



Automatic layout of variable-content print data

Eldan Goldenberg
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-286
October 23rd, 2002*

E-mail: eldango@cogs.susx.ac.uk

print layout,
paste-up,
floorplanning,
genetic
algorithm,
self-organising
documents,
BICAS

The increasing quantity of data held by organizations about individuals, and the recent development of digital press capable of one-off printing at a quality rivalling offset machines, have created a demand for a method to automatically generate page layouts. The present solutions to this are either to use skilled labour to hand-design each page, or constrain the design tightly by fitting everything to a template, both of which have significant drawbacks. This thesis describes a Genetic Algorithm (GA) that automatically generates page layouts, without the need for costly and time-consuming human design, and with considerably more flexibility than a template-based approach. The GA is based on related work in VLSI floorplanning, which is described and adapted for the print context. This method was found to produce attractive layouts with a relatively small number of iterations, even though the only explicit goal in the program was to minimize wasted space. Visual representations of the layout are presented and discussed, together with an analysis of the search space and the speed with which the GA finds a solution. The range of document types for which this method produces attractive layouts is considered, and finally suggestions are made for future work, which would make the system into a more complete layout generation tool. The key novel ideas in this thesis are summarised in a patent application entitled 'Page composition' submitted by Hewlett-Packard to the UK Patent Office on Friday 30th August 2002.

Automatic layout of variable-content print data

MSc Dissertation

Eldan Goldenberg (eldango@cogs.susx.ac.uk)

MSc in Evolutionary and Adaptive Systems, August 2002

**School of Cognitive & Computing Sciences
University of Sussex, Brighton, UK**

**Research conducted at HP Labs, Bristol
May – August 2002**

Acknowledgements

I would like to thank my supervisors – Paul Layzell & Dave Cliff at HP Labs, and Chris Thornton at Sussex – for their guidance and encouragement.

Contents

1. Introduction	5
2. Background	6
2.1. Existing software	6
2.2. Work on related problems.....	7
2.2.1. Fitting websites into variable sized windows.....	7
2.2.2. Bin packing problems	7
2.2.3. Knapsack packing	9
2.3. Techniques	9
2.3.1. Genetic Algorithms	9
2.3.2. 'Hybrid' Genetic Algorithms	10
2.3.3. Non-evolutionary approaches	10
2.4. Summary	10
3. The program	11
3.1. Choice of approach.....	11
3.2. Method of the original papers	11
3.2.1. Problem representation.....	11
3.2.2. Solution evaluation.....	12
3.2.4. Mutation and crossover operators.....	16
3.2.4. Cohoon et al.'s parallel GA	16
3.3. Technical details of the program	17
3.4. Summary	17
4. GA Results.....	18
4.1. Parameter settings	18
4.2. Comparison with known data	18
4.3. Performance on randomly generated data	19
4.3.1. A note on measuring the speed with which a GA finds a solution.....	19
4.3.2. Results from random data	20
4.4. Summary	22
5. The size of the search space.....	23
6. Evaluating the GA in the print context	26
6.1. Aesthetic quality of layouts produced by the GA	26
6.2. Controlling the aspect ratio.....	29
6.2.1. Incorporating the aspect ratio into the cost function	30
6.2.2. Incorporating the aspect ratio into the area calculation	31
6.2.3. Allowing relaxation of the aspect ratio constraints	32
6.3. Grouping items together	33
6.3.1. Division into two groups.....	33
6.3.2. Division into more than two groups	34
6.4. Summary	35
7. Discussion & further extensions.....	36
7.1. Constraints on the selection of objects.....	36
7.2. Strategies for adding whitespace.....	36
7.3. Taking account of content	37
7.4. Summary	37
8. Conclusion.....	38
References.....	39
Appendix A: Table of results	41

1. Introduction

As marketers gather ever more detailed information about consumers and demographic groups, there is an increasing need for the ability to automatically generate customised documents for individual recipients. Systems do exist for this, but they are currently limited to template-based layouts, and lack the flexibility to deal with unspecified numbers of print objects, where these objects may have variable sizes and proportions. This project aims to lay groundwork for a more powerful self-organising document architecture, by adapting techniques used for formally related and well-studied optimisation problems from different domains.

After high-level decisions are taken, such as the overall look and feel of documents to be created, there are two stages to the production of an individual document: Content selection and layout. First a set of items have to be chosen from a database such that they fit onto a fixed-size page, while maximising the value of the contents. To generate documents for individual recipients, this content selection could be based on scoring each item in a database for how interesting it is likely to be to the chosen recipient or group of recipients. Once a selection of items has been made, they must be fitted into the available space, with the additional constraints of ordering and grouping the objects in a meaningful way, and making the result aesthetically pleasing.

Both parts of this task are analogous to general classes of problem that have been well studied for different domains. The selection of content can be seen as a knapsack-packing problem, and fitting items into a finite space is a form of bin packing, both of which are discussed in more depth in Section 2.

For a purely template-based solution, these two sub-problems are independent, because the contents can be selected with the prior knowledge that the template has n spaces for items of each type, so precisely n items should be chosen. In a more flexible system this is not the case; a more space-efficient layout allows more items to be selected, the selection of more flexibly displayed items allows for a more space-efficient layout, and if the layout produces a space of a certain size, an additional print object of the right size and shape could be selected to fill that gap. Due to restrictions of time and space, this study will only approach the problem of layout, but ultimately a working system would have to optimise both the content selection and the layout together.

The remainder of this report is structured as follows: Section 2 describes the problem in more detail, along with relevant previous work and the techniques that have been used. Section 3 outlines the workings of the layout system I have developed, while Section 4 presents the results of a number of test runs. Section 5 analyses the search space of this problem and compares my approach to some alternative methods. Section 6 evaluates the aesthetic quality of layouts produced by this layout engine and describes some extensions that make it more useful, and Section 7 suggests future directions for continued development of this system.

2. Background

The value to organisations of being easily able to send a message to a list of people has long been recognised. Computers are already used to automate this task to some extent, by providing the ability to draw names and addresses from a database and paste them into a standard letter. This helps to give recipients the impression that the letter they receive has been drafted for them individually, as opposed to a “Dear car owner” letter, or letters with blank spaces to be filled in by hand. However, this technology is very limited in terms of how much content can really be customised.

The volume of data that is gathered about individual consumers is enabling a move from pull-based marketing – where customers asks for information about specific products – to push-based marketing: Organisations proactively contacting potential and existing customers to advertise products they are expected to be interested in, based on what is already known about their buying habits. Online services such as the Amazon.com recommendations engine, which recommends products based on a user’s past purchases and feedback [1], show how an automated system can target information to individual customers. Users are presented with a list of product images and descriptions, and can choose to have a recommendation of the day emailed to them.

At present the layout options available for serving customised content (and for the related problem of adapting web pages to display neatly in different sized windows) are not very flexible. For the most part, systems simply work by fitting items into boxes in a template, keeping the page tidy by pre-setting the size and number of images, and constraining the amount of text in an item. If the potential of customised document delivery is to be fulfilled, a technique is required that will achieve a ‘finished’ look without relying on fixing the size of the document contents: a physically self-organising document.

A general-purpose tool to create document layouts, as opposed to one that simply fills a set of templates for a particular class of documents, can not rely on any assumptions about the structure of the information to be printed. This means that the large space of potential layouts must be searched freshly for each new selection of print objects, which can be immensely time-consuming.

It is also difficult to encapsulate the rules used by a human paste-up artist in a form that is useful for the design of a computer system, because they tend not to be explicit. Graphic design textbooks tend to illustrate good design with a selection of clear examples, rather than formally describe the rules or principles to be followed [15], making it hard to craft a function for scoring layouts aesthetically. However, there are some general principles that characterise attractive page layouts, such as increasing visual symmetry, and balancing contents and whitespace evenly around a page.

Section 2.1 examines existing software that partially automates the page layout process, 2.2 describes formally related problems which have been extensively studied, and 2.3 suggests techniques that may be brought to bear on this problem.

2.1. Existing software

A number of systems have been developed to provide assistance with the creation of page layouts, ranging from options in software that do a little tweaking for the user, through to a fully automated layout system, though they all constrain the task in some way.

