILEFTMOST or ITOPMOST. These heuristics will be explained later.

4.2.3 Chromosome Class (base)

The chromosome class encapsulates an ordered sequence of gene-type objects. The following services are defined in the chromosome class interface and all derivatives of this class must supply code to implement these:

- *create* – returns a pointer to a newly allocated chromosome, which has identical type to the chromosome used to call this function – the data in the new chromosome is not set
- *create_array* – same as create, except an entire array of chromosomes is allocated – this is used by the population class
- *create_loaded_array* – same as create_array except the pointer to a source array of chromosomes is provided, and this is used to supply gene data for the new chromosome array
- *clone* – same as create, except the data in the new chromosome is an exact copy of the data in the chromosome used to call this function – that is, all genes are copied
- *chromosome copy* – Given a pointer to a source chromosome, the contents of the source chromosome is duplicated in the chromosome used to call this function.
- *size_of* – returns the number of bytes used to store the chromosome
- single gene element access using the [] operator
- *fitness* – returns the cached fitness value for the chromosome if the invalidate_fitness function has not been called – otherwise, if the invalidate_fitness function has been called, the fitness value is recalculated before being returned
- *invalidate_fitness* – called when the genes of a chromosome have been changed in some way, requiring the cached fitness value to be re-calculated next time fitness function is called.
- *init_chromosome* – fills the genes in the chromosome with fixed, meaningful data
- *random_chromosome* – randomizes the gene data in the chromosome
- *ordered_chromosome* – sets the gene data in the chromosome to a particular setting
- *get_permutation* – Given a pointer to an array of integers, an ordered sequence of gene_ids are stored, corresponding to the order of the genes in the chromosome.
- *get_permuation_char* – Same as get_permutation except a character array pointer is provided and data is stored in a format suitable for outputting to the screen.
- *get_length* – returns the number of genes in the chromosome

- *distance* – Provided with a chromosome pointer, this function compares the supplied chromosome and the chromosome used to call this function, returning the hamming distance between the two.

4.2.4 LDA_chromosome Class

An LDA_chromosome has exactly *n* shape_genes where each gene uniquely refers to a single shape and all shapes are derived from exactly one shape group. The LDA_chromosome represents an ordered permutation of shapes, where each shape must be referenced exactly once. Each shape_gene carries orientation and heuristic data which instructs the layout determining algorithm how to place the shape.

Each LDA_chromosome contains cached fitness data which corresponds to the current fitness of the chromosome. If the chromosome is changed in any way, the code responsible for the change calls invalidate_fitness. This directs the fitness to be re-evaluated on the next occasion fitness data is requested from the chromosome. Caching fitness data increasing efficiency as fitness doesn't get re-evaluated if the chromosome hasn't been modified.

The distance function for LDA_chromosome compares the gene_id sequencing of the supplied chromosome and the calling chromosome. Distance begins at zero and is incremented for every mismatched gene_id. For example, if two genes place the same 5 shapes in the same order, but then place another 6 shapes in disparate order, then distance is calculated as 6.

All LDA_chromosomes share a static connection to an *LDA_environment* which contains the layout algorithms. This allows LDA_chromosomes to evalutate their own fitness and initialise themselves to a random permutation or pre-determined permutation. Fitness, init_chromosome, random_chromosome and ordered_chromosome, get_permutation and get_permutation_char function calls are processed indirectly by the LDA_environment.

All LDA_chromosomes also share a static length variable which sets the length of all LDA_chromosomes. The length corresponds to the number of shapes that are to be placed on the stock-sheet.

4.2.5 Gene and Chromosome Representation Summary

An LDA_chromosome consists of a permutation of shape_genes. Each shape_gene refers to one shape in a common shape group. Each shape_gene can determine the shape's orientation and placement heuristic. Chromosome fitness is calculated by an LDA_environment which is common to all LDA_chromosomes.

4.3 Genetic Operators - Crossover

These operators take features from two parents to generate a child, whilst maintaining the permutation property of the chromosome. Each operator is encapsulated in a class of its own, which is derived from the crossover class.

4.3.1 Crossover Base Class

All crossover operator classes are derived from the crossover base class. The class defines the interface for all crossover operators. All crossover classes have a name string used to identify the crossover operator. The interface requires the following functions:

- *cross function* – supplied with pointers to parent one, parent two and child chromosomes. Note that the cross function does not allocate memory for the child. The cross function performs the crossover operation pertinent to the class.
- *clone function* – returns a pointer to a new crossover-type object which has identical derived-type and data as that of the crossover object used to call the clone function

4.3.2 Order-based Crossover – order_c class

This operator selects a random collection of genes from parent 2. The allele-order of these genes is noted. The child is made identical to parent 1, except those alleles selected in parent 2 are reordered in the child according to their order in parent 2.

In the example below, the 1s in the selection mask label chosen genes, digits in the chromosome sequences represent gene-ids (allele data) for each gene, and underscore & italic are used to hilight genes.

Choose genes 0, 1, 3, 5 by location in parent 2:	1 1 0 1 0 1 0	(Selection Mask)
Chosen genes contain alleles 4, 3, 2, 1 in parent 2:	*4* *3* 0 *2* 5 *1* 6	(Parent 2 chromosome)
These alleles are affected in parent 1:	6 5 *1* *2* 0 *3* *4*	(Parent 1 chromosome)
The child contains a copy of parent 1, where affected alleles are reordered according to the order that they appear in parent 2 :	6 5 *4* *3* 0 *2* *1*	(Child chromosome)

A summary of the algorithm is as follows:

(Step 1) A random number of genes are chosen from parent 2 - the allele order is noted.

(Step 2) FOR all genes (*i*) in parent 1:

IF gene *i* in parent 1 contains an allele which isn't in one of the chosen genes

copy the allele in gene *i* of parent 1 to gene *i* of the child,

ELSE

get next uncopied allele in ordered set of chosen genes from parent 1

copy this allele to gene *i* of the child.

4.3.3 Segmented Order-based Crossover – seg_order_c class

This operator is identical to the order-based crossover, except that:

- the number of genes selected in the selection mask is randomly chosen between a minimum and a maximum setting
- these chosen genes form a continuous segment, eg. `0 0 1 1 1 0 0`, has a segment length of 3.

By preserving segments of chromosome, this operator may be more successful than the standard order-based crossover.

4.3.4 Position-based Crossover – position_c class

This operator selects a random collection of genes from parent 2. These genes have their alleles copied directly into the corresponding genes in the child. Remaining genes in the child are filled in-order with alleles that haven't yet been copied, by scanning genes in-order from parent 1.

In the example below, the 1's in the selection mask label chosen genes, digits in the chromosome sequences represent gene-ids (allele data) for each gene, and underscore & italic are used to hilight genes.

Choose genes 2, 3, 4, 6 by location in parent 2:	0 0 1 1 1 0 1	(Selection Mask)
Chosen genes contain alleles 4, 7, 6, 3 in parent 2:	1 2 *4* *7* *6* 5 *3*	(Parent 2 chromosome)
These genes are copied from parent 2 to the child:	- - *4* *7* *6* - *3*	(Child incomplete)
Genes to be copied, preserving order, from parent 1:	6 *1* *5* 4 3 7 *2*	(Parent 1 chromosome)
Result after copying genes from parent 1 to child:	*1* *5* 4 7 6 *2* 3	(Child complete)

A summary of the algorithm is as follows:

(Step 1) A random number of genes are chosen from parent 2.

(Step 2) FOR all genes in the set of chosen genes (*i*):
copy gene *i* of parent 2 to gene *i* of the child

(Step 3) FOR all genes (*j*) in parent 1:
IF allele in gene *j* is NOT already in child
copy gene *j* into the *leftmost unoccupied gene* in the child *
ELSE
skip this iteration

* *leftmost unoccupied gene* – the leftmost gene in the child chromosome that has yet to be copied into.

4.3.5 Segmented Position-based Crossover – seg_position_c class

This operator is identical to the position-based crossover, except that:

- the number of genes selected in the selection mask is randomly chosen between a minimum and a maximum setting
- these chosen genes form a continuous segment, eg. 0 0 1 1 1 0 0, has a segment length of 3.

By preserving segments of chromosome, this operator may be more successful than the standard position-based crossover.

4.3.6 Half-uniform Crossover – HUX_position_c class

This crossover operator is identical to segmented position-based crossover, except the segment is either the first half or the second half of the chromosome.

4.3.7 Edge-Recombination Crossover – edge_recombination_c class

The child chromosome is derived from the two parents as shown in the following example:

1. *Build **Edge Map** for parent1* (3716524):
   The *edge map* shows the 'edges' or 'neighbours' associated with each allele. The allele is defined as the gene-id of each gene. The chromosome is treated as circular, hence the neighbours of the '3' allele are 4 and 7.

Alleles:	1	2	3	4	5	6	7
Edges:	6,7	4,5	4,7	2,3	2,6	1,5	1,3

2. *Build **Edge Map** for parent2* (2761534):

Alleles:	1	2	3	4	5	6	7
Edges:	5,6	4,7	4,5	2,3	1,3	1,7	2,6

3. *Combine the two Edge Maps:*

Alleles:	1	2	3	4	5	6	7
Edges:	5,6,7	4,5,7	4,5,7	2,3	1,2,3,6	1,5,7	1,2,3,6

4. *Pick random parent for first allele of child and remove allele 3 from edge map:*
   Child: 3******

Alleles:	1	2	3	4	5	6	7
Edges:	5,6,7	4,5,7	4,5,7	2	1,2,6	1,5,7	1,2,6

5. *Find edge map for current allele, 3: 4,5,7 – choose allele with smallest no. of edges (4), and remove allele 4 from edge map.*
   Child: 34*****

Alleles:	1	2	3	4	5	6	7
Edges:	5,6,7	5,7	5,7	2	1,2,6	1,5,7	1,2,6

6. *Find edge map for current allele, 4: 2 – choose allele with smallest no. of edges (2), and remove allele 2 from edge map.*

   Child: 342****

Alleles:	1	2	3	4	5	6	7
Edges:	5,6,7	5,7	5,7		1,6	1,5,7	1,6

7. *Find edge map for current allele, 2:5,7 – choose allele with smallest no. of edges (randomly choose, say 7), and remove allele 7 from edge map.*

   Child: 3427***

Alleles:	1	2	3	4	5	6	7
Edges:	5,6	5	5		1,6	1,5	1,6

8. *Find edge map for current allele, 7:1,6 – choose allele with smallest no. of edges (randomly choose, say 1), and remove allele 1 from edge map.*

   Child: 34271**

Alleles:	1	2	3	4	5	6	7
Edges:	5,6	5	5		6	5	6

9. *Find edge map for current allele, 1:5,6 – choose allele with smallest no. of edges (randomly choose, say 5), and remove allele 5 from edge map.*

   Child: 342715*

Alleles:	1	2	3	4	5	6	7
Edges:	6				6		6

10. *Find edge map for current allele, 5:6 – choose allele with smallest no. of edges (6), and remove allele 6 from edge map.*

    Child: 3427156

Alleles:	1	2	3	4	5	6	7
Edges:							

Designed for the Travelling Salesman Problem, this operator is good at preserving edges – ie keeping together groups of alleles. However, there are two ways that an edge can be preserved and it remains to be seen if this operator will work well with the stock-cutting

problem. This is because both groups of alleles and their order need to be preserved in the stock-cutting problem.

4.3.8 N-point Feature Crossover – n_point_c class

All the crossover operators discussed so far have combined characteristics of the gene permutations present in both parents. This crossover operator performs an n-point crossover on either feature 1, feature 2, or both features of both parents. Feature 1 corresponds to the orientation flag and feature 2 corresponds to the heuristic setting in each shape_gene.

The class has two parameters controlling the crossover:

1. *N_point* – sets the number of crossing points
2. *copier function pointer* – sets the copy function to be used, either feature_1, feature_2 or all_features

The child is first created from a clone of parent 1. The crossover operation then affects only the features specified in the copier parameter.

4.4 Genetic Operators - Mutators

These operators make changes to a single chromosome, whilst maintaining the permutation property of the chromosome. Each operator is encapsulated in a class of its own, derived from the mutation class.

4.4.1 Mutation Base Class

All mutation operator classes are derived from the mutation base class. The class defines the interface for all mutation operators. The following data members are common to all mutation classes:

- *name string* – identifies the type of mutator by an alphanumeric name
- *in_situ_mutation* – boolean indicating whether mutation should occur within the parent chromosome itself, or whether a separate child chromosome should store the result
- *finished* – boolean set by mutation operators which have state
- *restart* – boolean set by mutation operators which have a restart status

The finished and restart flag are not used by the mutation operators described here. They are used in simulated annealing and hill climbing which is covered in a later section.

The mutation class interface requires the following functions:

- *mutate*– supplied with pointers to parent one and child chromosomes. Note that the mutate function does not allocate memory for the child. The mutate function performs the mutation operation pertinent to the class.
- *clone*– Returns a pointer to a new mutation-type object which has identical derived-type and data as that of the mutation object used to call the clone function.
- *reset_mutator* – Performs any initialisation, eg used to reset the temperature variable in simulated annealing.

The following function is defined for the mutation class:

- *set_in_situ_mutation* – determines whether the mutation operation occurs in the parent or in a child copy of the parent. In-situ mutation is possible when performing simple mutations, such as swap mutations.

4.4.2 Single-gene Swap Mutation – swap_m class

This operator picks one pair of genes at random and swaps their locations in the chromosome.

4.4.3 Multiple-gene Swap Mutation – multiple_swap_m class

This operator picks *n* pairs of genes at random and swaps their locations in the chromosome. The operator guarantees that any single gene can change position a maximum of once. The operator can be provided with the value for *n* or a percentage corresponding to the amount of the chromosome to be mutated. For example, swapping 25% of chromosome corresponds to 5 swaps (each involving two genes) in a 40-gene chromosome.

4.4.4 Inversion Mutation – invert_m class

This operator first selects two *sentinel* genes in the chromosome. The string of genes between and including the *sentinel* genes have their ordering in the chromosome reversed. The example in Figure 10 shows the inverted string italicised, where the chromosome is a permutation of 10 shapes, labelled ‘0’ through ‘9’.

Parent: 0 1 2 3 4 5 6 7 8 9 Sentinel genes: 4, 7
Child: 0 1 2 3 *7 6 5 4* 8 9

Figure 10. Example of inversion mutation.

4.4.5 Shunt Mutation – shunt_m class

This operator first selects two *sentinel* genes in the chromosome. A third gene is chosen as an insertion location. The string of genes between and including the *sentinel* genes is moved to the insertion location in the chromosome. Genes not in this string are shunted to make room for the string. The example in Figure 11 shows the shunted string italicised, where the chromosome is a permutation of 10 shapes, labelled '0' through '9'.

Parent: 0 1 2 3 4 5 6 7 8 9 Sentinel genes: 4, 7 insertion gene: 1
Child: 0 *4 5 6 7* 1 2 3 8 9

Figure 11. Example of shunt mutation.

4.4.6 Standard Mutation – standard_m class

This mutation class makes random changes to the features (ie orientation and/or heuristic settings) pertaining to a fixed number of randomly picked genes in the parent chromosome. It has two parameters:

1. *Gene count* – determines the number of genes to randomize.
2. *randomizer function pointer* – determines the randomization function to use: randomize_feature_1, randomize_feature_2 or randomize_all_features.

Feature 1 corresponds to the orientation flag. Feature 2 corresponds to the heuristic setting.

4.4.7 Cataclysmic Mutation – cataclysmic_m class

This mutation operator is applied to an entire population of chromosomes, instead of individual chromosomes. The fittest chromosome is left untouched by this operator. All other chromosomes are mutated by a percentage, such as 35%. This is achieved using the Multiple-swap Mutator. For example, if the chromosome length is 100 and mutation percentage is 35%, then the number of swaps required is 17.5 – which is rounded down to 17. Thus, in total, 34 genes are swapped.

This mutation operator has a single parameter:

- *mutate_features* – boolean which signifies whether orientation and heuristic settings should be mutated, in addition to the gene permutation.

The cataclysmic mutator uses a standard mutation object (called *feature_mut*) to perform feature mutation. If feature mutation is required, then the randomizer parameter of the feature_mut has to be set appropriately.

As the cataclysmic mutation operator does not operate on single chromosomes, the standard mutate function performs no operation. In order to apply this operator to a population the following function is called:

- *global_mutate* – requires both a pointer to the population to be mutated and the percentage change required (supplied as a floating point number in the range 0 to 1, where 0 is no change and 1 is 100% change) to be provided.

5 Environment-Related Design

The following sections describe the layout determining algorithm §5.1 and the environment classes, §5.2 and §5.3, which allow the LDA to be used with the GA. Terminology is grouped in sections, which are situated in places where further explanation is required.

5.1 Layout Determining Algorithms

5.1.1 Introduction

Let us suppose that we have an algorithm for placing individual shapes on the sheet, called the Layout Determining Algorithm (LDA). The LDA's job is to place single shapes in positions which favour efficient sheet usage.

A very simple LDA (equivalent to the level-oriented Next-Fit First Descending algorithm) builds its shape layout in columns. Figure 12 illustrates the layout of eleven shapes on a stock-sheet with dimensions x-size by y-size. Starting from the left side of the sheet, columns abut across the sheet. A single column is built from the top of the sheet to the bottom. A column's height accumulates as shapes are added to the bottom of the column by the LDA. A column's width is equal to the width of the widest shape in that particular column. Once a column grows too far down the sheet, such that the next shape will not fit between the bottom of the column and the bottom of the sheet, a new column is started to the right of the old column.