Perhaps the simplest form of these is the addition of a simple ‘shrink to fit’ function into software: for example Microsoft PowerPoint will automatically reduce the font size of a paragraph if it extends beyond the bottom of a slide, and Microsoft Word has an option called “Shrink to Fit” which tries to find the smallest reduction in font size that reduces the number of pages required to print a document. These programs do not attempt to automatically create a layout, but they do provide users with some assistance that can save time.

There are a number of software products on the market that claim to automatically lay out photo albums or pages of photos, with varying degrees of flexibility. Most (such as [3]) are actually template-based applications, which simply fit pictures into slots in a pre-defined template, cropping or resizing the images to fit the space to which they are allocated. There is usually a degree of flexibility

offered to the user, who can move pictures once they have been placed and sometimes over-ride decisions made by the software.

Some more sophisticated systems generate free layouts. In some cases, the only criterion is saving space by fitting a given number of pictures onto the smallest possible number of pages of expensive photo paper [11], but Kodak have developed two systems which also attempt to take into account aesthetic considerations. The commercially available “Kodak Memory Albums” [13] has an automatic layout option, which produces a photo album from a collection of pictures, assigning each image to a page and placing it on the page. This software only takes image size into account, ignoring any other information about the images, but Kodak also report a prototype system [8] which also allows users to specify how important particular images are (more important ones are displayed larger) and to control the order of pictures. Both of these systems are specific to the domain of photographic images, and they do make domain-specific assumptions: both allow images to be resized considerably to fit, and the prototype system [8] also rotates images and allows partial occlusion, which would make little sense with text.

2.2. Work on related problems

There is a great deal of work in the literature on related layout problems, which are not specific to the print domain but have important similarities.

2.2.1. Fitting websites into variable sized windows

A rarely-used element of the Cascading Style Sheets (CSS) specification [30] allows designers to create “liquid layouts” for web pages, which adapt to changes in the relative size of contents and window with slightly more flexibility than simply changing the word wrapping [18]. It is also possible to use this technology for printed documents, and provide some flexibility, such as constraints to describe where page breaks are acceptable or should be avoided [31]. Even for web pages, this technology does not allow enough flexibility to cope with a wide range of document contents, and they do not work particularly well for paged media such as print documents.

A more flexible approach is being developed by a group at Washington University, based on constraint satisfaction [28]. Their prototype software allows a web designer to specify a complex set of interacting constraints, based on which the browser has to dynamically render a page [2]. This has the advantage of allowing different templates to be selected based on certain conditions (for example choosing a layout with more columns for a wider browser window), and preserving meaningful relationships between parts of the content.

Both the CSS techniques and the constraint-based system are useful to web designers, but they are principally aimed at fitting fixed size contents to a single variable-sized page. For print media the challenge is slightly, but importantly, different: to fit variable sized contents into one or more fixed size pages. This falls into a general class of problems commonly referred to as *bin packing*.

2.2.2. Bin packing problems

‘Bin packing’ [6] describes a general class of problems based on re-arranging a number of items to fit into bins of a fixed size, in order to minimise the wasted space and use the least number of containers. None of the studies listed in this section deal with aesthetic considerations, because they are not relevant to the specific applications that were studied, but they provide useful approaches to fitting content space-efficiently onto a page. There are several types of bin packing problem, which have been applied to different real-world problems:

2.2.2.1. One-dimensional bin packing

In one-dimensional bin packing, there is only one variable that distinguishes between items: each takes up a particular amount of a limited resource, such as space or time. The objective is to distribute items between multiple containers so as to use the space most efficiently, minimising the total number of containers required. One-dimensional bin packing has been used as an analogy for task scheduling problems such as assigning jobs to different workshops or CPUs in order to minimise the idle time of each [5].

2.2.2.2. Two-dimensional bin packing

In two-dimensional bin packing, items must be placed on a flat surface, which is finite in both dimensions, with the aim of minimising the space wasted between items. Although there are some examples of algorithms to reduce the number of fixed-size containers (such as [11] – the photo printing system described in Section 2.1), most applications of this paradigm focus on reducing the size of a single container that must hold all of the items to be packed.

One such example is the ‘stock cutting’ problem, in which a pattern must be found to cut requested shapes from an (to all intents and purposes) endless roll of material, such as paper, steel or cloth [25, 7]. The primitives to be organised can either be rectangular objects that are only amenable to re-ordering (as in [25]), or they can be irregular shapes that may also be rotated (as in [7]). In either case, the optimal arrangement is one that minimises wasted material by using as close to the full width of the material as possible at all times.

Another common application of 2d bin packing is for reducing the footprint, or ‘floorplan area’ of circuits in VLSI manufacturing [27, 22, 12, 16]. In this case, the objective is to minimise the unused spaces between circuit modules, thereby reducing the overall area required to contain the circuit, which could either allow it to be fabricated on a smaller silicon wafer or allow more circuits to fit on one wafer. This is sometimes attempted by re-ordering alone and sometimes by changing the aspect ratios of modules. The aspect ratio can be regarded as continuously variable, but this does not realistically account for the possible ways that a circuit module could be re-arranged, so it is often treated as a discrete variable with a small number of acceptable values [12].

There are actually other important constraints to this problem in the real world, because the position of certain modules is fixed by other design constraints, and modules in the circuit are connected by varying numbers of wires. In order to keep the overall length of wiring required to connect the circuit, modules that are densely interconnected need to be placed close to each other. There is a range of emphasis in different studies, from [27] which only considers the length of wiring required, to [22] which only considers the area of a floorplan.

Wirelength has potential to be a useful concept for document layout generation, by analogy to *relatedness* between items. The same algorithms that group together densely interconnected circuit modules should be capable of grouping together closely related print objects, such as an image and text that relate to the same subject.

2.2.2.3. Three-dimensional bin packing

One and two dimensional bin packing can be regarded as special cases of 3d bin packing, in which it is simply assumed that one or two dimensions are uniform [17]. The paradigm here is that objects have to be fitted into the smallest possible number of finite-size containers, and the most common practical application of 3d bin packing is for the loading of goods into shipping containers [21].

2.2.2.4. Guillotineable packings

In some real world applications, it is important that the solution be *guillotineable*, which means that it can be produced by successive straight cuts all the way across the material. This is important for stock cutting with certain types of cutting equipment [25] and has the advantages of lending itself easily to a variety of representations [29] and reducing the number of possible packings for a given set of items, potentially making the problem computationally less costly [19].

Some commonly used document layouts are guillotineable, such as this report (each section can be separated from the others by a single cut across the page), and any columnar layout that does not allow items to span multiple columns or text to flow around images that partially occlude it.

2.2.2.5. Online bin packing

A further variant of the standard bin packing problem formulation is *online* bin packing, which operates in real time. Rather than having a full list of objects to be packed, the system only has information about open containers, the items packed so far and one new item at a time [5]. This requires a completely different approach, but is not important for page layouts as they can be generated with complete information. Even when a fresh set of items is selected dynamically for

each layout, the layout engine can operate with complete knowledge of the size of the container (the document being produced), and can wait for a complete selection of items to be sent.

2.2.3. Knapsack packing

All of the studies reported in Sections 2.2.1 and 2.2.2, and all of the software in Section 2.1, have assumed that there is a predetermined list of items to be packed. This is useful for creating self-organising documents, but need not necessarily be the case. Given a large database of entries, each of which has been ranked for relevance, and a fixed size document, there are various ways to approach the selection of items. One could rigidly select the first n items in order of relevance, or a more flexible approach could be more productive.

Knapsack packing problems are a general class of problems in which items must be selected to fit into a container of limited size, to maximise the total value of the packed items. This has been applied to many resource allocation problems, such as radio bandwidth [14]. The selection of items to place on a page can be regarded as a knapsack packing problem.

Whether the selection of items is regarded as input to the system, or part of the problem to be solved, good selection is crucial to producing a good layout. In extreme cases, a layout engine can not possibly succeed if it is provided with too many items to fit on a page, and if it either has too few items or a set which are not compatible with each other (such as a column as long as the whole page and a banner as wide as the page, which could not be placed together without overlap) it will not be able to produce good results. Selection can also inform placement in more subtle ways, by flagging objects with different levels of importance, and determining which items are related to which others, and hence should be placed close to each other.

2.3. Techniques

The nature of both bin packing and knapsack packing problems makes an exhaustive search of the solution space highly impractical, because bin packing is NP-hard [17] and knapsack packing is NP-complete [14]. For a page layout engine, the relationship between the number of objects to be placed and the size of the search space is exponential and will be explored in detail in Section 5.

Given the size of the search space, a technique that only searches a portion of it will be more practical, provided that it still finds solutions of a high quality.

2.3.1. Genetic Algorithms

Evolutionary Algorithms [10] are a broad class of algorithms that use “selection acting repeatedly on heritable variation, where that variation is essentially blind, rather than incorporating detailed heuristics” [26]. There are many variants of exactly how to implement these, but the form considered here (Genetic Algorithms, or GAs [9]) is that the algorithm starts with a population of solution candidates, and successively produces new candidates (‘offspring’) by small random variations from the existing population. These potential solutions are then given a score (‘fitness’) in relation to how well they solve the problem being tackled, and the more fit individuals are more likely to be used in producing the next generation.

GAs are often used to tackle NP-hard problems because the amount of compute time required to arrive at a satisfactory solution does not necessarily increase in step with the size of the search space, as is the case for an exhaustive search. The effectiveness or otherwise of a GA depends on the nature of the search space. If the search space has no structure at all – in other words if a small change to a potential solution produces a large change in its fitness – a GA can be expected to perform little better than a random search. GAs work best in a problem space in which solutions with a high fitness are surrounded by a group of similar solutions that lead the search towards them, rather than simply being isolated fitness ‘spikes’ surrounded by poor solutions. In these conditions a GA will not necessarily find the best possible solution, but will find a good solution in a reasonably short time.

Characterising the search space is usually impossible for any problem for which a GA is likely to be a useful approach, because it requires an exhaustive mapping of the fitness of each potential solution, and the intractably large size of the solution space is normally the reason for using a GA. The choice of operators applied to solutions is also of great importance here; appropriate operators must always

produce small changes in the offspring, rather than radically transforming it in a way that loses the structure of the parent.

Because a GA can be stopped at any time in the run, and often finds a reasonably good solution some time before the optimal one, a cut-off time can be specified for each run if the search is not fast enough. GAs of various flavours have been applied to 2d bin packing problems in [7, 23, 16, 22] and to knapsack packing in [32].

2.3.2. 'Hybrid' Genetic Algorithms

Configuring a GA such that it reaches a good solution in a reasonable amount of time is not an exact science; because this must be done without complete knowledge of the search space, it takes a certain amount of educated guesswork, and sometimes the wrong choice of parameters or operators can stop a solution from being discovered at all, by either restricting the search to part of the solution space or introducing too much noise. To reduce the amount of guesswork required, and avoid having to re-invent the wheel, many researchers have used 'hybrid' GAs which combine knowledge-based heuristics with a GA to search the intractable part of the problem space. This approach often produces systems that run faster than 'pure' GAs, though at the cost of no longer being able to generalise to other problems, and it does run the risk of constraining the search space in a way that stops evolutionary search short of finding an adequate solution.

Hybrid GAs have been applied to 2d bin packing problems in various ways. [7] combines an expert system with a GA to solve a stock cutting problem, and [22] adds heuristic operators to a standard GA, resulting in faster solution finding than the standard GA against which the technique is benchmarked.

2.3.3. Non-evolutionary approaches

GA papers often lack useful comparisons between the GA and other algorithms for solving the same problems. In some contexts this may be because it is difficult to conceive of a solution to the problem that does not rely on some sort of evolutionary search [26], but often the omission is not justifiable on those terms. Even when comparisons are provided, authors often fail to compare like with like, as in [22] which compares run times across inconsistent computer systems or [24] which reports a reduction in the number of program cycles required to find a solution without mentioning an order-of-magnitude increase in the computational overhead of one cycle.

To evaluate a GA system usefully, it must be compared with other techniques. This paper will look at exhaustive searching of the problem space, and comparisons between the GA and simulated annealing (which has been used for 2d bin packing by [29]) in Section 5.

2.4. Summary

Section 2 has described the problem of generating customised page layouts. Section 2.1 examined existing software that addresses this problem, Section 2.2 considered related problems that have been more widely studied, and Section 2.3 suggested techniques that might be used to approach this task.

3. The program

A complete page layout system must satisfy a number of related requirements. It must be able to select an appropriate group of print objects from a large collection, and it must then lay out these objects in a way that is space-efficient, aesthetically pleasing and keeps the information content coherent by placing related items together, in an appropriate order.

A proper consideration of the aesthetics of page layouts is beyond the scope of this study, because it would require an extensive investigation of the unwritten rules used by human paste-up artists, or a sophisticated AI system capable of learning by example, as postulated by [15]. The other objectives listed above are made more tractable by tackling one at a time, so this study started with a system to produce space-efficient layouts, which it is hoped will also make a contribution towards the aesthetics of the result, by avoiding yawning expanses of whitespace.

Sections 3.1 & 3.2 describe the method of a paper that I have chosen to replicate as a starting point, and Section 3.3 describes specific details of the program I developed.

3.1. Choice of approach

Because of the close correspondence between VLSI placement and the problem of placing objects on a printed page, Cohoon et al.'s previously published VLSI placement system [4] was used as a starting point. This system used the total area required for a layout and the total length of wiring required to connect the circuit as the measure of solution quality. Minimising the area transfers directly to the print context, and the wiring length will be adapted (as described in Section 6.2) as way of grouping items together.

The problem representation that Cohoon et al. used was set out initially in an earlier paper by Wong & Liu [29], who used simulated annealing to find solutions, and Cohoon et al. [4] adapted this to a GA.

3.2. Method of the original papers

3.2.1. Problem representation

The chief innovation of the Wong & Liu algorithm [29] is their use of normalised Polish postfix expressions to represent a floorplan layout. This method has the advantage of a 1-1 correspondence between expressions and unique layouts, whereas previous work used representations that have a far larger number of expressions; in effect a larger problem space.

The problem is stated as an attempt at producing the floorplan layout with the smallest area and shortest wiring necessary given a list of modules to be placed (equivalent to finding the most space-efficient layout of a given set of print objects). For each module, the area (A), upper and lower bounds for aspect ratio (r and s), and connection strength to each other module (which represents the wiring density between pairs of modules) are pre-specified. Given this information, the system aims to find the best guillotineable packing of these modules.

Any guillotineable packing can be represented as a slicing structure; the series of straight cuts that must be made across the container to produce the desired floorplan. This structure can be represented in turn as a tree [20], or as a Polish postfix expression, with a number to represent each module (or operand), and two operators: "*" to represent a vertical cut, and "+" to represent a horizontal cut. Thus the floorplan in Figure 1:

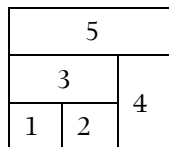


Figure 1: an example slicing floorplan (Figure 8 from [29])

Can be represented by the slicing tree¹ in Figure 2:

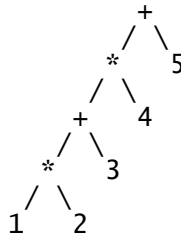


Figure 2: the slicing tree for the floorplan in Figure 1

Or the following Polish expression¹:

1 2 * 3 + 4 * 5 +

In the form described so far, this notation still allows multiple well-formed expressions for one slicing structure, which Wong & Liu claim is a drawback because it increases the size of the search space, and distorts the search by giving some slicing structures many expressions, while others may still only be represented by one [29, p102]. To remedy this, they added an additional constraint – that no consecutive operators may be the same – and called expressions that satisfy this “normalised Polish”. There is only one normalised Polish expression for each slicing structure; in GA terms, there is a one-to-one genotype to phenotype correspondence, whereas if non-normalised Polish expressions are allowed there are multiple genotypes corresponding to one phenotype.

Cohon et al. [4, p453] found that the GA performed better if genotypes were only required to be well-formed Polish expressions, not necessarily *normalised* ones. Following their finding, the system presented here does not restrict its search to normalised Polish expressions, but the concept of normalised Polish expressions is helpful for analysis of the problem space, because it facilitates calculation of the number of possible unique slicing structures.

3.2.2. Solution evaluation

In order to work out a cost for each possible solution, it is necessary to work out the exact position and dimensions of each module, from which the total solution area and the wiring length required to connect modules can be derived.