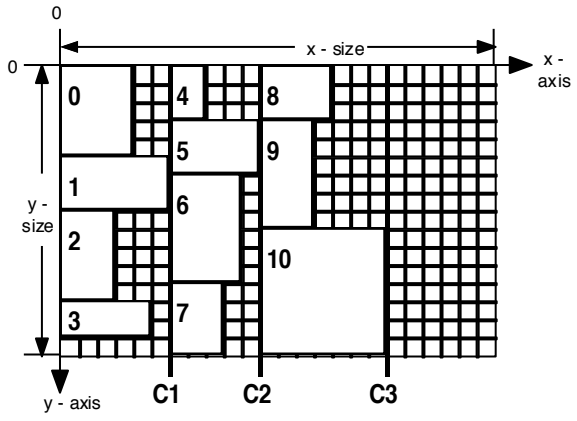

Figure 12.The layout of 11 items using the simple LDA. Annotation numbers signify order with which shapes were placed, beginning with zero. Column boundaries are labelled C1, C2 and C3. Column width is set by the widest shape in the column. The scale for all layout diagrams is a grid-square is one logical coordinate high and wide.

- **Terminology (Part 1)**

Examples of how the terminology (italicised) is used to describe the relative positions of shapes throughout this project (referencing Figure 12):

- shape 0 is ***behind*** shapes 4 and 5
- shape 8 is ***in-front-of*** shape 4
- shape 4 is ***above*** shape 5
- shape 5 is ***below*** shape 4
- the ***placement position*** of shape 0 is (0,0) (Top-left vertex)
- the ***x-offset*** of shape 1 is 0 (x component of placement position)
- the ***y-offset*** of shape 4 is 0 (y component of placement position)
- the ***top-*** and ***bottom-faces*** of shape 0 are on the lines x=0 and x=5, respectively
- the ***left-*** and ***right-faces*** of shape 0 are on the lines y=0 and y=4
- the ***sheet*** has an area of ***x-size . y-size***

In more advanced LDAs, heuristics are used to direct shape placement. A typical placement heuristic is to place a shape in the ‘leftmost’ area of the sheet which is entirely unoccupied by any previously placed shapes. A ‘topmost’ heuristic would operate in a similar way. A possible regime for placing shapes would be to alternate between these two heuristics, beginning with say ‘leftmost’.

Consider a regime where the ‘leftmost’ heuristic is always used. The sheet would get filled-up in a sweep from left to right, building in an approximate columnar arrangement from top to the bottom of the sheet. If there are two positions which are equally ‘leftmost’ then the LDA favours the position which is ‘topmost’ out of these two.

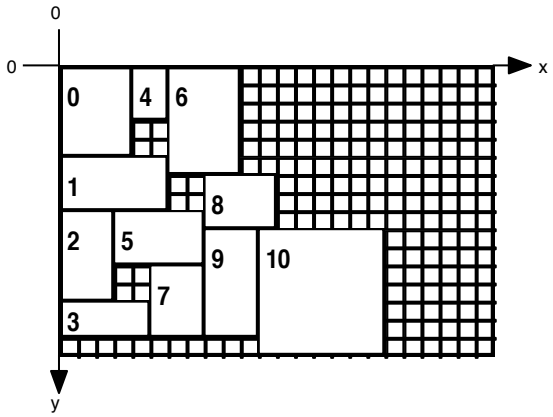

Figure 13. A greedy placement algorithm uses the 'leftmost' heuristic to layout the shapes used in Figure 12.

A regime which always uses the leftmost heuristic may favour placement of narrow, tall shapes. It may be biased against wide, short shapes due to the columnar construction of the layout. The same argument applies to a constant 'topmost' regime.

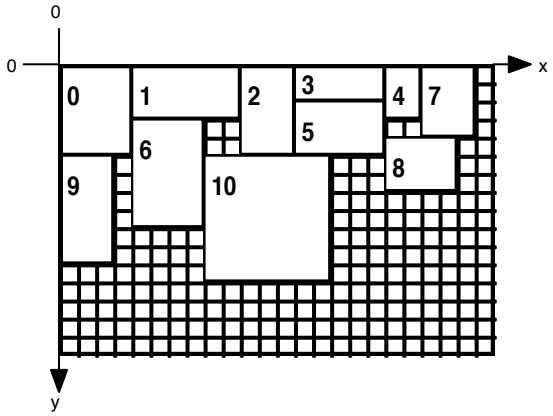

Figure 14. A greedy placement algorithm uses the 'topmost' heuristic to layout the shapes used in Figure 12.

Alternating the regime favours neither row or column building, so may be more capable of searching a large search space.

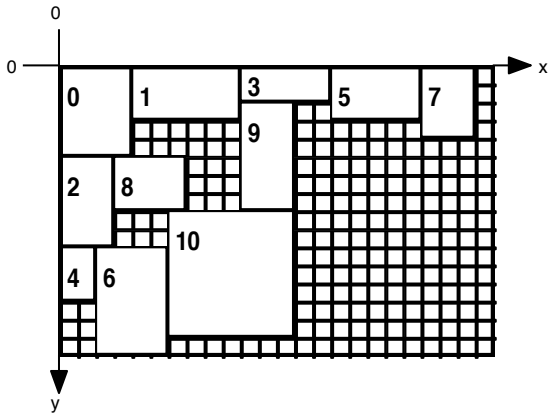

Figure 15: A greedy placement algorithm alternates between leftmost and topmost heuristics, beginning with leftmost, to layout the shapes used in Figure 12

Alternatively, a refinement to flipping between 'leftmost' and 'topmost' is to store a heuristic inside every shape_gene. The LDA would then extract the heuristic setting from the shape_gene before placing the corresponding shape. This dynamic regime would offer the chromosome three methods for diversifying:

1. Exploring permutations of shape_genes
2. Manipulating the orientation flag of each shape_gene
3. Manipulating the placement heuristic of each shape_gene

To summarise, the simplest placement regime is to place all shapes 'leftmost' or 'topmost'. This may be refined to alternating the heuristic between 'leftmost' and 'topmost', providing a more balanced placement mechanism. This may be further refined by allowing individual shape_genes to dictate individual placement heuristic, enabling the GA to evolve its own dynamic placement regime. An example dynamic regime consisting of five shapes is: left, top, top, left, top.

5.1.2 LDA Features – Overview

The LDA's purpose is to make the search-space of possible layouts smaller. Erroneous layouts with overlapping shapes are eliminated from the search-space by using the LDA. This makes the GA more likely to find a good optimisation. Without the LDA, the GA would initially create random locations, in 2-dimensions, for the shapes. Overlapping shapes would be inevitable and layouts with at least one pair of overlapping shapes would form the

majority of the search-space. Layouts with zero overlapping pairs would be sparsely scattered throughout the search-space, resulting in unfeasible conditions for evolving a good solution.

The LDA developed keeps track of the convex area encompassing free-space on a sheet. Placing a shape involves the LDA searching for sufficiently sized gaps in free-space in which the shape may fit. A choice is then made as to which gap to use, directed by the placement heuristic. A record of currently placed shapes and their locations is kept, whilst the free-space representation is updated on each placement. The LDA places all shapes in the order described by a chromosome, and subsequently calculates fitness metrics for the layout. These are then used to direct selection operators in the GA. The use of an LDA reduces the optimisation from a 2-dimensional problem to a 1-dimensional permutation-based one.

The LDAs designed for this project have the following features:

1. ***Layout:*** A record of which shapes have been placed where on the sheet. Allows the entire layout to be examined once all shape_genes in a chromosome have been placed.
2. ***Free-Space Representation:*** Used to determine where in the current sheet new shapes can be placed.
3. ***Placement Builder:*** An analyser for (2) which yields all possible placement positions for a particular shape.
4. ***Placement Chooser:*** Picks a single placement position from (3) which satisifies the current placement heuristic, say ‘leftmost’.
5. ***Free-Space Updater:*** Adjusts (2) to take account of a newly placed shape.
6. ***Layout Updater:*** For adding the shape and its location to (1)
7. ***Sheet Creator:*** Processes the situation where the current sheet has insufficient space to place the next shape, in where (3) yields no placement positions.
8. ***Fitness Evaluator:*** Returns fitness value for the current layout corresponding to how efficiently sheets are used. This is calculated by considering the final state of (2) after all shape_genes in the LDA_chromosome have been placed.

The following section details design and implementation issues for each of these features.

5.1.3 LDA Features – Design and Implementation

5.1.3.1 Layout

This is a record of which shapes have been placed where on the sheet. This allows the complete layout to be analysed after all shape_genes in a chromosome have been processed.

To satisfy this, an array of 'layout_pieces' is used. A single layout_piece contains:

1. a copy of the shape_gene enhanced with an (x,y) placement coordinate, encapsulated into a shape_gene_l object
2. a boolean (*new_sheet_needed*) which is true if the layout_piece requires a new blank sheet to be used.

The first element in the layout_piece array always has its boolean set to true. If all shapes in the chromosome have been placed onto a single sheet then all other layout pieces in the array have their booleans set to false. If the *xth* layout_piece cannot fit on the current sheet its boolean is set to true and an additional sheet is created for placement of the *xth* shape and any other unplaced shapes. As many sheets as are required are created as each sheet becomes full. Layout_pieces which fit on the current sheet have their boolean set to false.

5.1.3.2 Free-Space Representation

5.1.3.2.1 Introducing the left-profile

This representation is used to determine where in the current sheet new shapes can be placed. Firstly, we shall consider a representation which can work with the leftmost heuristic. The free-space representation trades-off completeness against computational complexity, and is required only to describe a 'good' proportion of the free-space available for new shapes. Figure 16 illustrates a representation for free space. All shapes have been placed using the leftmost heuristic.

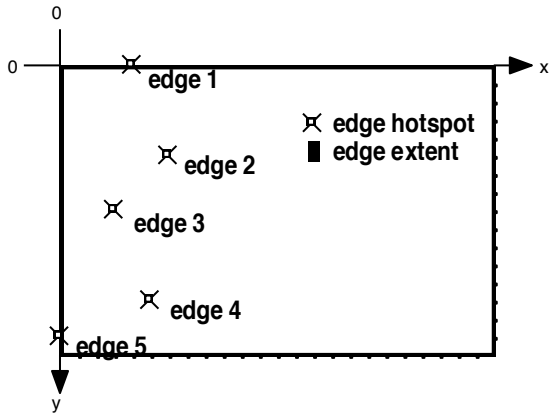

Figure 16. A representation for describing a convex outline of the free space remaining on the sheet.

An ‘edge’ exists wherever a shape has a segment of its right-face in line-of-sight (in a direction along the x-axis) with the right-face of the sheet. Each edge has a length (equal to the segment length) and an (x,y) coordinate corresponding to the edge’s upper-extent, known as the *hotspot* – marked with ‘⌑’ in the figure. Edges are stored in a linked list of edge nodes where nodes are ordered according to increasing *y* location of hotspot, *y-edge-offset*. This edge list is know as the *left-profile*. The left-profile maintains the list of edges which defines the left-most limit of the free space available on the sheet. No two edges may overlap in the y-axis, and the left-profile represents the entire extent of the y-axis without discontinuity. The total length of all edges in the edge list is equal to the height of the sheet (the y-size of the sheet).

Figure 17 illustrates how both the current layout and the left-profile is updated as each shape is added to the layout, using a leftmost placement heuristic.

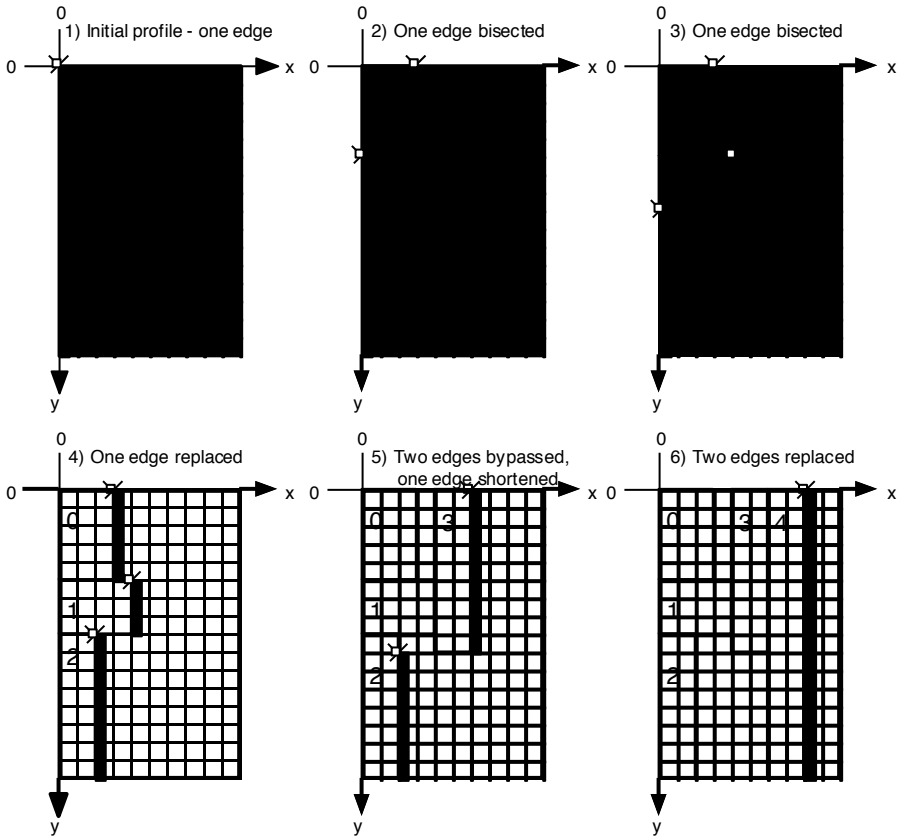

Figure 17. Layout construction and left-profile update illustration. In step 1, no shapes have been placed yet and there is one edge (hotspot=(0,0); length=y-size). Adding shape 0 and shape 1 both result in a single edge being bisected. Adding shape 2 results in an edge having its *x* location of hotspot, its *x-edge-offset*, increased by the width of the placed shape. Adding shape 3 bypasses edges 1 (top edge) and 2, whilst displacing and shortening edge 3. Adding shape 4 bypasses both remaining edges with a single edge equal in length to sheet y-size.

5.1.3.2.2 Limitations of the left-profile

The left-profile is good for placing shapes using the leftmost heuristic. However, when using a topmost heuristic areas of free-space are *masked*, making them unavailable for placement. Figure 18 shows how successively top-placed shapes mask areas behind the left-profile.

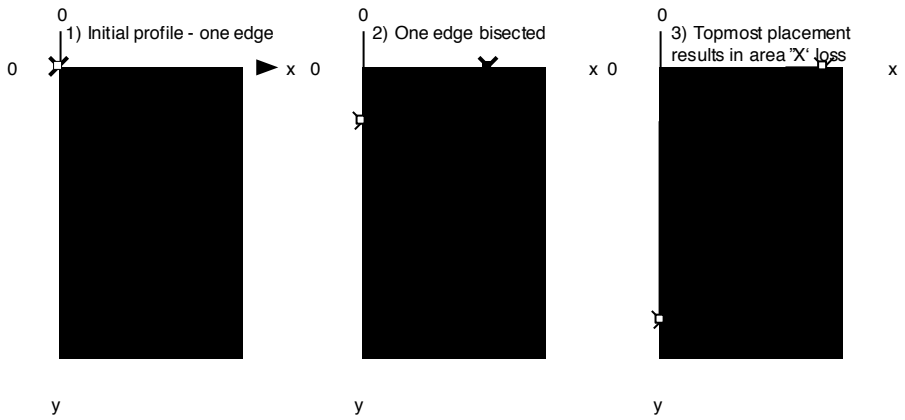

Figure 18. The addition of shape (1) in step (3) renders area 'X' invisible to the left-profile.This area is lost from the free-space representation. Invisible areas make it harder for the LDA to create efficient layouts. Free-space is masked from the left-profile when there is a shape to the right of the free-space.

5.1.3.2.3 Overcoming left-profile limitations – introducing the top-profile

The top-profile is a method by which free-space may be represented more effectively when used in-conjunction with a left-profile. Figure 19 illustrates the top-profile representation. All shapes have been placed using the topmost heuristic.

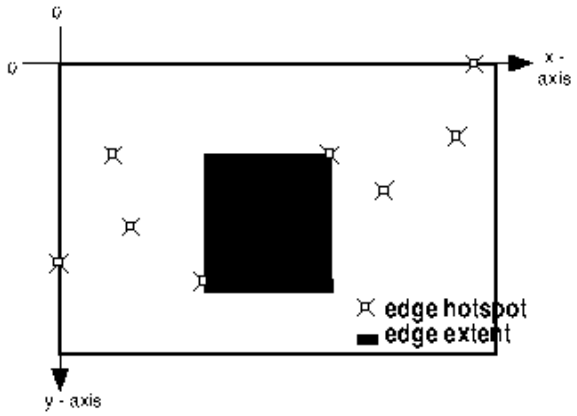

Figure 19. Top-profile is equivalent to left-profile rotated through 90 degrees. An edge hotspot corresponds to the left-extent of the corresponding edge, and edge length is measured in the x-axis. Edges are ordered according to increasing x-edge-offset in the top profile. The top-profile defines the topmost limit of the free space available on the sheet and considers the positioning of the bottom-faces of placed shapes. This is in contrast to the left-profile, which considers right-face positioning. The total length of all edges in the top profile edge list is equal to the width of the sheet (the x-size of the sheet).

Before any shapes are placed, the top profile consists of a single edge (hotspot=(0,0); length=x-size), which spans the x-axis.

Top-profile is good for placing shapes using the topmost heuristic alone. Using the leftmost heuristic, with the top-profile representation, results in area masking as illustrated in Figure 20.