3.2.2.1. Area computation

The Polish notation described in Section 3.2.1 only specifies the overall structure of slices that must contain a list of modules. To produce an actual layout from this, it is necessary to choose dimensions for each slice such that it is large enough to contain the module that has been assigned to it. If the modules all have fixed dimensions, this is a simple matter of adding up the sizes of the containing rectangles into which each module fits, each of which must be tall and wide enough to contain its module, but for variable aspect ratio modules it is not so straightforward. Different choices for the implementation of each slice can change the overall area of the layout, and the choice for each slice is not independent of the others, as illustrated by Figure 3:

¹ Neither the Polish expression nor the slicing tree specify the width and height of the modules or the slices; they only describe the slicing structure itself. Section 3.2.2.1 explains how the size and proportions of each section of the slicing floorplan are worked out, and Sections 6.1 and 7.2 discuss the issue of how to place the module itself within its slice.

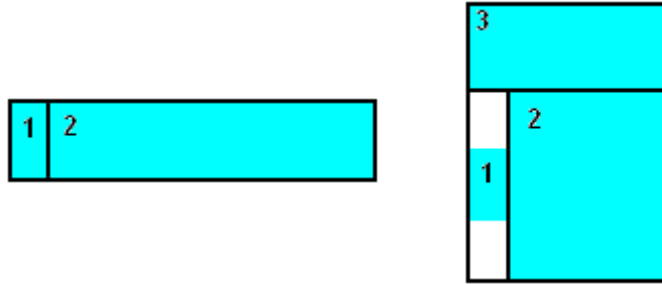


Figure 3: the interdependence of module implementations – both implementations of module 2 are within its aspect ratio constraints, but the choice which is the most efficient option in the diagram on the left, is no longer the most efficient option once module 3 (which is not permitted to stretch any wider) is added, as in the diagram on the right.

Wong & Liu [29, pp103-4] outline a method for dealing with variable aspect ratio modules. For each individual module, a graph can be drawn of width against height, and a line (bounding curve) can be plotted that joins all of the points with the desired area. This line will be a hyperbola, as in Figure 4:

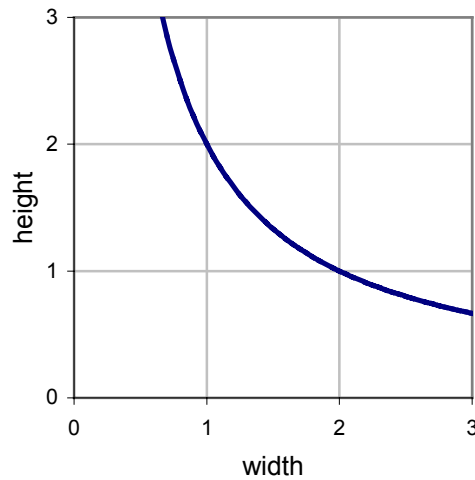


Figure 4: the bounding curve for a module with no constraints on its aspect ratio and a fixed area of 2

In practice, the aspect ratio of a module is normally bounded, which means that the line is only a hyperbola in between the upper and lower aspect ratio limits. Outside these limits, there are vertical and horizontal lines, because increasing one dimension beyond its range will not decrease the other, producing a graph as shown in Figure 5:

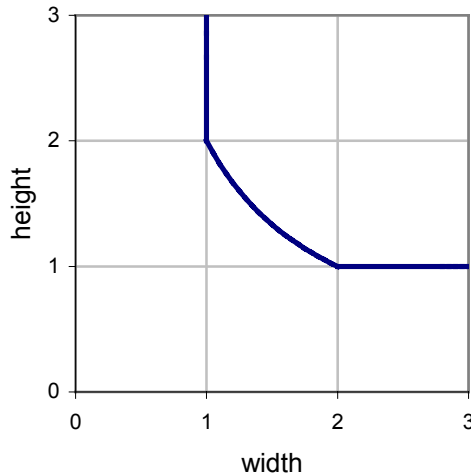


Figure 5: a bounding curve with aspect ratio constraints; in this example the module area is 2, and the aspect ratio may vary from 0.5 to 2.0, allowing dimensions from 2x1 to 1x2

The region above the line represents all of the possible dimensions for a rectangle that will contain the module, and any point on the hyperbolic section will have no wasted space. For any individual module, any point on that curve would produce an optimal packing, but once modules are combined the picture becomes more complicated. The bounding curve for the composite produced by any operator can be derived from the bounding curves of its two operands; if the operator is a vertical cut then the widths must be added and the greater of the two heights used, and vice versa. In practice this becomes very costly to compute, as hyperbolae have to be added to each other, so Wong & Liu follow a precedent from previous work [20] and use a linear approximation of each module’s bounding curve. To derive a first-order approximation, one need only compute the corners at each end of the hyperbolic section, which are the (x,y) co-ordinates for the two allowed extremes of aspect ratio, as shown in Figure 6:

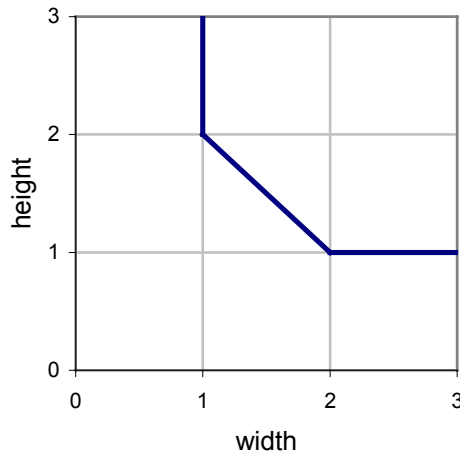


Figure 6: a first order linear approximation of the bounding curve in Figure 5

To work out the bounding curve for a composite module produced by placing two modules next to each other, the bounding curves of the two modules need to be combined appropriately. If linear approximations are used, we only need to add the ‘corners’ of the two curves to produce the composite. If modules ‘a’ and ‘b’ are to be combined, the composite curve is derived from their bounding curves (henceforth Γ_a and Γ_b), as follows (from [29], p103):

For a horizontal slice: $\Gamma_{ab+} = \{(u, v+w) \mid (u, v) \in \Gamma_a \text{ and } (u, w) \in \Gamma_b\}$

For a vertical slice: $\Gamma_{ab*} = \{(u+v, w) \mid (u, w) \in \Gamma_a \text{ and } (v, w) \in \Gamma_b\}$

The example below illustrates this process for the modules set out in Table 1:

Module	Area (A)	Minimum aspect ratio (r)	Maximum aspect ratio (s)
a	2	0.5	2
b	8	0.5	2

Table 1

For these modules to be combined by a horizontal slice ($ab+$ in Polish notation), the corners of the composite bounding curve can be derived as in Table 2:

Corner from b	Corresponding value from a	Composite corner
(2,4)	(2,1)	(2,5)
(4,2)	(4,1)	(4,3)

Table 2

This can be represented by the graphs in Figure 7:

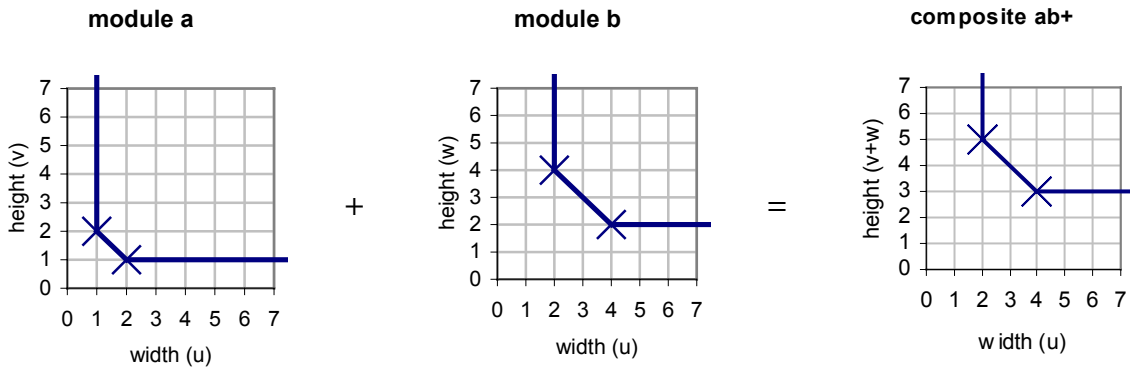


Figure 7: combining two modules

A given Polish expression can be translated into a single composite bounding curve by iteratively replacing each operand with its result, and the best area can easily be found from the composite bounding curve, by simply finding the point on the bounding curve for which the product of x and y is minimum.

3.2.2.2. Wirelength computation

It is not actually necessary to decide where each module will be placed in order to work out the area of a solution, as long as the containing rectangles are big enough. In order to work out the wiring length, the actual positions of the modules must be established, which also means that for each containing rectangle one set of dimensions must be chosen from the corners that have been calculated. This can only be done once the overall floorplan area is known, because the slicing tree must then be parsed again from the root downwards, and each module's dimensions chosen on the basis of the implementation of the composites to which it contributes.

Once this has been done, the result is still not the exact positions of the modules, only the positions, sizes and proportions of their containing rectangles. Wong & Liu [29] stated that they assumed each module was simply centred in its container, and Cohoon et al. followed their lead [4, p454].

Once the positions of the individual modules are known, the wirelength is calculated as the Manhattan distance from the centre of one module to the centre of another, multiplied by the connection strength specified in the dataset.

3.2.2.3. The cost function

Wong & Liu [29] calculated the cost of each solution as:

$$\text{cost} = A + \lambda W$$

Where A is the floorplan area, W is the total wiring length, and λ is a constant that determines how large a contribution wirelength makes to the overall cost. Setting λ to 0 makes the system ignore wirelength entirely, and the higher the value of λ , the less area is emphasised.

Wong & Liu experimented with values of λ from 0 to 3, and found that it does provide some degree of control over how much each term is optimised, but that minimising the area still produces a shorter wirelength than random floorplans, and vice versa, because the two objectives are partially correlated.

3.2.4. Mutation and crossover operators

Wong & Liu used three moves (each selected with equal probability) to change from one Polish expression to another, as follows (from [29] p102):

- M1 Swap two adjacent operands
- M2 Reverse the operators in one chain² of operators
- M3 Swap an operand with an adjacent operator

Note that M1 and M2 can only create well-formed Polish expressions, and will never change a normalised expression into a non-normalised one, whereas M3 can create illegal strings. Therefore the expression produced by any M3 move has to be checked, and if it is not well-formed Polish (or normalised Polish in Wong & Liu's case) the move must be rejected, and another operand-operator pair must be selected to try out.

Cohoon et al. [4] used these three moves as their mutation operators, and added three crossover operators, also each selected with equal probability, which were designed to copy building blocks from the parents, whereas a standard one- or two-point crossover would produce illegal strings more often than not. Their crossover operators are as follows:

- CO1 Copy the operands from one parent into the same positions in the child string, then use the operators from the other parent to fill the gaps
- CO2 Copy the operators from one parent to the same positions in the child, and fill the gaps with the operands from the other parent
- CO3 Copy one parent, randomly select an operator, and copy its operands (both the arguments to that operator itself and any below it in the slicing tree) in the same order, and then copy all of the other operands in the order in which they appear in the other parent³.

Each time a crossover operation is required one of these is chosen, with equal probability. All three of these operators can only create well-formed Polish expressions, and Cohoon et al. were not concerned about *normalised* Polish expressions (see Section 3.2.1), so no further checking was required.

3.2.4. Cohoon et al.'s parallel GA

Cohoon et al. [4] used an unusual sort of genetic algorithm, which they call a 'GAPE' (Genetic Algorithm with Punctuated Equilibria). The concept was devised to make efficient use of a parallel

² A chain of operators is an uninterrupted sequence of 1 or more operators between operands

³ Due to time constraints, CO3 is not implemented in the present version of the GA. This does not seem to have impaired performance, as shown in Section 4.2

processor array, but the actual tests that they report were run with a serial simulation of the parallel system.

Their technique differs from a standard GA in that the population is divided into N subpopulations, and selection and crossover only occur within a subpopulation. Every G generations (the 'epoch length') the populations are mixed, by copying a proportion (the proportion used is unspecified) of genotypes from each to the others. It is hoped that this strategy increases the diversity of the population, decreasing the risk of premature convergence on a local optimum that is far removed from the globally optimal solution.

3.3. Technical details of the program

The present study used a replication of Cohoon et al.'s GAPE [4] as a starting point. For the most part Cohoon et al. described their system in enough detail for precise replication, but they did not specify their selection method.

My implementation uses linear rank selection, which in turn makes the scaling of fitness scores unimportant, as long as the *order* of genotypes in a fitness-ranked list is preserved. Therefore, instead of using the standard deviation based fitness measure that Cohoon et al. used, this program simply multiplies the cost by -1 . The only feature of this measure that is important to the functioning of the system is that it must be *maximised*, whereas cost must be minimised, and this specific measure has the advantage of being faster to compute than one that requires a standard deviation to be calculated.

As with any rank selection system, the difference in selection probability between the best and worst scoring solutions in the population determines the selection pressure, and in this instance it is a tuneable parameter. The following parameters are all easily manipulated by the user:

- Population size
- Number of subpopulations (must be a factor of the population size)
- Epoch length
- Crossover rate (the probability for each reproduction of a crossover taking place)
- Mutation rate (the probability for each reproduction of a mutation happening)
- Selection pressure
- Wirelength weight (as per λ in Section 3.2.2.3.)

There are also twin stopping criteria: a maximum number of generations to run for, and a number of generations after which to stop if the best solution has not improved.

For every run, the program saves data including:

- The population at the start and end of the run
- The genotypes of each individual in every generation in which a new best solution emerged
- The area & wirelength of the best individual discovered in that run
- The number of generations run until that individual first emerged
- The highest fitness score in every generation

3.4. Summary

Sections 3.1 & 3.2 have described the GAPE used by Cohoon et al. [4] to minimise the area and wiring length required for VLSI circuits. Section 3.3 has outlined details specific to the present implementation, which will be used to minimise the whitespace in page layouts.

4. GA Results

This section presents results from tests of my implementation of Cohoon et al.'s GAPE, designed to compare its performance with the implementation in [4] and assess the algorithm's speed. Section 4.1 will describe the system parameters which were used for all subsequent experiments unless otherwise stated, Section 4.2 will compare my implementation directly with Cohoon et al.'s and a random walk on a dataset from [4], and Section 4.3 will summarise the performance of the system on a range of random data.

4.1. Parameter settings

The performance of any GA system is affected by the settings chosen for a large number of parameters, and while it is desirable for a system not to be too brittle in its response to parameter selections, setting them to appropriate values is important. To find appropriate parameter values, preliminary tests were run with a range of randomly generated datasets within the constraints set out in Table 3 (following the example of Cohoon et al. [4] p455), using a range of parameters as listed in Table 4.

Number of modules/objects to be placed	One set for each of 4 to 40
Area of each module (A)	Randomly selected from the uniform distribution [1,20]
Maximum aspect ratio (s)	Randomly selected from the uniform distribution [1,4]
Minimum aspect ratio (r)	1/s
Connection matrix	Each module is connected to a randomly selected set of others with a weight of 1

Table 3: The parameters used to generate random test data

Parameter	Minimum value tested	Maximum value tested
Population size	10	360
Number of subpopulations	1	6
Crossover rate (per genotype)	0.1	1.0
Mutation rate (per genotype)	0.1	1.0
Selection pressure	1	5

Table 4: The range of parameters tested in the preliminary exploration

From this range, the following parameters were found to produce good results after relatively few evaluations: Population size 100, divided into 2 subpopulations, crossover rate 0.8, mutation rate 0.6 and selection pressure 3.