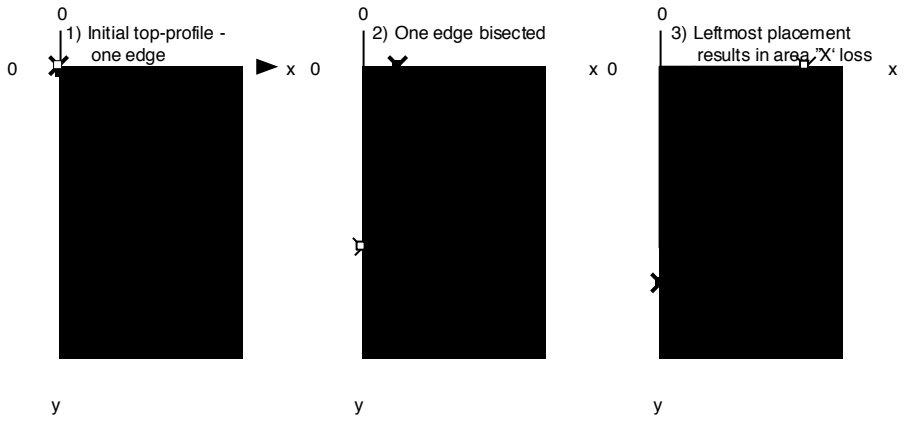

Figure 20. The addition of shape (1) in step (3) renders area 'X' invisible to the top-profile. This area is lost from the free-space representation. Invisible areas make it harder for the LDA to create efficient layouts. Free-space is masked from the top-profile when there is a shape below the free-space.

Once an area becomes masked it is no longer represented by any of the edge profiles. This restricts the number of places that the LDA can fit shapes. Although the space may be free, when masked from all profiles it cannot be utilised, so it is wasted reducing fitness.

5.1.3.2.4 Combining both left- and top-profiles

By using both a left-profile and a top-profile (independent of each other), a free-space representation better than either of these methods alone is possible. Using this dual-representation allows the previously masked 'X' areas in both Figure 18 and Figure 20 to be represented as free-space. When using the left-profile alone, free-space is masked when there is a shape to the right of the free-space. Similarly, when using the top-profile alone, free-space is masked when there is a shape below the free-space. The advantage of using the dual-representation is that only the free-space that has *both* a shape to its right *and* a shape below becomes masked. Figure 21 (parts (a) and (b)) shows how the final layouts in Figure 18 and Figure 20 are represented using two profiles – the shaded areas are no longer masked. Part (c)

of Figure 21 illustrates the limiting case where a free-space area has both a shape to its right and a shape below it.

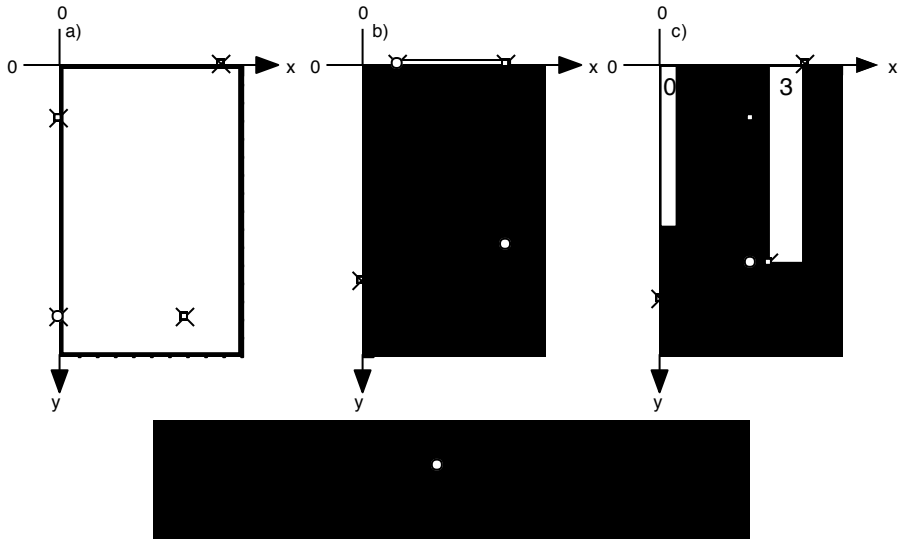

Figure 21. (a) Layout from Figure 18, step (3). The dual-representation prevents the shaded area from being masked, whereas it is in the left-profile only case. (b) Layout from Figure 20, step (3). The dual-representation prevents the shaded area from being masked, whereas it is in the top-profile only case. (c) This layout has been placed using a top-left heuristic, beginning with topmost, and illustrates the limitation of the dual-representation. The masked area cannot be represented by either the left-profile or the top-profile and so is unavailable free-space.

It is thought only a complete constrained-polygonal representation can improve upon the dual-representation. The ‘constrained’ polygon would be composed of lines parallel to one of the axes. This representation would be considerably more complex and so we return to the completeness against complexity tradeoff that exists when designing the free-space representation. A constrained-polygonal representation would not use the profile concept at all and would treat the limiting case, as in Figure 21(c), as a single polygonal area – see Figure 22.

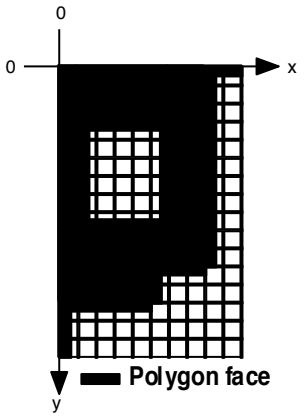

Figure 22.A polygonal representation of free-space. Placing a shape would require testing each polygon vertex and then checking if other vertices interfere with the shape placement.

The advantage of this representation is that any 'holes' (free-space completely surrounded by placed shapes) developed in the layout can be represented as additional constrained-polygonal free-space areas. There is no method to make holes in the single- or dual-representations available for shape placement. It is the job of the GA to fill holes by evolving a suitable permutation which prevented hole creation in the first place.

To conclude this section, the dual-representation has been chosen as it allows a fairly complete representation of free-space, without being overly complex.

5.1.3.2.5 Comparison of algorithms between left- and top-profile functions

Manipulating the top-profile requires the same algorithms as for the left-profile, except all references to a particular axis are switched to the opposite axis. The parallel axis refers to the axis parallel to the direction of an edge in a profile. The orthogonal axis refers to the axis orthogonal to the direction of an edge in a profile. Algorithms may be coded in terms of operations working with parallel and orthogonal axes, allowing them to work with both left- and top-profile, given appropriate initial settings according to the profile being manipulated. The differences are summarised in Table 1.

Feature	Left-Profile	Top-Profile
length of an edge	height of shape	width of shape
edge length measured in	y-axis	x-axis
edges ordered by	y-edge-offset	x-edge-offset
edge hotspot	top-right vertex of shape	bottom-left vertex of shape
parallel axis	y-axis	x-axis
orthogonal axis	x-axis	y-axis

Table 1. Differences between left-profile and top-profile.

The parallel/orthogonal axis terminology is good for coding the algorithms, but is not convenient for describing the algorithms. The code proves that the algorithms for both profiles are the same, except for the initialisation stage where the meanings of parallel and orthogonal axis are set according to the profile being updated. The algorithms are described for the left-profile only, using the appropriate x/y axis terminology, to provide clarity.

- **Terminology (Part 2)**

  For left-profile:

  - an ***edge*** corresponds to the right-face of a shape
  - the ***edge hotspot*** describes the location of the upper-extent of a left-profile edge

  For top-profile:

  - an ***edge*** corresponds to the bottom-face of a shape
  - the ***edge hotspot*** describes the location of the left-extent of a top-profile edge.

  For both profiles:

  - the ***y-edge-offset*** of an edge corresponds to the y component of the edge hotspot
  - the ***x-edge-offset*** of an edge corresponds to the x component of the edge hotspot
  - the ***parallel axis*** is the axis which is parallel to edges in the profile
  - the ***orthogonal axis*** is the axis which is orthogonal to edges in the profile

5.1.3.3 Placement Builder

The placement builder analyses the free-space representation and yields all possible placement positions for a particular shape.

All locations in the left-profile where a single new shape can be placed without overlapping existing shapes in the layout and without overlapping the edges of the sheet are placed into an array of coordinates called *left-places*. Locations found in the top-profile are added to the *top-places* coordinate array. The algorithm below is for extracting locations from the left-profile.

```
FOR all edges (n) in the profile:
        LET current_edge=edge_n
        LET accumulated_length=current_edge.length
        LET fit_location=current_edge.hotspot
        label1:
        IF new_shape.height > accumulated_length THEN
                IF (current_edge+1).hotspot.x > fit_location.x THEN
                        LET fit_location.x = current_edge.hotspot.x
                ENDIF
                SET current edge to next edge in profile
                INCREASE accumulated_length by current_edge.length
                GOTO label1
        ELSE
                IF fit_location.x + new_shape.width < sheet.width
                        Add fit_location to left_places array
        ENDIF
ENDFOR
```

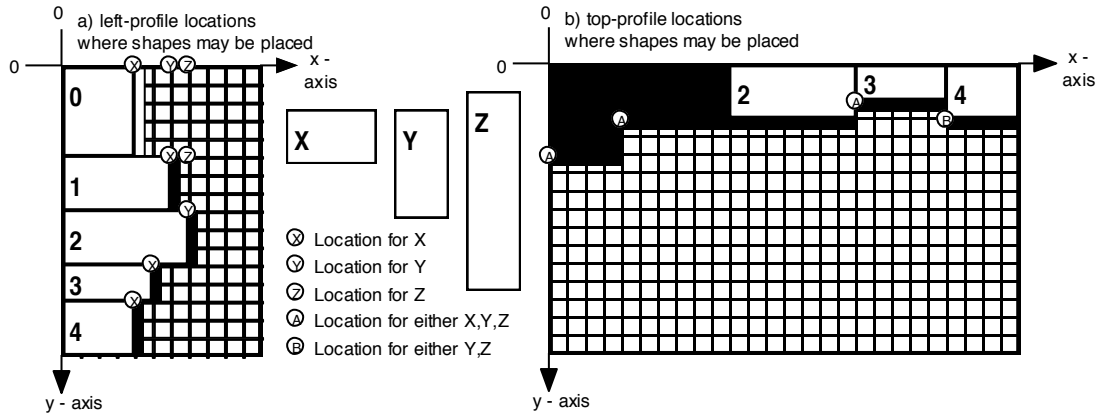

Figure 23. Locations to place shapes X, Y or Z in two layouts, where (a) shows locations derived from a left-profile; and (b) shows locations derived from a top-profile. All locations correspond to the positioning of the top-left vertex of the placed shape.

Figure 23 shows the places found for shapes X,Y and Z by the algorithm above.

5.1.3.4 Placement Chooser

The chooser picks a single placement position, from all the places found by the placement builder, which satisifies the current placement heuristic, say ‘leftmost’.

The locations stored in *left_places* and *top_places* are considered, and a single location which satisfies the current placement heuristic is picked.

If the shape is to be placed leftmost, then all locations in *left_places* and only the first location in *top_places* are considered. The location found with smallest *x-offset* is picked. If there are two or more locations with equally small *x-offset*, the location with smallest *y-offset* is picked. (i)

If the shape is to be placed topmost, then all locations in *top_places* and only the first location in left*_places* are considered. The location found with smallest y-*offset* is picked. If there are two or more locations with equally small y-*offset*, the location with smallest x-*offset* is picked. (ii)

The reason for considering only the first location in *top_places* for (i) is that locations in *top_places* are added to the array (part (3)) in the order in which they have been found in the

top-profile. The top-profile is searched from its head, which is the edge leftmost in the top-profile (this is the edge with smallest *x-offset*).

The reciprocal argument is true for (ii) where only the first element of *left_places* is considered.

Figure 24 shows how a series of shapes has been placed using a left-top placement regime. Shape 0 is placed leftmost, shape 1 – topmost, shape 2 – leftmost, and so on. Shape 8 is the next shape to be placed. Edge profiles are shown, and edge hotspots have been omitted for clarity.

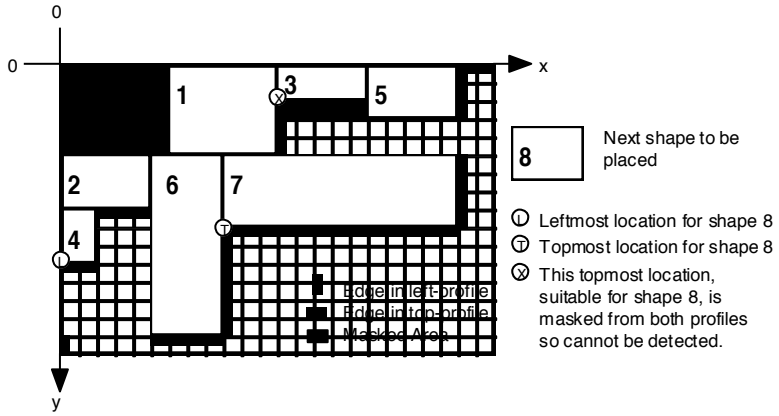

Figure 24. Shapes 0-7 have been placed using a left-top placement regime, beginning with the leftmost heuristic. Shape 8 is the next shape to be placed. Topmost and leftmost locations are labelled. The masked area is unusable as it has a shape to the right and a shape below it, rendering the area 'invisible' to both profiles. Loss of location X is an example of the limitations of the dual-representation.

5.1.3.5 Free-Space Updater

The free-space updater adjusts the free-space representation to take account of a newly placed shape.

Once a place has been chosen for the shape, both left- and top-profiles are updated to account for the revised layout. The algorithm deals with four scenarios:

a) Update left-profile using the right-face of new shape - location chosen from *left_places*
b) Update left-profile using the right-face of new shape - location chosen from *top_places*
c) Update top-profile using the bottom-face of new shape - location chosen from *left_places*
d) Update top-profile using the bottom-face of new shape - location chosen from *top_places*

In cases (a) and (d) the chosen location has its *x-offset* and *y-offset* corresponding to either: the *x-edge-offset* and *y-edge-offset* of a single edge hotspot (as typified by the 'X' shape locations in Figure 23); or the *x-edge-offset* of one edge hotspot and the *y-edge-offset* of a different edge hotspot (as typified by the 'Z' shape locations in Figure 23).

In cases (b) and (c) the chosen location does not correspond to hotspot locations in the corresponding edge profile. This is because the profile from which we have derived the placement position is not the same as the profile we are updating.

Cases (b) and (d) are now examined as an example where both profiles are updated. Figure 25 (a) shows a current layout and the states of both left- and top-profiles. Consider Figure 25(b.i), where shape 3a has been added to the layout using a topmost heuristic. The placement chooser returns the position marked with a circle.

We shall firstly consider case (b) – updating the left-profile. Notice that a part of the right-face of shape 3a is masked behind shape 1 and so does not affect the top edge in the left-profile. Part of 3a juts out below the bottom-extent of this edge and affects the second edge in the left-profile, whilst also creating a new edge. In Figure 25(b.ii), shape 3b has been added to the original layout using a topmost heuristic. This time the new shape is completely masked behind shape 1 and so no changes need to be made to the left-profile.

The top-profile is easier to update (case (d)) because the placement position coincides with the hotspot of the second edge in the top-profile. Updates (b.i) and (b.ii) are equivalent problems for updating the top-profile, where a single edge in the top-profile is bisected.

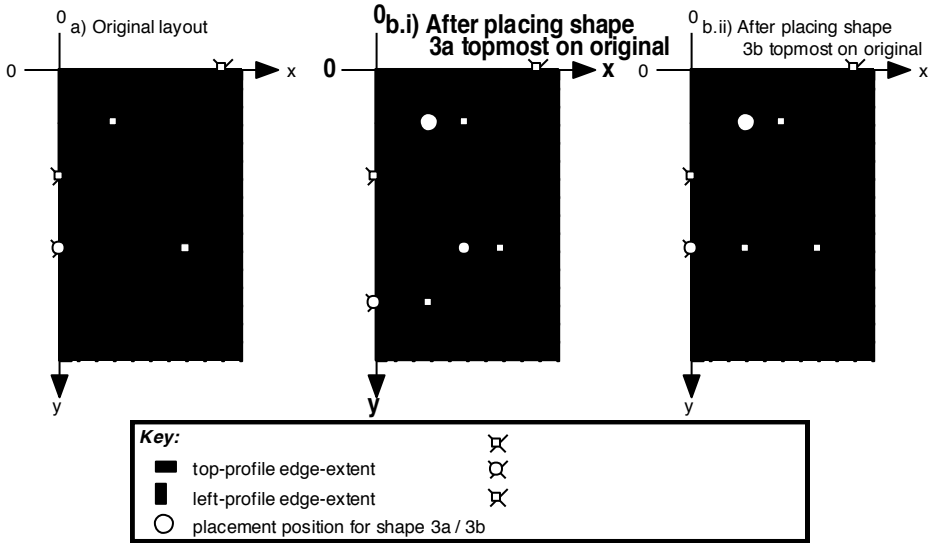

Figure 25. Update scenarios for both left- and top-profiles. (a) A snapshot of the original layout; (b) The layout after adding shape 3a to the original layout; (c) The layout after adding shape 3b to the original layout

These are the types of scenario which the update algorithm has to deal with. Cases (a) through (d) are not explicitly distinguished in the algorithm, but they are all accounted for implicitly. The algorithm was originally written explicitly dealing with the different cases, but the algorithm was found to degenerate into the more elegant solution proposed below.

The algorithm described outlines how the left-profile is updated. The top-profile is updated using the same algorithm, with the necessary changes outlined in Table 1. Below is a list of operations which are carried out by the algorithm. Steps (i-vi) are always executed. Steps (vii-x) are not executed if the new shape is invisible to the left-profile.

(i) Create a new edge to represent the shape to be added to the layout.

(ii) Find out if any edges in the left-profile connect exactly with the new edge.

(iii) Find the first-edge in the left-profile that *might* by affected by new edge inclusion.

(iv) Calculate some values.

(v) Find the start-edge which *actually is* the first edge to be affected and also find the last-edge-affected by the inclusion of the new edge.

(vi) Check if new shape is invisible to left-profile – if so no update needed – exit algorithm.

(vii) Delete all edges from start-edge to *edge before* last-edge-affected.

(viii) Delete last-edge-affected if required.

(ix) Adjust new edge hotspot and length if behind an existing edge.

(x) Integrate new edge into left-profile.

Each of these steps are now detailed.

5.1.3.5.1 Create a new edge to represent the shape to be added to the layout.

The new edge is initialised accordingly:

edge-x-offset = *x* location of right-face of placed shape

edge-y-offset = *y* location of top-face of placed shape

length = height of placed shape

5.1.3.5.2 Find out if any edges in the left-profile connect exactly with the new edge.

Scan the entire edge list to find out if the new edge *abuts* directly above and / or below any existing edges in the list. The first condition for abuttage is for the *edge-x-offset* of the existing edge and the new edge to be equal. Secondly, the existing and new edges must run contiguously in the y-axis.

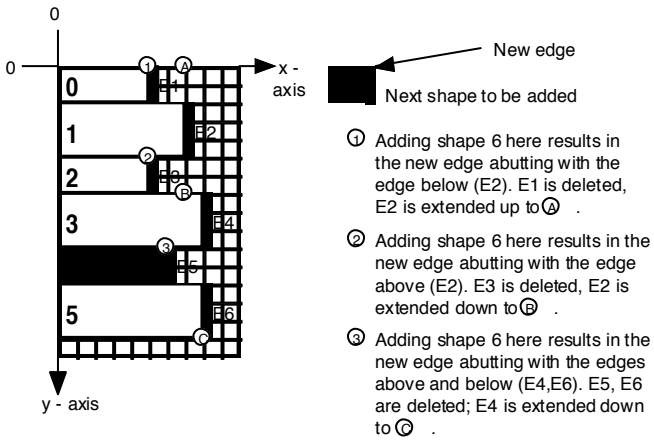

Figure 26. Diagram showing the three types of abutting which can occur when the left-profile is updated.

The ***following_edge*** pointer is set to point to any existing edge which abuts below the new edge. The ***previous-edge*** pointer is set to point to any existing edge which abuts above the new edge. In the case where there are no abutting edges, these pointers are set to NULL.

5.1.3.5.3 Find the first-edge in the left-profile that might by affected by new edge inclusion.

Scan the edge list to find the edge which is horizontally in line with the top face of the new edge. Store a pointer to this edge in both *first-edge* and *start-edge* pointers. Figure 27 gives an example.

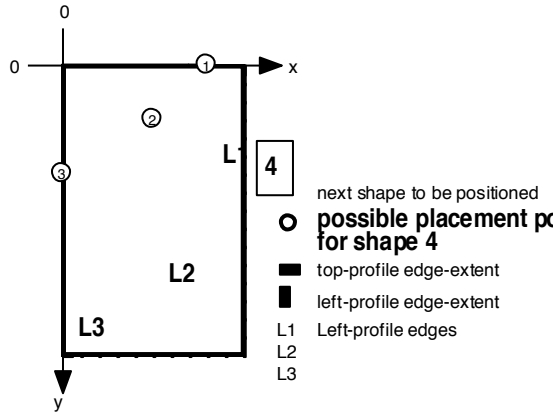

Figure 27. Placing shape 4 at any of the circle locations results in first-edge and start-edge being set to L1.

5.1.3.5.4 Calculate some values.

start-offset stores the difference in *y-edge-offset* between the top of the new shape and the top of the first-edge. This value is 0 for position (1) in Figure 27, 3 for position (2) and 6 for position (3).

accumulated-edge is a distance which is measured starting from the hotspot of first-edge in an increasing y-axis direction. It is used when progressing through the edge profile to find out which edges are affected by the new shape being added and in particular which edge is inline with the bottom-face of the new shape (the *last-edge-affected*).

The *accumulated-edge* value is initially set to such a value that results in *accumulated-edge* equalling the distance between the top-face of the new shape and the bottom-extent of the first-edge (measured in the y-axis direction), *after* the length of the first-edge has been added

to the initial *accumulated-edge* value. Thus, *accumulated-edge* is initially set to negative *start-offset*.

5.1.3.5.5 Find the start-edge which actually is the first edge to be affected and also find the last-edge-affected by the inclusion of the new edge.

Setting start-edge to an edge other than first-edge is necessary when the new shape juts out from behind edges in the profile, as shape 3a does in Figure 25(b.i). In the original layout of Figure 25(a) when considering the placement of shape 3a, first-edge is set to the top edge in the left-profile, whilst start-edge is the second edge in the left-profile. The last-edge-affected is also the second edge in the left-profile in Figure 25(a). The algorithm for setting start-edge and last-edge-affected follows:

Initially, SET *n* to first-edge.
label_1:
Add the length of the *nth* edge to accumulated-edge and store the result in accumulated-edge. IF new edge has an *x-edge-offset* less than edge *n* (ie new edge is behind edge *n*), set start-edge to point to edge *n+1* (the next edge after *n* in the edge list).
IF accumulated-edge < height of the new shape
    GOTO *label_1* after setting *n* to *n+1*
ELSE
    SET the ***last-edge-affected*** pointer to edge *n*.

5.1.3.5.6 Check if new shape is invisible to left-profile – if so no update needed and exit the algorithm.

If the hotspot of new edge is behind the hotspot of the last-edge-affected, AND if the bottom edge of the new shape is above the lowest reach of the last-edge-affected, then the new shape is 'invisible' to the left profile – so no changes to the left profile are required and we can ***exit the algorithm***. See Figure 25(b.ii) for an example where the new shape is invisible to the left-profile.

5.1.3.5.7 Delete all edges from start-edge to *edge before* last-edge-affected.

Delete all edges beginning with start-edge, up to, but not including, last-edge-affected. This removes any edges bypassed by the new edge. Also set ***remaining*** value to accumulated-edge minus new shape height. (*remaining* is equal to the length of the last-edge-affected that is not

overlapped by the new edge. See Figure 25(b.i) where the length of the bottom edge in the left-profile corresponds to this *remaining* value.)

5.1.3.5.8 Delete last-edge-affected if required.

IF the *remaining* value is zero, delete the last-edge-affected from the edge list, as the last edge is completely overlapped by the new edge.
ELSE move the hotspot of the last-edge-affected such that last-edge-affected abuts below the new edge and reduce the length of last_edge_affected to equal *remaining* value (See Figure 25(b.i) where the bottom edge in the left-profile is created in this way.)

5.1.3.5.9 Adjust new edge hotspot and length if behind an existing edge.

If the new edge hotspot is behind the first edge hotspot then move the new edge hotspot down, such that the new edge is inline with the top-extent of start-edge. Next, reduce the new edge length by the amount the hotspot moved. See Figure 25(b.i) where the new edge in the left-profile, due to shape 3a, is not as long as shape 3a is high. Also, the *edge-y-offset* of the new edge has been shifted to be in line with the top-extent of the start-edge.

5.1.3.5.10 Integrate new edge into left-profile.

(See Figure 26 to see how the abutting process merges existing edges with the new edge.)
IF there is no abutting to do above or below the new edge then add the new edge to the edge list keeping the edge list ordered on increasing *y-edge-offset*.

OTHERWISE:
If abutting occurs only with following-edge (ie below new edge), set the following-edge hotspot to coincide with the new edge hotspot, then add the length of the new edge to the following- edge length, storing the result.

OTHERWISE:
If abutting occurs only with previous-edge (ie above new edge), increase previous-edge length by new edge length.

OTHERWISE:
If abutting occurs both above and below the new edge, increase previous-edge length by the

sum of new edge length and following-edge length. Remove the following-edge from the edge list.

This completes the design and implementation of §5.1.3.5, the *Free-Space Updater* part of the LDA features. The remaining LDA features are discussed below.

5.1.3.6 Layout Updater

The layout updater adds the newly placed shape, and its location on the stock-sheet, to the layout. Once a location has been chosen for the new shape, the shape along with its location and a flag indicating whether a new sheet is required is encapsulated inside a layout_piece and is added to the layout_piece array. The layout_piece array is used to recover coordinates for all shapes after the entire layout has been arranged.

5.1.3.7 Sheet Creator

The sheet creator processes the situation where the current sheet has insufficient space to place the next shape (where the placement builder yields no placement positions).When no placement positions are available for the new shape, a new, empty sheet is used to accommodate this shape and any remaining unplaced shapes. If this new sheet itself becomes full, then another sheet is created. This is a recursive process which generates as many sheets as are required to lay out the permutation described in the current chromosome. All sheets are made to the same (x-size,y-size) dimensions.

5.1.3.8 Fitness Evaluator

The evaluator returns a fitness value for the current layout corresponding to how efficiently sheets are used. This is calculated by considering the final state of the free-space representation after all shape_genes in the LDA_chromosome have been placed.

The fitness evaluator has to provide a simple metric by which we can evaluate the quality of a layout, and thus the quality of a chromosome. A floating point number in the range 0 to 1 is ideal. There is a two stage process to generate this fitness metric.

Firstly, all sheets required to lay out the permutation, described in the chromosome being evaluated, are analysed. After a sheet becomes full and once the entire layout has been placed, the area used on the current sheet is evaluated and stored in a fitness-report. Once all

shapes have been laid out, the fitness-report contains separate area data for each sheet used in the layout.

The area used in each sheet is calculated by dropping a perpendicular line coinciding with the rightmost edge in the sheet's left-profile, leaving a rectangular free-space area to the right. This free-space is considered 'good-scrap' as it may be suitable for reuse. The rectangular area to the left of the perpendicular line is deemed to be the sheet-usage area. Figure 28 illustrates an example of what is included in the sheet-usage area. This may be refined by considering both the free-area to the right of the shapes and the free-area below the shapes on the stock-sheet. In this case there is the possibility for two pieces of good-scrap.

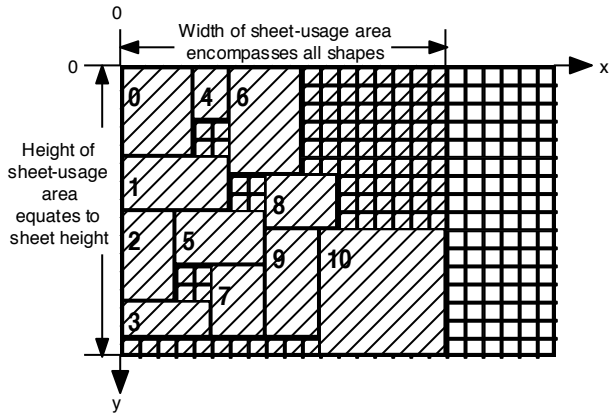

Figure 28. The hatched area represents sheet-usage area for this sheet.

Once all sheet-usage areas are available, an overall-usage function considers:

- the total sheet-usage area for all sheets (*total-used-area*)
- the total area of all shapes in the layout (*min-achievable-area*)
- the number of sheets used by the layout (*actual-sheet-number)*
- the minimum number of sheets that *theoretically* could contain all shapes. (*min-sheet-number*)

The result of the *overall-usage* function is a floating-point number in the range 0 to 1, where 1 represents perfect fitness. The result is used as the fitness value for the entire layout. The GA uses this fitness value when selecting parent chromosomes for breeding.

5.2 LDA_environment class (LDA_e base)

5.2.1 Introduction

The LDA_e class implements a simple level-oriented layout determining algorithm (LDA), as described in §5.1.1. The class is used as a base for the LDA_new class which implements the LDA in §5.1.3.

The environment is responsible for evaluating the layout resulting from considering all genes in a chromosome and for providing a fitness measure for the layout. Environment settings include the size of the stock-sheet to be used. The environment also contains a pointer to a shape group holding all shapes that are to be laid out.

5.2.2 Supplying shapes to the environment

The provide_shapes function is used to supply the environment with a shape group.

5.2.3 Layout Tracking

When evaluating a chromosome, two types of layout data are maintained. The LDA data is used to decide where to place the next shape.The Finalised Layout data is an array of all shapes placed so far, with each shape having a coordinate locating their top-left vertex.

5.2.3.1 Layout Determining Algorithm Data

In the level-oriented environment, a pointer (max_width_index) refers to the widest shape in the column of shapes currently being placed. Max_width_index references the appropriate shape in the Finalised Layout Data. This data is used by the add_to_layout function to calculate if the next shape is small enough to fit in the current column. If there is insufficient space, a new column is started and the shape is placed there, otherwise the shape is added to the top of the current column.

5.2.3.2 Finalised Layout Data

The finalised layout consists of an array of *layout_piece* objects. Each layout_piece object contains:

- a shape_g_l object called piece
- a sheetflag_t object called new_sheet_needed

The piece contains shape size data and a coordinate locating the top-left vertex of the shape. New_sheet_needed is one of:

s_YES: Start using a new stock-sheet from this shape onwards
s_NO: Continue using the current stock-sheet

The first layout_piece has its New_sheet_needed flag set to s_YES. Subsequent layout pieces have their flag set to s_NO. Once a stock-sheet becomes full, only the first shape to be placed on the new sheet has its flag set to s_YES.

5.2.4 The Fitness Function

This function is passed a chromosome for evaluating, which is passed to the *fitness_r* function. The fitness_r function evaluates the entire chromosome, recursively creating new LDA_e objects to layout across multiple sheets if the current stock-sheet becomes full. The fitness_r function calls the *add_to_layout* function to process each shape_gene in the chromosome.

Once a stock-sheet becomes full, or once all shapes have been laid out, the *sheet_usage* function is called to evaluate the dimensions of the area used by the layout on the current stock-sheet. The result of this is stored in a *fitness_report* object.

A fitness_report object is an array of *area_t* objects, each containing a width and height measure. After all shapes have been evaluated, the fitness_report object contains a count of the number of sheets used, and the dimensions of the area used on each sheet.

After the fitness_r function has finished placing all shapes, the *overall_usage* function is called by the fitness function. This function evaluates an overall fitness score for the chromosome, in the range 0 to 1, using the data stored in the fitness_report object. The dimensions and size of the used area on all sheets is taken into consideration. Total areas are compared with the total area of all the shapes that were laid out.

5.2.5 Chromosome Services

As the environment has a pointer to the shape group holding all shapes to be laid out, the environment can initialise chromosomes to hold meaningful data.

5.2.5.1 Init_chromosome Function

This function initialises the genes of a chromosome with shapes such that the gene-ids progress from 0 to l-1, where l is the length of the chromosome. In other words as the genes are scanned from the gene with index 0, shapes are ordered identically to the shape ordering in the shape group.

5.2.5.2 Random_chromosome Function

This function stores a random permutation of genes in the chromosome. This results in a random ordering of the shapes. If orientation or heuristic genetic operators are being used, then the respective features of each shape_gene are randomised as well.

5.2.5.3 Ordered_chromosome Function

This function requires an array of integers which describes the permutation of shapes that is to be stored in the chromosome. Shapes are referred to by their id's. An array which holds the integers 0 to *l-1* in increasing order, where *l* is the length of the chromosome, would yield identical results to an init_chromosome call.

5.2.5.4 Get_permutation Function

This function requires a pointer to an array of integers and stores in this array the id's of all the shapes stored in a chromosome, in order.

5.2.5.5 Get_permutation_char Function

This function is identical to Get_permutation, except the output is a character array – integers are stored as ASCII characters separated by white space.

5.3 LDA_new class (LDA_e base class)

5.3.1 Introduction

This class inherits from the LDA_e class and so shares a large amount of code with this class. The class implements the LDA in §5.1.3 which is the major contribution of this project. The LDA allows shapes to be placed using a variety of fixed and dynamic heuristics across multiple stock-sheets. Fixed heuristics require that one heuristic is used to place all shapes. Dynamic heuristics involve the LDA looking up the heuristic setting for every shape_gene that is processed.

The main functions that LDA_new class overrides are:

- add_to_layout
- clear_layout
- sheet_usage
- overall_usage

5.3.2 Layout Tracking

The Finalised Layout data is stored in the same way as for LDA_e class. The Layout Determining Algorithm Data stores information pertaining to the left and top profiles and also lists of coordinates which hold possible locations for the current shape.

5.3.2.1 Layout Determining Algorithm Data

The left- and top-profiles for the current sheet are implemented as linked lists of *edge_node* objects. An edge_node object contains:

1. A hotspot location *coord* object – marks the coordinate of the start of the edge
2. A length integer
3. An edge_node pointer

Two coord_group objects (*left_places* and *top_places*) contain an array of coord objects which are used to hold the collection of locations where the current shape can be placed.

5.3.3 The Fitness Function

The fitness function and the fitness_r function are identical to that of LDA_e class. However, the add_to_layout, sheet_usage and overall_usage functions, which are called by the fitness functions, are overriden.

5.3.3.1 Add_to_layout Function

This function adds individual shapes to the layout, updating both the profiles and the finalised layout data. Shapes can be placed using one of four heuristics:

- *leftmost* – The shape is placed as far left as possible on the stock-sheet.
- *inner-leftmost* – This is the same as *leftmost,* except locations on the extreme left of the sheet are avoided if possible.
- *topmost* – The shape is placed as far top as possible on the stock-sheet.
- *inner-topmost* – This is the same as *topmost*, except locations on the extreme top of the sheet are avoided if possible.

The inner-leftmost and inner-topmost heuristics bias layouts away from being built-up, initially, along the edges of the stock-sheet, whilst usage of the sheet interior is favoured.

The algorithm follows: (*item* is the shape_gene to be processed)

1.
```
process_current_rule(item);
```
Sets the current_rule to be the heuristic setting in the item if the shapes are to be placed dynamically.

2.
```
status left_build_status=build_left_profile_places(item);
status top_build_status=build_top_profile_places(item);
```
Examines both profiles for all locations where the item can be placed, storing the results in left_places and top_places coordinate arrays. The status returned is false if no locations have been found in the respective profile.

3.
```
if (left_build_status==FAILURE && top_build_status==FAILURE)
   return FAILURE;
```
Return a failure if there are no locations where the item will fit on the sheet

4.
```
coord chosen_coord;
switch (current_rule)
{
    case LEFTMOST:
            chosen_coord=choose_leftmost_place(left_build_status, top_build_status);
            break;
    case ILEFTMOST:
            chosen_coord=choose_ileftmost_place(left_build_status, top_build_status);
            break;
    case TOPMOST:
            chosen_coord=choose_topmost_place(top_build_status, left_build_status);
            break;
    case ITOPMOST:
            chosen_coord=choose_itopmost_place(top_build_status, left_build_status);
            break;
    default:
            break;
}
shape_g_l chosen_gene(item, chosen_coord);
```
The current_rule stores the heuristic which decides which of the locations to choose to place the item. The chosen_gene variable stores both the shape dimensions and the shape's location on the stock-sheet.