4.2. Comparison with known data

One of the experiments reported by Cohoon et al. [4] is described in enough detail to reconstruct the dataset that they used and repeat the experiments, so as to benchmark the performance of the system presented here against theirs. The dataset is the 16-module structured set, for which every module has a unit area, an aspect ratio fixed at 1, and connection weights such that in the ideal configuration (as shown in Figure 8) each module is connected to each of its neighbours (horizontal and vertical only; no diagonal connections) with a weight of 1. The optimal configuration has an area of 16 and a total wirelength of 48, as shown in Figure 8:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 8: the optimal solution for the structured dataset

The GA was run 100 times using the parameters listed in Section 4.1 and a wirelength weight of 1, as in [4], for 2560 generations, in order to test the same number of evaluations as Cohoon et al. used⁴. Each run lasted approximately 2 minutes on a Pentium-IV based PC running Windows. The average cost of solutions found by the GA presented here was 98.6 (standard deviation: 9.02), which is worse than the 97.9 for the GAPE used by Cohoon et al., but only marginally⁵ so. With a wirelength weight of 1 the cost of the ideal solution is known to be 64.

Additionally a random walk was executed by running the GA with a selection pressure of zero and all other parameters as above (so that mutation and crossover still operated as normal, but there was no selection at all). The mean cost of solutions found by the random walk was 142 (standard deviation 10.2). The GA performs significantly better than a random walk ($p < 0.0005$)⁶.

In many of the GA runs, the best solution was found close to the end, hinting that the limit of 2560 generations may be stopping the GA before it has finished optimising. To test this, another batch of 100 tests were run with the same parameters, except that the stopping criterion was changed to continue until 1000 generations had passed without improving on the best fitness. In these conditions, the average cost of solutions found was 92.3 (standard deviation 4.33), and the average run length was 3570 generations (357,000 evaluations). The standard deviation of the run length was 2100, which shows that the number of generations required is far from constant, so choosing a fixed number of generations to run for must waste time in runs that quickly find a good solution, and stop the search early in those that search more gradually. Because this more flexible stopping criterion allowed the GA to find better quality solutions, it was used for subsequent experiments.

4.3. Performance on randomly generated data

4.3.1. A note on measuring the speed with which a GA finds a solution

It can be difficult to compare the speed with which different published algorithms find solutions to the same problem, because there is a range of different, and non-equivalent, ways to report this information. Studies that report the actual run time of their algorithm can not be compared to each other, because computer speeds vary dramatically, and the performance of two versions of the same algorithm can be radically changed by details of the implementation. For this reason it is more useful to report the number of times a standard set of steps have to be repeated. GA results are often reported in terms of the number of generations required to find a solution, but this is also not a useful measure, because the amount of computation required per generation is proportional to the population size.

The clearest way to report the speed of a GA is to report the number of fitness evaluations carried out for each run. This is because the calculation of solution cost will have to be carried out for every potential solution evaluated by any algorithm, so the number of times this operation must be repeated provides a measure that can be meaningfully compared between different algorithms.

⁴ The calculation of the number of evaluations is not explicit in [4], but they detail that the GAPE was run for 16 epochs of 50 generations each, using 4 subpopulations of 80 individuals each, giving $16 * 50 * 4 * 80 = 256000$ evaluations. The GA presented here was run with a population of 100, so 256000 evaluations will require 2560 generations.

⁵ It was not possible to test the significance of this difference, because [4] only lists the mean cost, and gives no information about the distribution of results.

⁶ $t = 10.61$ in a paired t test.

4.3.2. Results from random data

With the parameters as set out in sections 4.1 and 4.2, the GA was performance-tested using a collection of datasets, randomly generated using the ranges of values for A , r and s as described in Table 3. The wirelength weighting for all of these tests was set to zero, making them purely a test of how compact a layout the algorithm could find. The effect of wirelength will be revisited in Section 6.2. The average performance of 10 runs for each dataset is summarised by the graphs below, and listed in full in Appendix A. Figure 9 shows the mean number of evaluations taken for the GA to find a solution for each dataset size, and Figure 10 shows the proportion of the best layout for each dataset that was taken up by whitespace:

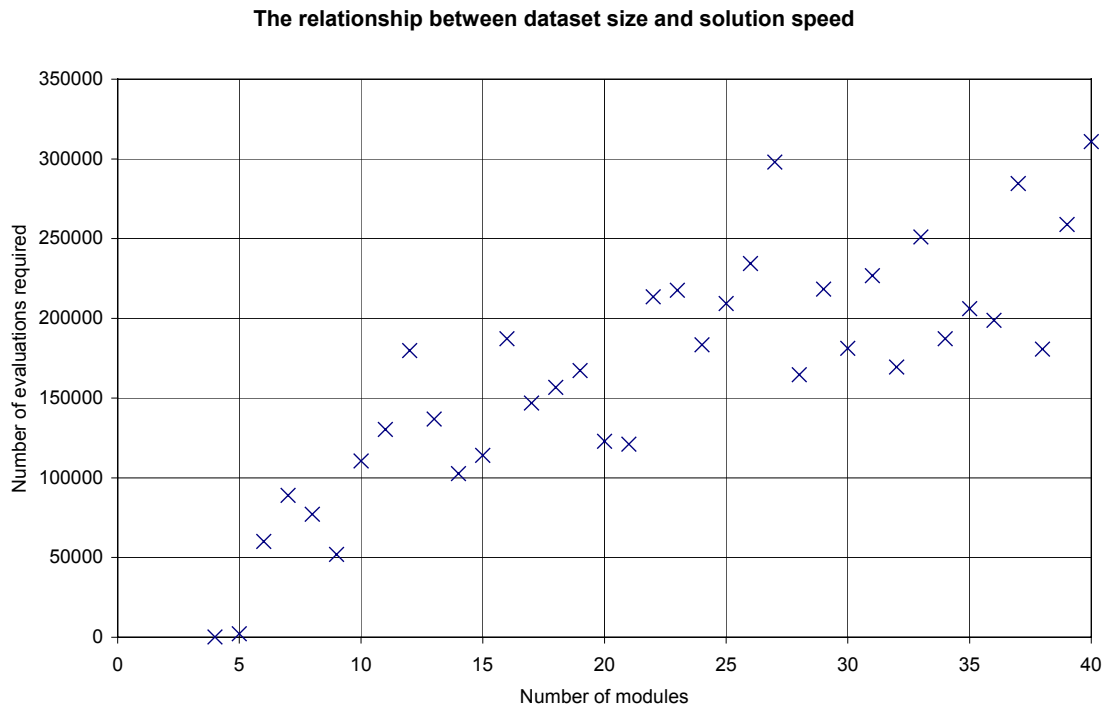


Figure 9: the speed of the GA

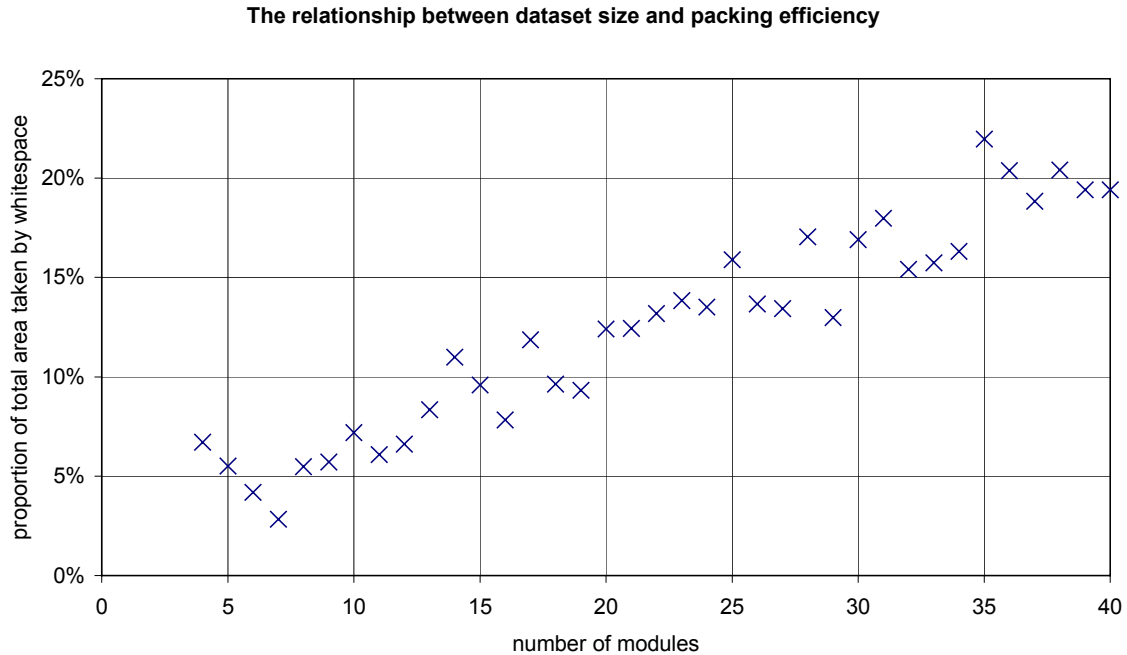


Figure 10: the packing efficiency of solutions found by the GA

The first two data points on both of the graphs above are special cases, because for 4 and 5 modules the GA consistently found global optima (as corroborated by exhaustive search). The global optima for 6 and 7 modules were found by some, but not all, GA runs, and for more than 7 modules the global optima are not known (for reasons that will be explained in Section 5).