5.
```
layout[next_index].piece=chosen_gene;
next_index++;
```
The layout array stores the shape and its location on the stock-sheet.

```
6.  add_shape_at_place_6(chosen_gene, MODIFY_LEFT_PROFILE);
    add_shape_at_place_6(chosen_gene, MODIFY_TOP_PROFILE);
```

Calling this function updates both left- and top-profiles to take into account the new shape on the stock-sheet.

5.3.3.2 Sheet_usage Function

This function returns an area_t object, which consists of width and height integer data. The width of the layout is determined by finding the right-most edge in the left-profile. The height of the layout is determined by finding the bottom-most edge in the top-profile.

5.3.3.3 Overall_usage Function

This function calculates the fitness of an entire multi-sheet layout:

sheets_used = number of sheets required for the layout
last_sheet_usage = sheet_usage for the stock-sheet holding the last placed shape
total_usage = sum of sheet_usage areas for all stock-sheets used by the layout
total_area = sum of individual shape areas
sheet_area = sheet_width * sheet_height

$$area\_quotient = \frac{total\_area}{total\_usage}$$

$$minimum\_sheets = \frac{total\_area}{sheet\_area}$$

$$actual\_sheets = sheets\_used - 1 + \frac{last\_sheet\_usage}{sheet\_area}$$

$$penalty = \frac{1}{actual\_sheets - minimum\_sheets + 1}$$

$$overall\_fitness = area\_quotient * penalty$$

The main influence on overall fitness is the *area_quotient*. If the packing of the layout is very efficient (ie little wasted space), then this value will be close to one. The *penalty* function penalises the use of more sheets than is the absolute theoretical minimum. This is necessary as otherwise a layout which uses one stock-sheet per shape would have a fitness of one. This is because the sheet_usage area consists of the bounding box containing all the shapes placed on a single stock-sheet.

6 Population-Related Design

This section covers the population representation and the selection operators that pick chromosomes from the population.

6.1 Population Class

The population class is a collection class for chromosomes. When a population object is created, it is supplied with two parameters:

1. an example of the type of chromosome required in the population
2. the size of the population

The *create_array* function available in the chromosome class is used by the population class to create an array of chromosomes. The following services are provided by the population class:

- *get_candidate* – returns a reference to an integer-indexed chromosome in the population. Indexes range from 0 to *p*-1, where *p* is the population size
- *new_candidate* – uses the chromosome clone function to return a newly allocated chromosome that is compatible with the other chromosomes in the population
- *replace_candidate* – this is provided with an index location which locates a chromosome in the population to be overwritten – a chromosome pointer is also provided, which points to the chromosome that is to be copied into the population
- *randomize population* – calls the random_chromosome function for each chromosome in the population
- *standardize population* – calls the init_chromosome function for each chromosome in the population

6.2 Selector Classes

Selector classes are used to pick chromosomes from a population for breeding and for removal.

6.2.1 Selector Base Class

This class forms the base for all selection classes. When a selector is created, it must be supplied with a pointer to a population to select from. All classes derived from the selector class have to provide the following functions:

- *select_index* – this returns an integer indexing a single chromosome in the population – the result of the selection procedure.
- *select_candidate* – this returns a pointer to the selected chromosome.
- *clone* – this returns a pointer to new selector-type object, with duplicate data as that of the selector object used to call this function.

The following function is provided by the selector class and is not overridden by derived selector classes:

- *make_child* – this returns a pointer to a new chromosome, which is of a type compatible with the population being selected from.

6.2.2 Rank Selection – rank_s class

All candidates in the population have their fitness evaluated, and the population is ordered according to decreasing fitness. The first member of the ordered population (the fittest candidate) has *n*-times the probability of being selected than the least fit candidate, where *n* is the number of candidates in the population. The second-most fit candidate has (*n-1*)-times the probability of being selected than the least fit candidate, and so on for the rest of the population. The probability of the least fit candidate being selected is $1 / (1 + 2 + ... + (n-1) + n) = 1 / (n * (1+n) / 2)$.

6.2.3 Tournament Selection – tournament_s class

A tournament may be held amongst *m* randomly picked candidates in the population. The tournament winner is the fittest candidate amongst those picked. *m* defaults to two, but may be set to any integer values greater than or equal to two when the tournament_s object is created.

6.2.4 Best Selection – best_s class

This class always selects the best chromosome in the population.

6.2.5 Worst Selection – worst_s class

This class always selects the worst chromosome in the population.

6.2.6 Worst By Rank (Inverted Rank) Selection– worst_rank_s class

This class uses inherits from rank_s class, but inverts the selection function so that the worst chromosome is most likely to be picked.

7 System-Related Design

This section discusses the system components which drive the evolution process. In §7.1 methods for selecting which genetic operators to use are discussed. In §7.2, generational models are described. These provide the control to evolve a population of chromosomes one generation. In §7.3, the evolution driver is described. This component pulls together the whole system.

7.1 Operator Rate Adaptors

Operator rate adaptors determine which genetic operator to apply next to the population.

7.1.1 Adaptor Base Class

The base class has the following data members:

1. A 100 element array of crossover-type pointers
2. A 100 element array of mutation-type pointers
3. A pointer to a selection method

When creating an adaptor, a selection operator must first be created and passed to the adaptor. The selection operator provides a link to the population to evolve and decides which chromosomes to breed. The crossover and mutation operators that have been selected for use are then supplied to the adaptor, using:

- *add_crossor* – an example of the mutation operator required is passed using this function
- *add_mutator* – an example of the crossover operator required is passed using this function

These two functions clone the operator examples which they are given and store these new operators in the crossover and mutation operator arrays of the operator rate adaptor.

The *apply_operator* function has the job of applying a genetic operator to the population and obtaining a resulting child. The apply_operator function is a pure virtual function, and so must be overridden in a derivative class. The apply_operator function is used to :

1. pick a single operator from one of the operator arrays
2. select one parent in the case of mutation, or two parents in the case of crossover
3. create a child chromosome which will store the result of the genetic operator
4. pass the parent(s) and child to the chosen genetic operator
5. return a pointer to the child

Note that the operator adaptor assigns memory for the children which are produced. The adaptor is not responsible for putting the child into the population; this is the job of the Generational Model (7.2).

7.1.2 COst Based Operator Rate Adaptor – COBRA_a class

All classes derived from the adaptor class have to override the apply_operator function. The COBRA rate adaptor does the following:

- For the first *i* calls to the apply_operator function, the next genetic operators to use is chosen at random. A score is kept of how good each genetic operator is at generating children which are better than their parents.
- Once *i* calls to the apply_operator function have been made, the overall scores for all the genetic operators are used to generate a list which orders operators on decreasing score.
- For all subsequent calls to the apply_operator function, instead of operators being chosen at random, they are chosen probabilistically depending on their place in the score list. A rank-based selection is used which is identical to that used in the rank chromosome selector, rank_s class. The chance of selecting the least fit operator is

  $1 / (1 + 2 + ... + (n-1) + n) = 1 / (n * (1+n) / 2)$, where n is the number of operators

  and the chance of selecting the fittest operator is n times this value.

The COBRA rate adaptor has three parameters:

1. *iteration_point* – determines the number of calls that are to be made to the apply_operator function before the genetic operators are assessed for overall score. The function, *set_iteration_point* sets this value.

2. *mutation_points* – the number of points to be added to a mutation operator's score when a child is generated which is fitter than the parent. By default this is set to the value, 2.
3. *crossover_points* – the number of points to be added to a crossover operator's score when a child is generated which is fitter than one parent. The score is doubled if the child is fitter than both parents. By default this is set to value, 1.

The mutation_points value is twice the crossover_points value so that crossover operators aren't unfairly advantaged. When a crossover operator generates a child, the child is compared in fitness to both parents. A mutation operator can have its child compared only to the one parent.

7.1.3 Random Rate Adaptor – random_a class

This operator adaptor makes a random choice made by considering all crossover and all mutation operators available. The chosen operator is then used to generate a child, having been supplied with selected parent(s).

7.1.4 Fixed Rate Adaptor – fixed_a class

This operator adaptor chooses whether to apply a crossover operator or a mutation operator probabilistically. The probability of using a crossover operator is set using the *set_rate* function. After deciding which type of operator to apply, a random choice is made from all the operators of that type. The chosen operator is then used to generate a child, having been supplied with selected parent(s).

7.1.5 Adaptive Mutation – adaptive_mutation_a class

This operator adaptor uses the Hamming distance between two chromosomes to determine whether to apply a crossover or a mutation operator. The probability of using a crossover operator is determine by:

$$\frac{\text{distance}(parent1, parent2)}{length\_of\_chromosome}$$

The distance operator compares each gene in parent1 against the respective gene in parent2. If the gene-ids for a single comparison do not match, the distance measure is increased by one. Comparisons are made for every gene in the chromosome.

After deciding which type of operator to apply, a random choice is made from all the operators of that type. The chosen operator is then used to generate a child, having been supplied with selected parent(s).

7.2 Generational Models

The generational model has the task of controlling the creation of a new generation of chromosomes from a population. An operator rate adaptor is required to generate child chromosomes. A strategy for integrating the child (or children) back into the population is also required.

7.2.1 Generational_Model Base Class

The base class has the following data members:

- a pointer to the population to be evolved
- a pointer to an operator rate adaptor
- a pointer to a selection operator
- a *finished* boolean – used in some stochastic models

The base class requires the *evolve_once* function to be overriden. This function uses the member data objects to complete one generation of evolution.

7.2.2 Steady-State Model – steady_state class

The steady-state model uses the operator adaptor to create one child chromosome from the population. In the steady-state model the population size stays constant. To deal with this, a selection operator is used to pick a chromosome for removal. A usual selector operator to use is the worst_s selector which picks the least-fit chromosome. The *replace_candidate* function, in the population class, is used to replace the chosen chromosome with the child chromosome. This involves copying the gene information from the child to the picked chromosome in the population. After copying the child into the population, the child chromosome is no longer required and is deleted.

7.2.3 Stochastic Model – stochastic class

The stochastic model is used for Hill Climbing (HC), Steepest Ascent Hill Climbing (SAHC) and Simulated Annealing (SA). The model requires a population which has a size of one.

Each of HC, SAHC and SA has a special mutation operator which is used in conjunction with the stochastic model.

When a stochastic object is created, it requires pointers to a special mutation operator, a population and an operator rate adaptor. The special mutation operator determines whether HC, SAHC or SA is to be performed.

The stochastic model keeps a record of the fittest chromosome formed since the model was created. Also, after each call to evolve_once, the model checks to see if the special mutation operator has terminated. If this is so, the *finished* boolean (defined in the base class) is set to true. This is required for SA as the process finishes once the minimum temperature has been reached.

The special mutators require an operator rate adaptor in order to apply a normal mutation operator to the single chromosome in the population. The literature review contains additional information concerning HC, SAHC and SA.

7.2.4 Special Mutator – Random

This mutation operator simply randomizes the parent chromosome.

7.2.5 Stochastic – Hill Climbing

Hill Climbing has one parameter – the restart threshold. The operator adaptor is used to apply a mutation operator (say swap mutation) to the current chromosome. If the child chromosome is fitter than the original, then the original is replaced by the new chromosome. Otherwise, the original solution is kept and the operator adaptor is applied again. If the operator adaptor generates the threshold number of children without improving on the current solution, then the hill climbing algorithm is restarted. This involves randomizing the current solution.

7.2.6 Stochastic – Steepest Ascent Hill Climbing

The sample rate parameter determines how many samples to take before hill climbing. In standard hill climbing a single sample is taken. In SAHC the best chromosome out of the sample obtained is selected.

7.2.7 Stochastic – Simulated Annealing

This mutator has the following parameters:

- *start_temp* – Temperature with which to start the annealing process
- *end_temp* – Temperature which signals the end of the annealing process
- *cooling_ratio* – Rate at which temperature is decreased – must be less than 1.0 and greater than 0.0.

The *reset_mutator* function is called before annealing begins, in order to reset the current temperature to start_temp.

Each time the mutate function is called, one step in the simulated annealing process is made and the cooling schedule is updated (start_temp is multiplied by the cooling_ratio). If start_temp becomes less than or equal to end_temp, the *finished* flag is set to true.

7.3 Evolution Driver

7.3.1 Introduction

The evolution driver uses the settings in the GA Settings Dialog Box (see the User Manual) to prepare and run the GA experiment. §7.3.2 summarises the algorithm.

7.3.2 Evolution Driver Algorithm

Get the frequency at which reports are to be made, from GA Settings Dialog Box

IF generational model is one of : HC, SAHC, SA or Random
    Set population size to one
ELSE
    Get population size
ENDIF

Create a population of the appropriate size

IF generational model is Steady-State
    Create a breed-selector of type according to GA Settings Dialog Box setting
ELSE
    Create a breed-selector of select best type
ENDIF

Create a removal-selector based on GA Settings Dialog Box settings

Create a best-selector and a worst-selector.

Create an operator rate adaptor according to GA Settings Dialog Box settings
Pass the breed-selector to the operator rate adaptor

IF generational model is Steady State
    FOR all **crossover operators** that have been selected in the GA Settings Dialog Box
        add an example of each crossover operator to the operator rate adaptor
    ENDFOR
ENDIF

FOR all **mutation operators** that have been selected in the GA Settings Dialog Box
    add an example of each mutation operator to the operator rate adaptor

ENDFOR

Randomize the population:

1. Randomize the permutation of genes in all chromosomes.
2. If crossover/mutation operators that manipulate orientation flags have been selected for use, then randomize orientation flags in all genes for all chromosomes.
3. If crossover/mutation operators that manipulate heuristic settings have been selected for use, then randomize heuristic settings in all genes for all chromosomes.

Create a Generational Model according to GA Settings Dialog Box settings

Create a Cataclysmic Mutator:

1. If crossover/mutation operators that manipulate orientation flags have been selected for use, then set the randomizer to randomize_feature_1. Set mutate_features to true.
2. If crossover/mutation operators that manipulate heuristic settings have been selected for use, then set the randomizer to randomize_feature_2. Set mutate_features to true.
3. If both (1) and (2) are required then set the randomizer to randomize_all_features
4. If none of (1) to (3) are applicable, set mutate_features to false.

INITIALISE *iteration_count* to zero, and *duplicate_count* to zero

DO

Set the *finished* parameter of the generational model to false.

Call the evolve_once function of the generational model.

If the *finished* parameter is now true

BREAK out of the DO loop

IF a report is to made on this iteration

Get the best and worst chromosomes in the population and evaluate fitness

Set *duplicate_count* to no. of chromosomes similar in fitness to best

Output report

IF cataclysmic mutation is active (set in GA Settings Dialog Box)

IF no. of duplicates exceeded threshold (in GA Settings Dialog Box)

Cataclysmically mutate the population

IF best fitness is within 0.01 of max fitness (1)

BREAK out of the DO loop

WHILE (Number of generations not reached maximum)

8 User Manual for StockGA Application

8.1 Introduction

This section explains the user operation of the StockGA application. First, the input file format is described. The input file holds shape information for all layouts, or tests, which are to be loaded in to the application at start-up. In the following section, an application snapshot and its visible features are presented. Next, the application menu items are covered.

8.2 Application Overview

StockGA processes layouts which are contained in the 'shapes.txt' input file. Each layout, or test, is an independent entity. When a test is initially displayed on the screen, the shapes defined in the test are laid out in the order found in the input file. If the layout requires more than one stock-sheet, the user can navigate through all sheets using the menu commands. Likewise, all layouts in the input file can be navigated using the menu commands.

By using the GA Settings dialog box, the user sets all parameters for running a GA experiment. To run the experiment, the 'Run GA' menu command is used. Whilst the GA experiment is running, status information including: number of generations processed and best and worst fitnesses in the population, are displayed in the output window. Once the experiment has finished, the evolved layout is displayed in the application window.

The layout currently on display in the application window can be exported to a text file (called layout.txt). All shape and location data, the total area used on each sheet and an overall fitness score are stored in this file.

A facility for batch running multiple GA experiments over multiple test-layouts, and over a specified number of repetitions (to obtain results for statistical analysis) is also provided. At any time during the batch-process, and after completion, results may be copied out of the output window for placing in, say, a spreadsheet. An option is provided to limit output to just the best fitness obtained in each experiment repetition.

8.3 Test Data File Format

Test data must be stored in a text file called ‘shapes.txt’. There is no limit to the number of tests that this file may contain. Each test is separated by white space, and the format to be used for each test is as follows:

<test-id> <shape-list> <test-description> <!> <regime> <sheet-width> <sheet-height>

All fields <> are delimited by white space. Table 2 describes the contents of each field.

<test-id>	A single string delimited by white-space which may contain up to 9 characters.
<shape list>	This field contains a list of shape specifications. Each specification contains an integer width, an integer height and a trailing comma (,). All specifications are separated by white-space. The last specification has a trailing colon (:). eg 2 2, 3 3, 4 4:
<test-description>	This field may contain up to 999 characters, including white-space. The exclamation (!) character must not be used in this field.
<!>	This field consists of a single ‘!’ character and must have white-space either side of it.
<regime>	This field must contain exactly one of the following (capitalised) labels: { LEFTMOST ILEFTMOST TOPMOST ITOPMOST DYNAMIC } This field sets the regime (or heuristic) used by the layout determining algorithm for placing shapes.
<sheet-width>	A positive integer which sets the width of a single stock-sheet.
<sheet-height>	A positive integer which sets the height of a single stock-sheet.

Table 2. Fields making up a single test in the input test file, ‘shapes.txt’. All shape and sheet sizes must be integer values.

An example consisting of three test entries:

1.1L) 20 10, 20 15: Force abutting in y-axis with previous ! LEFTMOST 30 30
1.2L) 20 10, 15 10, 20 10, 5 10: Force abutting in y-axis with previous and following ! LEFTMOST 30 30
1.3L) 20 10, 15 10, 20 10, 4 5, 5 5: Force abutting in y-axis with following ! LEFTMOST 30 30

8.4 Application Snapshots

The main application window is illustrated in Figure 29.

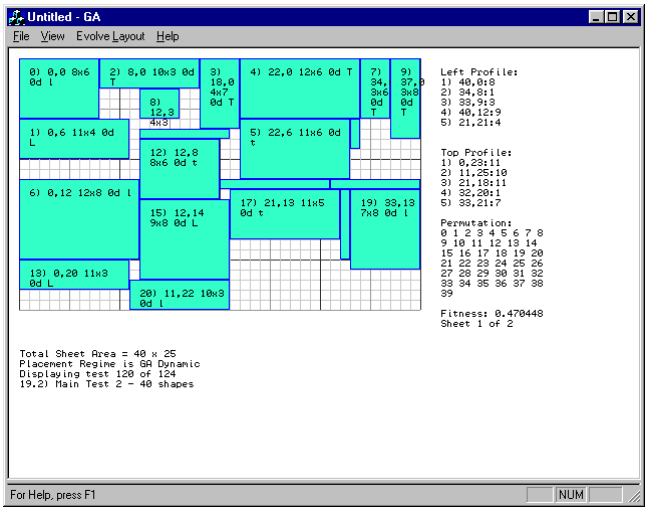

Figure 29. The StockGA Main Application Window.

The menu bar contains commands for manipulating the current layout, for running GA experiments, for navigating through sheets used in this layout, and for navigating other tests present in the test file.

The stock-cutting outline is placed on top of a grid which represents a single stock-sheet. The grid lines are drawn at unit intervals. Each placed shape has an annotation (which may not be visible if the shape is too small) which contains the following data: shape-id, coordinate of top-left vertex, dimensions of the shape, whether the shape is rotated 0 or 90 degrees, and a single character indicating the heuristic used to place the shape. For example, the top-left shape in the layout of Figure 29 has the following annotation:

0) 0,0 8x6 0d l

In this example, the shape-id is zero, the coordinate of the top-left vertex is (0,0), the size of the shape is (8x6), the shape has 0 degrees rotation and the heuristic ‘l’ has been used to place the shape. Shape-ids are assigned to shapes in the order in which they are found in the test specification.

Heuristic Label	Full-name of Heuristic
L	LEFTMOST
l	ILEFTMOST (Inner leftmost)
T	TOPMOST
t	ITOPMOST (Inner Topmost

Table 3. Heuristic labels and their corresponding full names.

Edges specifications for edges in the left and top edge profiles are displayed in the order found in their respective list. An edge specification consists of an (x,y) coordinate (called the hotspot), followed by the length of the edge. For left profile edges, the hotspot is the top-most extent of the edge. For top profile edges, the hotspot is the left-most extent of the edge.

Other information displayed includes: the permutation of shapes that has been laid out; the overall fitness of the layout; the number of sheets the layout uses and the current sheet being displayed; the total sheet area; the placement regime used to place shapes; and test-id and test description information.

The output window (see Figure 30) displays status data pertaining to the latest GA experiment. This display is updated whilst the GA experiment is running. The fields from left-to-right are: number of generations evolved, fitness of best chromosome, fitness of worst chromosome and the number of 'duplicate' chromosomes present in the population. A 'duplicate' chromosome has a 'similar' fitness to the fittest chromosome. The 'CM' annotation denotes that the number of duplicates has exceeded a 'threshold' and that the population has been cataclysmically mutated. The definitions of 'similar' and 'threshold' are set in the GA Settings Dialog.

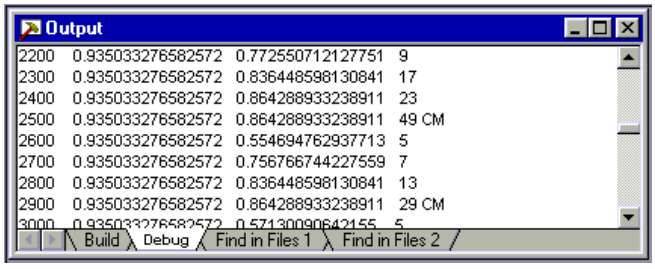

Figure 30. The StockGA Output Window.

8.5 Application Menu Commands

Menu commands having a keyboard-shortcut, have the shortcut in brackets after the command name.

8.5.1 File Menu

8.5.1.1 Export Layout (Alt+L)

The layout of all sheets for the test currently displayed in the application window is output to the ‘layout.txt’ text file. Figure 31 provides an example of the exported data.

```
Test performed: 09/07/98 15:09:50
Test 100 of 124
Test Description: 13.6T) Left placed shape masking edges 2,3,4
in top profile
Sheet Size = 30x30
Number of Sheets Used = 1

Fitness Report:
Sheet 0: Area Used = 25x30

Overall Fitness Of Layout: 0.475127

Formatted Output:
Col 1: S=New Sheet, -=No New Sheet
Col 2: Shape ID
Col 3: Rotation=0 or 90 Degrees
Col 4: Shape Size: (X-size x Y-size)
Col 5: Location: (X , Y)

S ID=0      R=0D  10x5  (0,0)
- ID=1      R=0D  5x5   (10,0)
- ID=2      R=0D  11x6  (0,5)
- ID=3      R=0D  6x3   (15,0)
- ID=4      R=0D  11x19 (0,11)
- ID=5      R=0D  4x4   (21,0)
- ID=6      R=0D  14x6  (11,5)
```

Figure 31. Typical layout information output to the ‘layout.txt’ file.

8.5.1.2 Print (Ctrl+P)

Outputs the contents of the application window to a connected printer. Only the currently displayed stock-sheet is printed. To print multi-sheet layouts, navigate through each sheet and print them individually.

8.5.1.3 Print Setup

Allows printer-specific parameters to be set.

8.5.1.4 Exit

Quits the StockGA Application.

8.5.2 View Menu

8.5.2.1 Refresh (Alt+R)

Reloads the 'shapes.txt' layout information file. This command is to be used after a modification has been made to the layout file whilst the StockGA application is running.

8.5.2.2 Normalise Chromosome (Alt+N)

Lays out the shapes for the current test in the order specified in the test file. This corresponds to a shape-id permutation of 0, 1, 2..., n-1, where n is the number of shapes in the test.

8.5.2.3 Shuffle Chromosome (Alt+W)

Lays out the shapes for the current test in a random order.

8.5.2.4 Order Chromosome (Alt+O)

Lays out the shapes for the current test in an order specified by the user. The order is specified as a list of non-negative integers in the range 0 to n-1, where n is the number of shapes in the test. The integer list is verified for being a valid permutation.

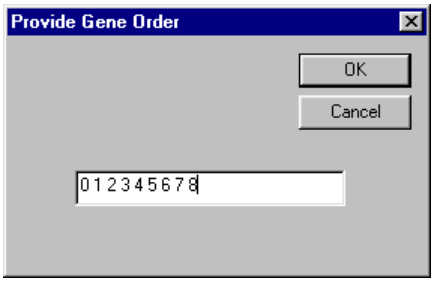

Figure 32. Dialog Box invoked when the Order Chromosome menu command is invoked.

8.5.2.5 Next Test (Alt+X)

Updates the application window with the layout and test information corresponding to the test following the test currently displayed.

8.5.2.6 Previous Test (Alt+Z)

Updates the application window with the layout and test information corresponding to the test preceding the test currently displayed.

8.5.2.7 Next 10th Test (Alt +D)

Changes the current test to the test which is the tenth test following the current test.

8.5.2.8 Previous 10th Test (Alt+A)

Changes the current test to the test which is the tenth test preceding the current test.

8.5.2.9 Goto Test... (Alt+G)

Changes the current test to a test specified by the user. Tests are identified by their position in the test file.

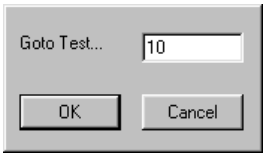

Figure 33. Dialog Box invoked when the 'Goto Test' menu command is invoked.

8.5.2.10 View Next Sheet (Alt+0)

Updates the display to show the next sheet for the current layout.

8.5.2.11 View Previous Sheet (Alt+9)

Updates the display to show the next sheet for the current layout.

8.5.2.12 Set Regime For This Test (Alt+T)

Allows the user to change the regime used for laying out shapes in the current test. The change is retained until the test file is reloaded using the 'Refresh' menu command.

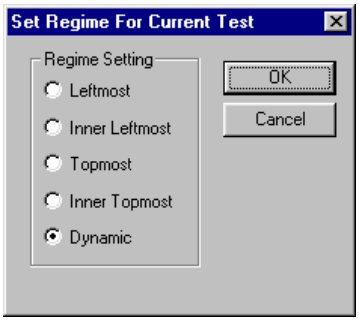

Figure 34. Dialog Box invoked when the 'Set Regime' menu command is invoked.

8.5.3 Evolve Layout Menu

8.5.3.1 GA Settings (Alt+S)

This command invokes the dialog in Figure 35.

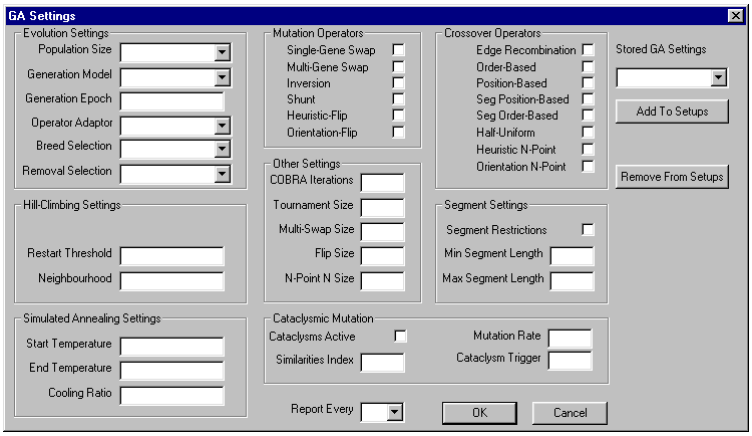

Figure 35. GA Settings Dialog Box used for configuring all GA experiment parameters.

Parameter	Possible Values			
Population Size	50	100	150	200
Generational Model	Steady-State GA	Hill Climbing	Steepest Ascent Hill Climbing	Simulated Annealing
Generation Epoch	Positive Integer Value – Number of generations to evolve			
Operator Adaptor	Adaptive Mutation	COBRA	Fixed Rate	Random
Breed Selection	Rank	Tournament		
Removal Selection	Least-Fit	Inverted Rank		

Table 4. Summary of values that evolution settings can have.

Settings Group	Parameter Name	Purpose
Hill Climbing Settings	Restart Threshold	If fitness is not improved after this no. of hill-climbing iterations, then the current solution is randomised.
	Neighbourhood	Number of samples to taken at each hill-climbing iteration in Steepest Ascent Hill Climbing
Simulated Annealing Settings	Start Temperature	Initial value that current temperature is set to
	End Temperature	Value for current temperature that terminates SA
	Cooling Ratio	Current temperature is multiplied by this factor at each iteration of SA
Segment Settings	Segment Restrictions	When selected, restricts segment crossover operators to segment lengths between Min and Max values.
	Min Segment Length	Minimum gene-length that segment crossover operators act upon
	Max Segment Length	Maximum gene-length that segment crossover operators act upon
Cataclysmic Mutation	Cataclysms Active	When selected, enables cataclysmic mutation
	Similarities Index	Ranged between 0 and 1 – determines how close a chromosome has to be in fitness to the best chromosome before being counted as a duplicate
	Cataclysms Trigger	Number of duplicates in the population that triggers cataclysmic mutation
	Mutation Rate	Ranged between 0 and 1 – determines percentage change made to all chromosomes except the best one, when cataclysmic mutation takes place
Other Settings	COBRA Iterations	Number of times that a COBRA rate adaptor may apply genetic operators at random to determine their relative usefulness
	Tournament Size	Number of chromosomes to be considered when tournament selection is used
	Multi-Swap Size	Number of pairs of genes to swap in multi-swap mutation
	Flip Size	Number of genes to mutate every time orientation mutation or heuristic mutation is used
	N-Point N Size	Number of crossover points to use in orientation n-point crossover and heuristic n-point crossover

Table 5. Summary of GA Settings Dialog parameters

Table 4 and Table 5 summarise the parameters that affect how the GA experiment runs. In addition to these, there are check-boxes that allow the user to select which mutation and crossover operators to use. The 'Report Every' parameter determines how often the GA reports its current status. The type of experiment is determined by the Generational Model parameter. The Steady-State value for this parameter enables a GA to run. All other settings for this parameter cause a stochastic experiment to run, such as Hill Climbing.

The 'Stored GA Settings' parameter provides a method for saving all current GA Settings in a file for future use. To enable this function, a name is entered into the 'Stored GA Settings' drop-down list-box and the 'Add to Setups' button is clicked. Previously stored setups appear in the drop-down box. The 'Remove From Setups' button allows setups to be erased. Setups are saved when the GA Settings Dialog is closed *using the OK button*. If the CANCEL button is clicked then any newly defined setups are not stored to disk. Setups are stored in a binary file called 'setups.bin'.

Certain sets of parameters are disabled depending on which Generational Model is selected. For example, only the Steady-State Generational Model uses crossover operators. Figure 36, Figure 37 and Figure 38 show how the GA Settings dialog appears for different Generational Model settings. For example, when this parameter is not set to Steady-State, all crossover operators are ghosted-out. Ghosted parameter fields are not active.

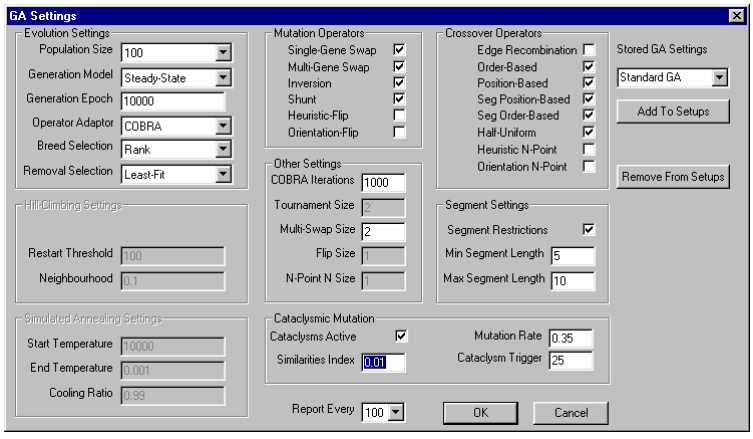

Figure 36. GA Settings Dialog status when Steady-State is selected

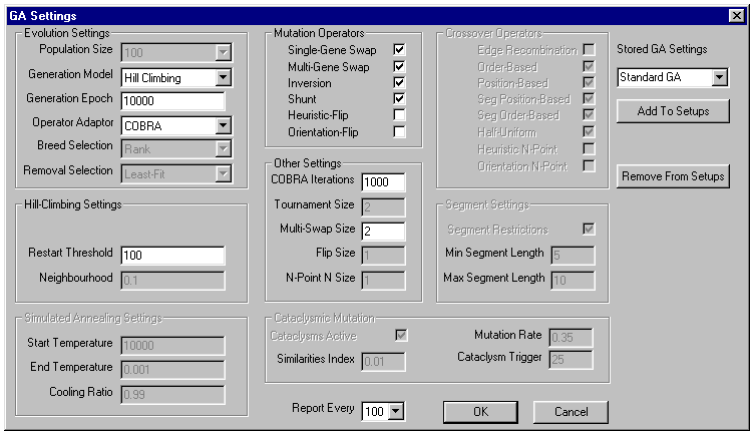

Figure 37. GA Settings Dialog status when Hill Climbing is selected

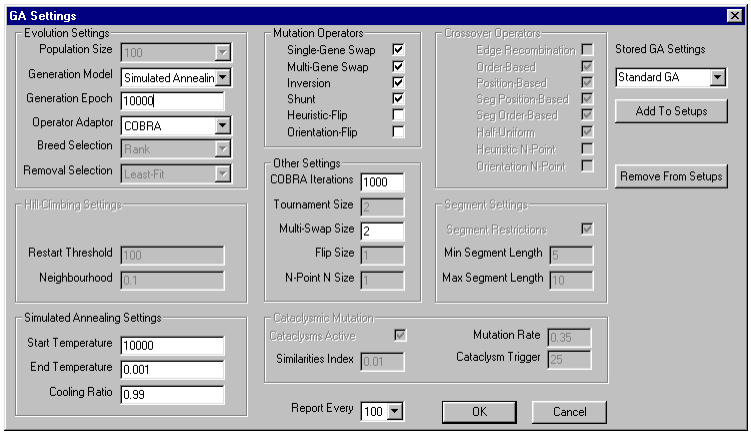

Figure 38. GA Settings Dialog status when Simulated Annealing is selected

8.5.3.2 Run GA (Alt+E)

This command runs the GA experiment using the parameters set in the GA Settings Dialog. Current status, throughout the experiment, is reported in the output window. After the experiment has concluded, the fittest chromosome has its layout displayed in the application window. Figure 30 illustrates the output window. Layout data can be written to disk using the 'Export Layout' command.

8.5.3.3 Batch Process (Alt+B)

This is the most powerful command available in the StockGA application. It's use is best described with an example. 'Experiments' may be set up with any valid combination of parameters present in the GA Settings dialog.

Let us say there are four different experiments to be each tested on three separate collections (or layouts) of shapes. The parameters for these four experiments are stored under the

following names in the GA Settings Dialog: set 1, set 2, set 3, set 4. The prefix can be any single word (i.e. no white-space) and must be common to all GAs to be tested. The proceding integer must be separated by a single space, begin at 1, and have no gaps in the numeric sequence.

So far we have defined 12 experiments to perform. It is necessary to perform each experiment a number of times, say 10, to obtain results that may be statistically analysed. So, including repetitions, there are 120 experiments to do.