Figure 9 clearly shows that the number of evaluations run (using the ‘end when there has been no improvement after 1000 generations’ stopping criterion) increases with the number of modules to be placed, as to be expected given that the search space increases in size as well. However the increase in the number of evaluations required appears to be close to linear, as opposed to the exponential growth of the search space itself. This will be considered in more depth in Section 5.

Figure 10 shows that the packing efficiency of the best solution found decreases as the number of modules to be placed increases. The impracticality of computing global optima for this data makes it impossible to be certain whether this is due to the optimal packing being less efficient for more modules, or to the GA finding solutions that fall further short of optimal. That the GA is finding suboptimal solutions is a more likely explanation, because the variance in the area of solutions found increases linearly with the number of modules to be placed, as illustrated by Figure 11:

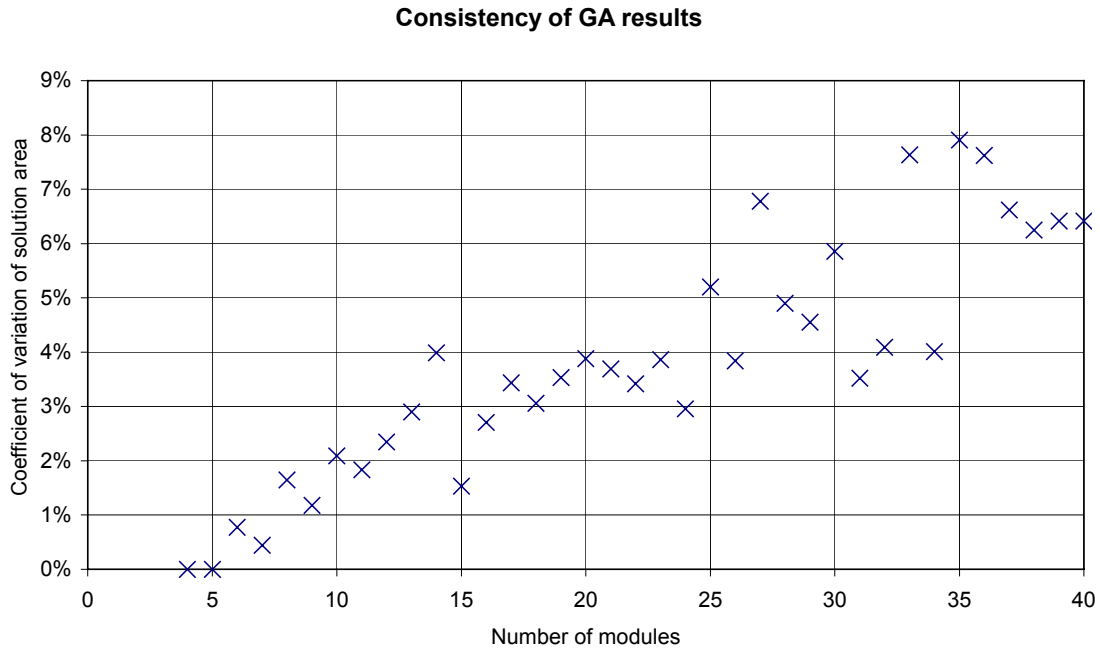


Figure 11: The increase in variance of GA solution quality with the number of modules to be placed

The increasing variance of solutions found for increasing numbers of modules implies that the mean area must be diverging further from the global optimum. However, it is not important that a GA necessarily find *optimal* solutions, as long as it finds *good* solutions. Section 6 will consider the aesthetic quality of layouts found by this GA.

4.4. Summary

Section 4 listed the parameter settings used for my implementation of the GAPE, described the test datasets, and considered the speed of the GA. It was found to perform very similarly to Cohoon et al.'s original implementation [4] on the data that could be replicated, in spite of only incorporating two of their three crossover operators, and considerably better than a random walk. For random data the relationship between dataset size and time to find a solution is close to linear, as contrasted with the exponential growth in the search space. The layouts produced by the GA become less parsimonious as the dataset increases in size, but Section 6 will consider whether the aesthetic quality of layouts decreases unacceptably.

5. The size of the search space

As reported in Section 2.3, bin packing problems in general are NP-hard, so the search space can be expected to grow exponentially as the number of items to be placed increases linearly. The characteristics of the search space for the specific problem of finding guillotineable layouts for n objects are as follows:

- n objects require $n-1$ cuts, or operators in Polish postfix notation, so the Polish expression to describe the layout of n objects has $2n-1$ characters.
- Each of those characters can be one of n module numbers, giving $n+2$ possible values in total.
- Therefore the total number of possible permutations of the character string is $(n+2)^{2n-1}$. Of these, the majority will not be well-formed Polish expressions.
- The number of well-formed Polish expressions (P) can be derived from the series⁷ $P_{i+1}=4P_i(2i-1)$, where $P_2 = 4$.

An exhaustive search program was written to find global optima for those datasets that were searchable in the amount of time available, and to confirm the explosive growth of the search space as the number of objects to be placed increases. The exhaustive searcher has to step through many of the lexical permutations, but a proportion of these were cut out by some simple heuristics. Because the first and last well-formed Polish expressions that will be checked are easy to work out (for example, the search will run from 0123*** to 32+1+0+ for $n=4$), only the permutations from one to the other need be examined, cutting out a significant number of evaluations, as shown in Figure 12:

⁷ This series was derived empirically, and it is a perfect fit to the known data for 2-7 objects.

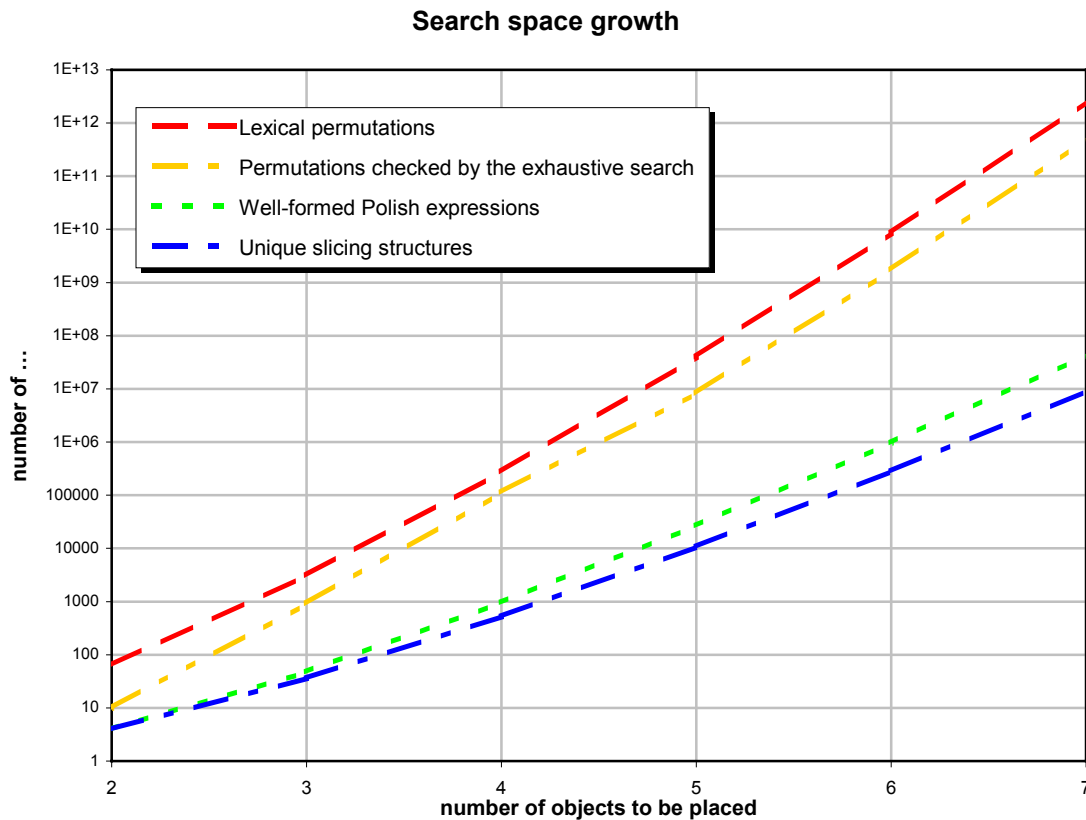


Figure 12: the combinatorial explosion in the search space

As Figure 12 shows, however, the search space is still very large. Running the exhaustive search for 7 objects took approximately 9 hours on a 1.8 GHz Pentium IV, and because the number of strings checked by the exhaustive search increases by more than two orders of magnitude when the problem is expanded by 1 object, it is clear that exhaustively searching the solution space for more than 7 objects is not practical.