To batch process the following sequence of actions are to performed by the user:

1) Store the sets of parameters for the experiments using a suitable naming convention in the GA Settings Dialog.
2) Set the currently displayed layout to the first one to be experimented with. All other layouts to be tested must immediately follow this first one in the sequence of tests stored in the input test file.
3) Select the 'Batch Process' menu command.
4) A dialog box, see Figure 39, appears in which the user specifies the number of layouts to process, the number of times to run each experiment and the prefix name pertaining to the set of GAs that are to be used. Clicking the 'OK' button begins the batch processing. All output is sent to the output window.

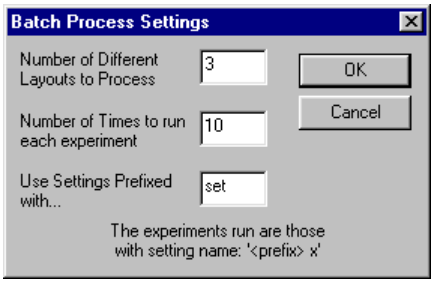

Figure 39. The batch process dialog box.

8.5.3.4 Detailed Output

This is a checked menu selection which is either on or off. When on, output produced by experiments is restricted to the best fitness obtained at the end of the experiment. When off, output is generated after every *i* generations have been evolved, where *i* is the setting of the

‘Report Every’ parameter in the GA Settings dialog. It is recommended that this option is set to off before batch processing.

8.5.4 Help Menu

8.5.4.1 About GA...

Displays a dialog box that summarises the use of the StockGA application.

9 Experimental Results

9.1 Introduction

This section presents experimental results obtained using the StockGA application. The aim of these experiments is to find how the system fairs in a variety of situations. These include, for example, comparing the performance of various GAs with Hill Climbing and with each other, on a selection of layout problems.

Raw data obtained from these experiments is presented in tabular form, in appendix 2. The results presented here include average fitness values obtained in each experiment and t-test which compares raw data. All experiments are repeated 10 times. The t-test is used to test the level of confidence with which we can make a conclusion, such as, 'GA1 is better than GA2 on shape-collection X'. The source code for the t-test program is in appendix 1, and was obtained from Mr D.W. Corne, Department of Computer Scince, University of Reading.

The layout problems used for testing the sequence are now presented. Two groups of layout problems are considered. These are the 'perfect-cut' problems and the problems presented in Bengtsson, [17].The size of shapes used in these layouts are in appendix 2.

9.1.1 Perfect-cut layout problems

Perfect-cut layouts are layouts which have zero wastage. A perfect-cut layout is made by taking a stock sheet and dividing the area into random sized, rectangular shapes. The process can be likened to cutting a jig-saw puzzle from a piece of wood. Once all shape dimensions are known, we 'forget' that we know the optimal layout, and present to the GA a random permutation of these shapes. It is the GA's job to find a good (and hopefully optimal) layout for these pieces. There may be more than one layout that satisfies zero wastage, due to rotations and equal-sized pieces, so the fitness-landscape is not necessarily uni-modal.

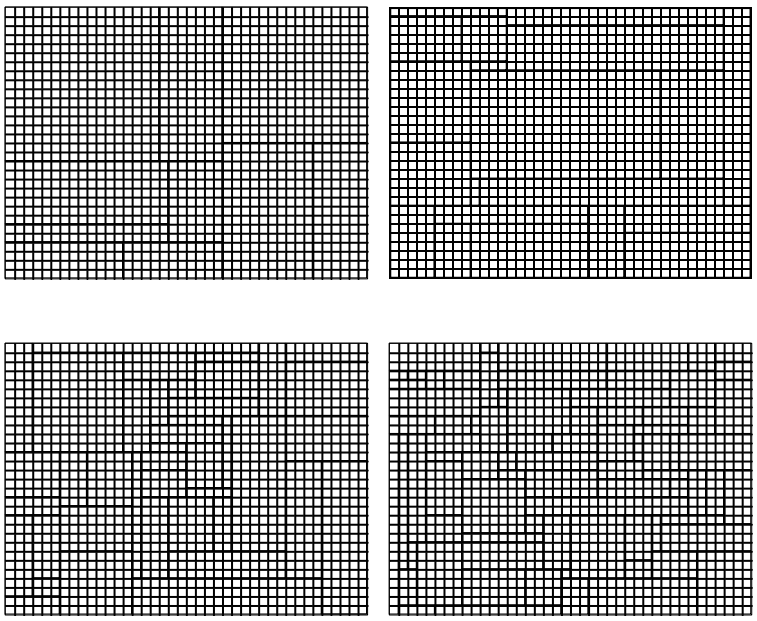

Figure 40. The collections of shapes, from top-left clockwise, which make up the shape collections, 10-perfect, 20-perfect, 80-prefect and 40-perfect.

Four collections of shapes are defined (see Figure 40), with varying numbers of pieces. They are referred to here as ‘10-perfect’, ‘20-perfect’, ‘40-perfect’ and ‘80-perfect’. Each is known to have at least one optimal solution, see . The shapes are ordered in the test file in such a way as to let the LDA generate the perfect-cut by applying the topmost heuristic on this exact ordering, if the shapes are placed on a 40x30 stock-sheet. This has the advantage that initial experiments can be performed which do not manipulate the heuristic or the orientation of the shapes. These tests offer a variety of shape-sizes in relation to the stock-sheet size.

Important Note: The 10-perfect layout has only 10 pieces and it has been found that all experiments carried out on this layout result in the optimal layout being produced. As such, no average fitness values are obtained, as the optimal fitness of 1 is obtained in every experiment. It is useful to compare the number of generations that have been required to obtain the optimal result when comparing 10-perfect experiments.

9.1.2 Bengtsson layout problems

These shape-collections have been obtained from Figure 17. The four collections are summarised in Table 6.

Layout Name	Number of Shapes	Stock-Sheet Size
B1	20	25x10
B2	40	25x10
B3	40	40x25
B4	80	40x25

Table 6. Bengtsson layout problems.

These layout problems involve placing shapes that are quite large in comparison with the stock-sheet size, limiting the number of shapes per sheet. These problems are known to require multiple sheets, whereas the perfect-cutting problems requires only one sheet. [17] includes many more shape collections for which results are presented.

9.1.3 GA Settings

9.1.3.1 Mutation Operator Codes

Mutation operators may be referred to by the following numbers:

M1. single gene swap (swap)
M2. multi-gene swap (multi-swap) – set to swap 2 pairs of genes at once (flip-size=2)
M3. inversion
M4. shunt
M5. Heuristic-Flip (H-Flip) – set to swap 2 pairs of genes at once (flip-size=2)
M6. Orientation-Flip (O-Flip)

9.1.3.2 Crossover Operators Codes

Crossover operators may be referred to by the following numbers:

C1. Edge Recombination (edge)
C2. order-based (order)
C3. position-based (position)
C4. segment position-based (seg-position)
C5. segment order-based (seg-order)
C6. Half Uniform (HUX)
C7. Heuristic N-point – 2-point
C8. Orientation n-point – 2-point

9.1.3.3 Global Settings

These settings are common to all experiments:

- Segment Restrictions are set to minimum of 2 genes and maximum of 7 genes in a segment.
- Least-fit removal selection is used in all experiments.

Cataclysmic Mutation is active in all steady-state GA experiments and is setup with the following parameters settings:

- Similarities index=0.01
- Cataclysms trigger=25
- mutation rate=0.35 (35%)

When in use, tourament selection is set to binary-mode, ie Selection size=2.
Restart threshold for Hill Climbing is always set to 100.

9.1.3.4 T-Test values

All experiments are repeated 10 times, so the t-values required for confidence levels of 90%, 95% and 99% are constant, see Table 7.

confidence level	t-value
90	1.33
95	1.734
99	2.552

Table 7. T-values required for confidence levels of 90%, 95% and 99% when comparing two sets of 10 results.

Negative t-values, which have larger magnitude, than any of the above values indicate that the hypothesis is true in the reversed sense.

9.2 Experiment Set 1: Comparing GAs with Hill Climbing and Random models.

9.2.1 Setup

This set of experiments uses the perfect-cutting problems. They investigate how population size and breed selection method affect output.The COBRA operator rate adaptor is used in all these experiments. The COBRA iteration point was set at 2000.

The GA settings investigated are presented in Table 8.

GA1	population=50, selection=tournament
GA2	population=100, selection=tournament
GA3	population=50, selection=rank
GA4	population=100, selection=rank
Hill Climb	Standard Hill Climbing
Random	Random Generation of chromosomes

Table 8. GA Settings used in experiment set 1.

Mutation Operators used: M1, M2, M3, M4
Crossover Operators used: C1, C2, C3, C4, C5, C6

9.2.2 Results

Each of these experiments were used in each of the perfect-cutting problems, and the average fitness results are presented in Table 9. Note: 10-perfect results, here and elsewhere in this report, are reported in terms of number of generations required to obtain the optimal solution.

20-perfect and 40-perfect tests were run for 10000 generations. 80-perfect tests were run for 5000 generations due to the processing time required.

Shape-Set	GA1	GA2	GA3	GA4	Hill Climb	Random
10-perfect	530	370	320	340	320	1040
20-perfect	0.91628	0.89866	0.91744	0.91088	0.87232	0.816987
40-perfect	0.85611	0.86225	0.86701	0.86281	0.819163	0.749878
80-perfect	0.84323	0.85251	0.84513	0.83592	0.831184	0.775088

Table 9. Average best-fitness for set 1 experiments. 10-perfect results refer to number of generations required to evolve optimal layout.

9.2.3 Analysis

The results of t-tests comparing Hill Climbing with the GAs are presented in Table 10. The results for 10-perfect have been multiplied by –1, as this is a minimising (of generation-count) problem, rather than a maximising problem (as in trying to obtain optimal fitness).

Hypothesis	10-perfect	20-perfect	40-perfect	80-perfect
GA1 is better than HC	-1.36 (-90%)	3.00 (99%)	3.08 (99%)	1.17 (-)
GA2 is better than HC	-0.36 (-)	2.53 (95%)	5.06 (99%)	2.31 (90%)
GA3 is better than HC	0 (-)	3.31 (99%)	5.72 (99%)	1.47 (90%)
GA4 is better than HC	-0.14 (-)	2.95 (99%)	4.90 (99%)	0.53 (-)
HC is better than Random	2.91 (99%)	5.78 (99%)	6.94 (99%)	6.14 (99%)

Table 10. T-Test results 1 for experiment set 1.

Firstly, the results show that Hill Climbing is significantly better than random chromsome generation in all cases. The GAs are decisively more competent than Hill Climbing when used on medium-sized problems, ie 20 or 40 shapes. For 10-perfect, the results suggest that due to the small number of pieces, the choice of Hill Climbing or GA is not important. The optimal result was obtained in all cases for 10-perfect. The size of the search-space in the 80-perfect problem is much larger than in the others, which may explain the GAs poorer performance.

Some additional t-tests were performed to make some comparisons between GAs. These results are presented in Table 11.10-perfect tests were not examined.

Hypothesis	20-perfect	40-perfect
GA3 is better than GA1	0.06 (-)	1.099 (-)
GA3 is better than GA2	1.19 (-)	0.92 (-)
GA3 is better than GA4	0.37 (-)	0.72 (-)
GA1 is better than GA4	0.29 (-)	-0.646 (-)

Table 11. T-Test results 2 for experiment set 1.

These tests were picked by comparing t-test values in Table 10. The results show that the GAs are roughly equivalent in performance in the 20-perfect and 40-prefect tests.

9.3 Experiment Set 2: Does restricting the number of genetic operators improve final fitness?

9.3.1 Setup

In this set of experiments, the number of genetic operators used in GA1, GA2, GA3 and GA4 are reduced – these GA settings are referred to as New GA1 – New GA4. Some operators are picked for removal, based on the COBRA score results obtained in the previous experiment set. The next experiment presents the COBRA score results. The COBRA operator rate adaptor is used in all these experiments. The COBRA iteration point was set at 2000.

The rationale for doing this is because if the poorer operators (ie the ones which generate the fewest good children) are removed, the operators that do better will be used more often as there are less operators now to pick from.

Mutation operators used are: M1, M2 (swap mutations)
Crossover operators used are: C4, C5, C6

Again, 20- and 40-perfect tests were run for 10000 generations and 80-perfect for 5000 generations.

9.3.2 Results

Table 12 presents the results for this experiment.

Shape-Set	New GA1	New GA2	New GA3	New GA4
10-shape	370	400	530	410
20-shape	0.921087	0.934502	0.908174	0.908738
40-shape	0.859137	0.858343	0.847191	0.86438
80-shape	0.841805	0.842049	0.844179	0.841782

Table 12. Average best-fitness for set 2 experiments. 10-perfect results refer to number of generations required to evolve optimal layout.

9.3.3 Analysis

Hypothesis	10-perfect	20-perfect	40-perfect	80-perfect
NewGA1 better than GA1?	1.20 (-)	0.23 (-)	0.27 (-)	-0.14 (-)
NewGA2 better than GA2?	-0.28 (-)	2.23 (95%)	-0.40 (-)	-1.24 (-)
NewGA3 better than GA3?	-1.38 (-90%)	-0.42 (-)	-1.83 (-95%)	-0.08 (-)
NewGA4 better than GA4?	-0.49 (-)	-0.15 (-)	0.23 (-)	0.55 (-)

Table 13. T-Test results for experiment set 2.

These results show that the New GA settings are not significantly better in terms of maximum fitness than the original GA settings. In fact, in some instances the New GAs are worse than the original GAs. This result should be kept in mind whilst the next experiment set is considered.

9.4 Experiment Set 3: Are more ‘good’ children generated when fewer, better genetic operators are used?

9.4.1 Setup

The operator scores output during the processing of GA2 and New GA2 are now considered. The scores obtained from processing the 20-perfect and 40-perfect problems with these settings are summed. The COBRA score is based on 2000 generations where a random operator has been chosen to generate a single child at each generation. Crossover operators have one added to their score if a child is generated better than one parent, or two added if a

child is better than both parents. Two is added to the score for a mutation operator if a child better than the parent is generated.

9.4.2 Results

Operator Name	Total Score
segment_order	1096
segment_pos	1059
HUX_pos	906
position	809
swap	794
order	784
multi-swap	680
invert	576
shunt	520
edge	314
Total	7598

Table 14. COBRA scores summed over 10 experiments on 20-perfect problem and 10 experiments on 40-perfect problem (using the GA 2 setting in both cases).

Operator Name	Total Score
HUX_pos	2070
seg_pos	1981
seg_ord	1955
swap	1234
multi-swap	1068
Total	8308

Table 15. COBRA scores summed over 10 experiments on 20-perfect problem and 10 experiments on 40-perfect problem (using theNew GA 2 setting in both cases).

9.4.3 Analysis

Firstly, it can be seen from the Table 14 that the segment-order, segment-position and HUX are the best crossover operators, and swap and multi-swap are the best mutation operators. Edge recombination does very poorly and is rated poorer than all the mutators. This is surprising as its use in the Travelling Salesman Problem (which is also a permutation-based problem) is noted.

It can be seen that the total number (8308) of good children generated in New GA 2 is around 10% higher than the total number good children generated in GA 2. This suggests that restricting the number of operators is a good idea. This counters the evidence found in Experiment Set 2.

9.5 Experiment Set 4: Comparison between operator rate adaptors.

9.5.1 Setup

GA 2 supplies all parameters, except for the operator rate adaptor. Tests are carried out on the perfect-cut problems, except for 10-perfect which doesn't run for sufficient generations. All tests are conducted over 5000 generations.

adaptor1	adaptive mutation – based on permutation distance measure
adaptor2	fixed-rate – 80% chance of crossover
adaptor3	COBRA – 2000 iterations for accessing score
adaptor4	random operator

Table 16. Adaptor settings for set 4 experiments.

9.5.2 Results

Shape-Set	adaptor1	adaptor2	adaptor3	adaptor4
20-shape	0.913003	0.893612	0.886935	0.883374
40-shape	0.854831	0.828225	0.857171	0.827536
80-shape	0.841064	0.846418	0.851754	0.847173

Table 17. Average best-fitness for set 4 experiments.

9.5.3 Analysis

Adaptor 1, adaptive mutation is significantly better than all the other adaptors. It achieves 95% confidence in beating adaptors 2, 3 and 4 in at least one test each. It has a total of four confidence levels of 95% when compared to the other adaptors. The only other relevant data is that adaptor 2 is poorer than adaptor 3 with confidence of 95%. The COBRA rate adaptor has a confidence of 99% in being better than the random adaptor in one test. The lack of

significant evidence to support COBRA being good is surprising, as the scoring system is intuitively better than say fixed rate.

Hypothesis	20-perfect	40-perfect	80-perfect
adaptor1 is better than adaptor 2	1.10 (-)	1.95 (95%)	-0.57 (-)
adaptor1 is better than adaptor 3	2.08 (95%)	-0.21 (-)	-1.27 (-)
adaptor1 is better than adaptor 4	2.44 (95%)	2.26 (95%)	-0.68 (-)
adaptor2 is better than adaptor 3	0.42 (-)	-2.45 (-95%)	-0.56 (-)
adaptor2 is better than adaptor 4	0.65 (-)	0.05 (-)	-0.07 (-)
adaptor3 is better then adaptor 4	0.37 (-)	2.97 (99%)	0.50 (-)

Table 18. T-Test results for experiment set 4.

9.6 Experiment Set 5: Comparison between placement heuristics.

9.6.1 Setup

The GA uses all operators, has an adaptive mutation rate adaptor and evolves for 5000 generations. In this set of experiments, the topmost, flip-top and flip-left heuristics are investigated. The flip-top heuristic alternates between placing shapes topmost and leftmost, beginning with topmost. The flip-left heuristic does the same except for starting instead with leftmost.

9.6.2 Results

Layout	topmost	flip top	flip left
B1	0.668537	0.675138	0.689865
B2	0.422592	0.427987	0.431285
B3	0.796677	0.697526	0.697725
B4	0.636764	0.59898	0.604725

Table 19. Average best-fitness for set 5 experiments.

9.6.3 Analysis

Hypothesis	B1 Layout	B2 Layout	B3 Layout	B4 Layout
topmost is better than flip-left	-0.58 (-)	-0.2544 (-)	6.66 (99%)	2.61 (99%)
topmost is better than flip-top	-2.019 (-95%)	-0.39 (-)	7.74 (99%)	2.87 (99%)
flip-left is better than flip-top	-1.49 (-90%)	-0.188 (-)	-0.015 (-)	-0.455 (-)

Table 20. T-Test results for experiment set 5.

There is evidence that the topmost heuristic is better than both flip-left and flip-top on layouts containing a large number of shapes, ie B3 and B4. Topmost is shown to be worse than flip-top for the 20 piece, small sheet layout, B1.

9.7 Experiment Set 6: Evolving permuation, orientation and heuristic data.

9.7.1 Setup

Each GA investigates a different combination of using permuation-based, orientation-based or heuristic-based evolution. Orientation-based allows shapes to be rotated and for these rotations to be evolved. Heuristic-based allows the heuristic used to place the shapes to be evolved.

The edge recombination operator is not used in any of these experiments as it cannot preserve heuristic and orientation features in the genes. All experiments evolve over 5000 generations.

nbeng1	permutation, orientation, heuristic
nbeng2	permutation, orientation
nbeng3	permutation, heuristic
nbeng4	permutation, Adaptive Mutation
nbeng5	permutation, COBRA

9.7.2 Results

Layout	nbeng1	nbeng2	nbeng3	nbeng4	nbeng5
B1	0.684073	0.713854	0.655146	0.656309	0.667802
B2	0.437444	0.513875	0.445407	0.474788	0.455639
B3	0.749914	0.819836	0.763233	0.830969	0.806872
B4	0.598091	0.673151	0.608624	0.670892	0.663572

Table 21. Average best-fitness for set 6 experiments.

9.7.3 Analysis

NBeng2 is better than NBeng3 and NBeng1 in t-tests for all layouts. This implies that evolving the heuristic is not an effective way of increasing fitness, and actually decreases it. NBeng2 is also better than permutation alone for layouts B1 and B2. It appears that evolving just the orientation and permutation is the best course of action.

Hypothesis	B1 Layout	B2 Layout	B3 Layout	B4 Layout
1 beats 2	-2.31 (-95%)	-4.11 (-99%)	-4.18(-99%)	-4.95(-99%)
1 beats 3	2.22 (95%)	-0.49 (-)	-0.71(-)	-0.71 (-)
1 beats 4	2.33 (95%)	-2.33 (95%)	-5.21(-99%)	-5.88 (-99%)
1 beats 5	1.68 (90%)	-1.05 (-)	-3.81(-99%)	-5.33 (-99%)
2 beats 3	3.92 (99%)	4.60 (99%)	3.71(99%)	4.32 (99%)
2 beats 4	4.1 (99%)	2.69 (99%)	-0.99(-)	0.180 (-)
2 beats 5	3.77 (99%)	3.65 (99%)	1.25(-)	0.77 (-)
3 beats 4	-0.08 (-)	-2.54 (95%)	-4.85(-99%)	-5.14 (-99%)
3 beats 5	-1.02 (-)	-0.77 (-)	-3.28(-99%)	-4.58 (-99%)
4 beats 5	-1.02 (-)	1.48 (90%)	2.87(99%)	0.82 (-)

Table 22. T-Test results for experiment set 6

9.8 Experiment Set 7: Varying size sheet

9.8.1 Setup

The GA uses Adaptive Mutation, 100 population, tournament selection and all operators except edge recombination. A dynamic heuristic was used which allowed the heuristic to evolve. The tests required the GA to build the optimal layout without the help of having a constant topmost heuristic (which is known to produce the optimal solution). Tests were

conducted on 40x30 sheet-size (original size), a sheet twice as wide and a sheet twice as tall also.

9.8.2 Results

Layout	40x30 (Normal)	80x30 (Wide)	80x60 (Wide & Tall)
10 shape-perfect	400	1740	0.925241
20 shape-perfect	0.828133	0.895198	0.852977
40 shape-perfect	0.823958	0.871963	0.872283
80 shape-perfect	0.812949	0.884286	0.88204

Table 23. Average best-fitness for set 7 experiments.

9.8.3 Analysis

The t-test results confirm that making the sheet wider, or wider and taller, improves the fitness results over those gained for the dynamic layout on the normal sized sheet.

Hypothesis	20-perfect	40-perfect	80-perfect
normal sheet is better than wider sheet	-7.512 (-99%)	-4.4 (-99%)	-9.178 (-99%)
normal sheet is better than wider, taller sheet	-1.95049 (-95%)	-3.85 (-99%)	-9 (-99%)
wider sheet is better than wider, taller sheet	3.41577 (99%)	-0.038 (-)	0.32 (-)

Table 24. T-Test results for experiment set 7

10 Conclusions

The analysis of the experimental results has revealed a number of interesting points. Firstly, it is true to say that varying the population size and the breed selection method do not unduly affect the outcome of the GA experiment. This didn't appear to be affected by the size of the layout problem.

Secondly, it has been found that restricting the number of operators being used increases the number of good children introduced into the population. However, it must be remembered that the COBRA rate adaptor, from where these facts were obtained, analyses the results of

breeding 2000 generations, when the experiment may run to 10000 generations. The point is that although the restricted selection of operators may produce a larger number of good children in the first 2000 generations, after this the limited variety in the types of operator available prevents the population from reaching fitness levels that it could obtain if a wider range of operator were available. This is true even if these operators don't usually produce good children themselves. By introducing new gene sequences into the population they are assisting the better operators by providing a larger gene pool to manipulate. This is why although the COBRA adaptor showed that more good children were being produced in the restricted case, the final fitness averages and t-test analysis showed that this reduced the best fitness in the population, after 5000 or 10000 generations.

A surprising result was the very poor performance of the edge recombination crossover operator. This operator has been used successfully in the Travelling Salesman Problem(TSP). In the TSP it is not so much the order that cities are visited, but the total distance covered. Thus, in TSP the optimum route can be traversed in either direction. This is patently does not apply to the permutation-based GA, as a reversal of the chromosome would indeed affect the fitness. It is for this reason why edge recombination is bad for stock-cutting problem. It considers edges between genes, but not the direction of the edge.

It was also found that adaptive mutation is better than the COBRA rate adaptor. The main driver of evolution in the early stages of evolution is the crossover operator as there is much genetic variation but not much organisation, thus in the COBRA operator score assessments crossover operators were found to be much better than the mutation operators. However, as the population converges, the use of mutation becomes more important as crossover between similar parents is not useful. The factors that drive increased fitness vary as the number of generations evolved increases. The COBRA rate adaptor determines how good operators are in the early stages of evolution. It then assumes that these are constant. Adaptive mutation chooses whether to apply a crossover or mutation operator based on how similar two randomly picked chromosomes are and so overcomes this problem.

Using a constant heuristic, such as topmost, was found to be more useful than a heuristic which switches between left and top at every gene along the chromosome. This is understandable because having a heuristic which depends so much on gene location is

restricting. A gene would have an evens chance of having its placement heuristic changed whenever it moved position in the chromosome.

It was found that allowing the heuristic to evolve (a dynamic heuristic) produced results that were significantly worse than when just orientation and permutation could change in the gene. A major factor was probably the use of simple, non-specialised operators to manipulate the heuristic, such as n-point crossover. Considering this, and the previous observation, it is advisable to stick with one heuristic, or perhaps two very similar heuristics such as topmost and inner-topmost (where the shape is preferred to be away from the sheet boundaries).

The last experiment investigated how the dimensions of the stock-sheet affected layout packing. These experiments used a dynamic a heuristic and it was found that it was preferable to provide a larger stock-sheet.

11 Summary and Future Work

11.1 The Layout Determining Algorithm

The LDA was a substantial part of the project, due partly to very few details concerning design and implementation of LDAs in the literature. It is for this reason that I have chosen to detail the algorithm so thoroughly. The LDA offers a variety of placement heuristics: leftmost, topmost, inner-leftmost, inner-topmost, flip-left and flip-top for the non-guillotine stock-cutting problem. In addition to this it can extract heuristic information from the chromosome. This is, to my knowledge, a unique feature. Allowing the GA to choose a heuristic has been shown, in this instance, to not be very useful in terms of poorer overall fitness results. However, with the development of suitable operators which can manipulate orientation, heuristic and permutation features in unison, rather than independently, these results may be improvable. By using topmost and inner-topmost heuristics it is possible to evolve the optimal solution to the ‘unsolvable by bottom-left heuristics’ example in Figure 3. This is a most definite improvement.

A heuristic that could be included in the LDA in the future is the ‘shortest anchor vector’ heuristic. This chooses heuristic chooses the location that has shortest distance from top-left vertex to the top-left vertex of the stock-sheet.

11.2 The StockGA application

The application is a versatile, visual and flexible tool which allows a large variety of GAs and stochastic methods to be studied. Strengths include the facility for loading and navigating through unlimited layout problems and for storing unlimited sets of GA parameters, allowing the user go back to previous work without additional effort.

The only drawback with the application is that it cannot use a text output window when the application is executed from outside the Visual C++ environment. All program functionality is preserved except for this output. The problem arises due to multi-threading issues which prevents a window from being updated while an experiment is running. Improving this functionality would be a priority.

GA fitness results are relatively good, but could do with some improvement, perhaps by implementing new specialist operators.

12 References

[1] Kendall, G., Stock cutting literature review, Computer Science, Nottingham University, *WWW location: http://www.asap.cs.nott.ac.uk/ASAP/stockcutting/stock.html* 1998

[2] Brooks, R.L., Smith, C.A.B., Stone, A.H., Tutte, W.T., The dissection of rectangles into squares. In *Duke Math. Journal.,* Vol. 7, pp 312-340

[3] Dantzig, G.B., Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production Allocation. Koopmans, T.C, Ed.* Cowles Commission Monograph, 13. Wiley, New York, pp 339-347

[4] Haessler, R.W., A heuristic solution to a nonlinear cutting stock problem. In *Management Science,* Vol. 17 (1971), B793-B803

[5] Dyson, R.G., Gregory, A.S., The cutting stock problem in the flat glass industry. In *Operations Research Quarterly*, Vol. 25 (1974), pp 41-53

[6] Lagus, K., Automated pagination of the generalized newspaper using simulated annealing". *Master's thesis, Helsinki University of Technology* 1995

[7] Kouvelis, P., Chiang, W.-C., and Fitzsimmons, J. Simulated annealing for machine layout problems in the presence of zoning constraints. *European Journal of Operational Research,* Vol. 57 (1992), pp 190-202

[8] Heragu, S.S., Recent models and techniques for solving the layout problem. *European Journal of Operational Research,* Vol. 57 (1992), pp 136-144

[9] Heragu, S.S and Alfa, A.S., Experimental analysis of simulated annealing based algorithms for the layout problem. *European Journal of Operational Research*, Vol. 57 (1992), pp 190-202.

[10] Hwang, S.-M., Kao, C.-Y., and Horng, J.-T. On solving rectangle bin packing problems using genetic algorithms. In *Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics,* Vol. 2 (1994), pp 1583-1590. IEEE Press, Piscataway, NJ, USA.

[11] Coffman, E.G. Jr, Garey, M.R., Johnson D.S., Tarjan, R.E., Performance bounds for level-oriented two-dimensional packing algorithms. In *SIAM Journal of Computing*, vol. 9 (1980), No. 4, pp 808-826. Society for Industrial and Applied Mathematics.

[12] Baker, B.S., Coffman, E.G. Jr, Rivest R.L., Orthogonal packings in two dimensions. In *SIAM Journal of Computing*, vol. 9 (1980), No. 4, pp 846-855. Society for Industrial and Applied Mathematics.

[13] Johnson, D.S., Demers, A., Ullman, J.D., Garey, M.R., Graham, R.L, Worst-case performance bounds bounds for simple one-dimensional packing algorithms. In *SIAM Journal of Computing*, vol 3 (1974), pp 299-325. Society for Industrial and Applied Mathematics.

[14] Coffman, E.G., ed., Computer and Job/Shop Scheduling Theory. John Wiley, New York, 1976.

[15] Garey, M.R., Johnson, D.S, Computers and Intractability: A guide to the theory of NP-Completeness, Freeman, San Francisco, 1979.

[16] Erdos, P., Graham, R.L., On packing squares with equal squares, Tech. Rep. STAN-CS-75-483, Computer Sciences Dept., Stanford University, March, 1975.

[17] Bengtsson, B.-E., Packing rectangular pieces – a heuristic approach. In *The Computer Journal*, Vol. 25, No. 3, 1982, pp353-357.

[18] Adamowicz, M., Albano, A., A solution of the rectangular cutting-stock problem. In *IEEE Transactions on Systems, Man and Cybernetics* SMC-6 (No. 4), pp302-310 (April 1976)

[19] Albano, A., Orsini, R., A heuristic solution of the rectangular cutting stock problem. In *The Computer Journal*, vol. 23, no. 4, pp338-343

[20] Beasley, J., E., An exact two-dimensional non-guillotine cutting tree search procedure. In *Operations Research*, Vol. 33, No. 1, January-February 1985, pp 49-64. Operations Research Society of America.

[21] Gilmore, P.C., Gomory, R.E., Multistage cutting problems of two and more dimensions. In *Operations Research*, Vol. 13 (1965), pp 94-120. Operations Research Society of America.

[22] Gilmore, P.C., Gomory, R.E., The theory and computation of knapsack functions. In *Operations Research*, Vol. 14 (1966), pp 1045-1075. Operations Research Society of America.

[23] Herz, J.C., A recursive computing procedure for two-dimensional stock cutting. In *I.B.M. Journal of Research and Development*, Vol. 16 (1972), pp 462-469. Operations Research Society of America.

[24] Christofides, N., Whitlock, C., An algorithm for two-dimensional cutting problems. In *Operations Research,* Vol 25 (1977), pp 30-44. Operations Research Society of America.

[25] Madsen, O.B.G., References concerning the cutting stock problem. IMSOR, The Technical University of Denmark, DK-2800, Lyngby, Denmark. 1980

[26] Corno, F., Prinetto, P., Rebaudengo, M., Sonza Reorda, M., Bisotto, S., Optimizing Area Loss in Flat Glass Cutting. In *GALESIA '97, IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems,* IEE Press, 1997

[27] Prinetto, P., Rebaudengo, M., Sonza Reorda, M., Hybrid genetic algorithms for the travelling salesman problem. In *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, Innsbruck, (A), 1993, pp. 559-566

[28] Ono, T., Watanabe, G., Genetic algorithms for optimal cutting. In *Evolutionary Algorithms in Engineering Applications, Dasgupta, D., Michalewicz, Z., (Eds.),* Springer 1997

[29] Azar, Y., Epstein, L., On two dimensional packing. In *Journal of Algorithms*, Vol. 25 (1997), pp 290-310

[30] Lai, K.K., Chan, J.W.M., Developing a simulated annealing algorith for the cutting stock problem. In *Computers and Industrial Engineering*, Vol. 32 (1997), No. 1, pp 115-127

[31] Lai, K.K., Chan, J.W.M., An evolutionary algorithm for the rectangular cutting stock problem. In *International Journal of Industrial engineering*, Vol. 4(1997), No. 2, pp 130-139

[32] Kirkpatrick, S., Gelatt, C. D., Vecchi, P.M., Optimization by simulated annealing. In *Science*, Vol. 220 (1983), pp671-680

[33] Herbert, E.A., Dowsland, K.A., A family of genetic algorithms for the pallet loading problem. In *Annals of operations research*, Vol. 63 (1995), pp 415-436

[34] Holland, J.H., Adaption in natural and artificial systems. *The University of Michigan Press,* Ann Arbor, Michigan, 1975

[35] Grefenstette, J.J., Optimization of control parameters for genetic algorithms. In *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 12 (1986), pp 122-128

[36] DeJong, K., The analysis and behaviour of a class of genetic adaptive systems. *PhD thesis, University of Michigan, 1975.*

[37] Bellman, R., Dynamic Programming. *Princeton University Press.* 1957

[38] Beasley, D., Bull, D.R., Martin, R.R., An Overview of Genetic Algorithms : Part 1, Fundamentals. In *University Computing*, Vol. 15, No. 2, pp 58-69

[39] Ackley, D.H., An empirical study of bit vector function optimization. In *Davis, L., Ed., Genetic algorithms and simulated annealing,* chapter 13, pp 170-204. Pitman, 1987

[40] Goldberg, D.E., *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989

[41] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-verlag, Berlin, third edition, 1996.

[42] Davis L., *Genetic Algorithms and Simulated Annealing*. Pitman, 1987.

[43] Antonisse, H. J., A new interpretation of the schema notation that overturns the binary encoding constraint. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1989

[44] Davis, L., (ed.), *Handbook of genetic algorithms*, Van Nostrand Reinhold, 1991

[45] Goldberg, D., Simple genetic algorithms and the minimal deceptive problem. In *Genetic Algorithms and Simulated Annealing,* L. Davis (ed.) Pitman 1987.

[46] Goldberg, D., A note on Boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing, L. Davis (ed.) Pitman 1987

[47] Ackley, D., *A connectionist machine for genetic hill climbing*, Kluwer Academic Publishers, 1987

[48] Whitley, D., *Scheduling Problems and Travelling Salesman: The Genetic Edge Recombination Operator*, Computer Science Dept., Colorado State University, 1989.

13 Acknowledgements

I would like to acknowledge the assistance and support of the following people: David Corne for supervising my project and for providing an excellent course in Evolutionary Techniques at Reading University, without which I doubt my interest in the GA field would have developed; Sarah Murgatroyd for our interesting discussions concerning the layout determining algorithm; and Rachel McCrindle for organising the Information Systems Engineering M.Sc. course.