The number of evaluations required for the GA to reach a high quality solution (equal to the global optimum for up to 6 objects) does not increase exponentially. For very small numbers of objects ($n < 7$) the GA is not much more efficient than the exhaustive search, but time required increases at a far lower rate than the size of the search space, so for larger numbers of objects ($n > 6$) the GA is considerably more efficient. The number of evaluations used by the GA is compared to the size of the search space in Figure 13:

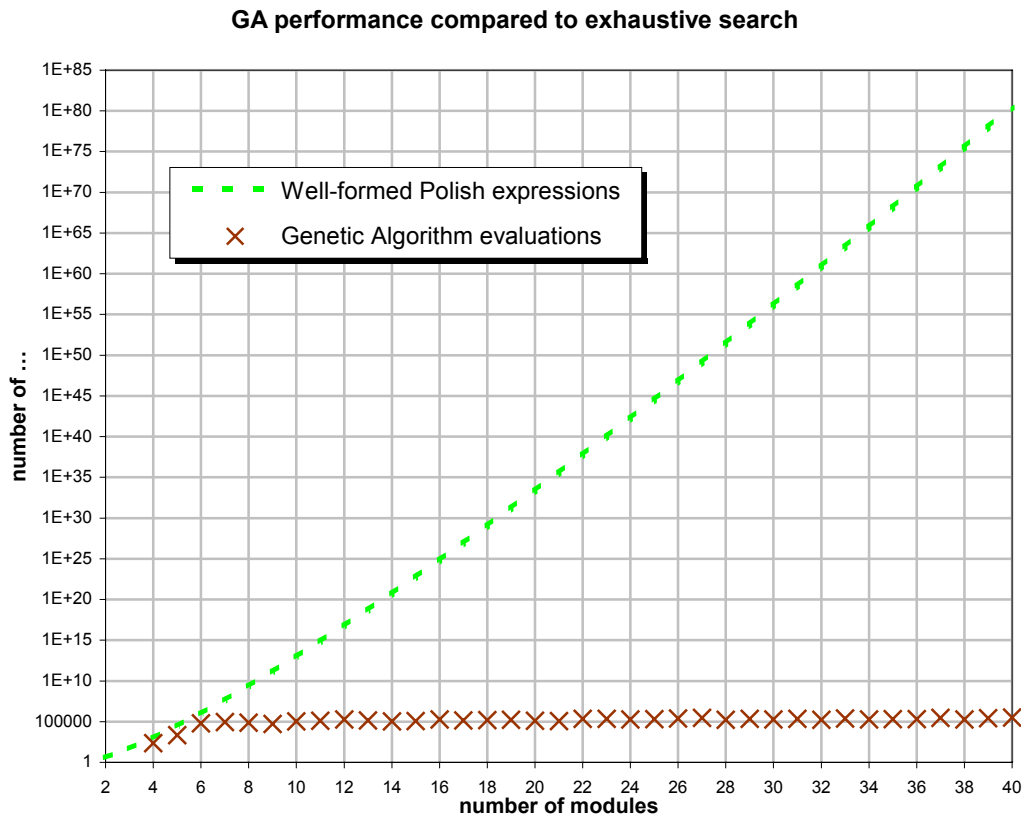


Figure 13: the performance of the GA relative to the size of the search space⁸

Figure 13 clearly demonstrates that the GA continues to be a practical method for tackling this problem with dataset sizes for which an exhaustive search is intractable. On the other hand, the imprecise stopping criterion adds an overhead that makes the GA slower than exhaustive search for the very smallest problems, where $n < 6$.

⁸ Note that this graph plots the number of evaluations taken to find the eventual best solution for each run. The GA actually runs for 1000 generations after finding its optimum, because there is no theoretical way for it to check that it has finished searching.

6. Evaluating the GA in the print context

The GA described in Sections 3-5 is a close replication of a system for producing VLSI floorplans, chosen (as explained in Section 3.1) for its close analogy to the problem of laying out documents. The remainder of this report will consider the aesthetic quality of layouts produced, and consider extensions to the GA with specific relevance to producing print layouts. Section 6.1 will present and discuss images of the layouts produced by the GA, Section 6.2 will consider ways of making layouts conform to aspect ratio constraints, and Section 6.3 will describe an attempt at grouping items together.

The images used in this section all have their aspect ratios preserved, but the graphics are not to the same scale as each other. The position of objects in these layouts is also not entirely specified by the GA; as explained in Section 3.2.2, only the details of the rectangle that *contains* each object are known. The graphics were all produced on the assumption that each object will have the aspect ratio closest to that of its containing rectangle among those possible for its position and constraints, and will be centred in its container. Section 7 will consider how more sophisticated placement strategies could improve results.

6.1. Aesthetic quality of layouts produced by the GA

As discussed in Section 4.3, the proportion of whitespace in layouts found by the GA increases as the number of objects to be placed is increased (Figure 10), and it seems likely that this is due to the GA converging on suboptimal packings. However, it is not necessarily important for the GA to find the most space-efficient layout for a given set of objects, as long as the result is a good quality layout. This is a subjective criterion that can not be judged by looking at measures of area alone, so this section will present images of the actual layouts produced.

Figure 14 is a globally optimal⁹ layout for the 7 module dataset, which is found by some but not all GA runs:

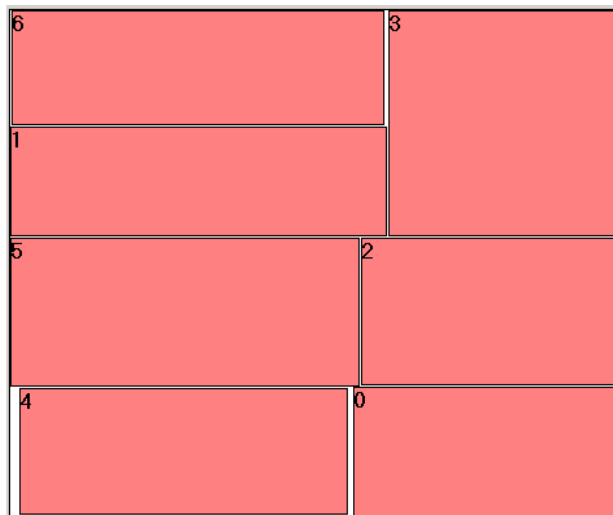


Figure 14: The most parsimonious layout possible for the 7 module dataset

This is the dataset for which the GA finds layouts with the least whitespace. It is not necessarily ideal to lay a document out with this small an amount of whitespace – certainly if the 7 objects were sections of text some space would have to be inserted to make the document readable – but by removing large clumps of whitespace the GA has produced a layout that would be easy to ‘finish’ automatically. The merits of possible strategies for adding whitespace will be discussed in Section 7, but it is clear from Figure 14 that a very simple strategy could distribute whitespace evenly between

⁹ It is never the case that there is only one globally optimal layout for a given dataset, because there will always be at least one pair of modules (the first two operands in the Polish expression) that can be swapped without changing the area. In this example, switching module 1 with module 6, module 5 with module 2, or module 4 with module 0 will not change the area.

all of the objects with good results. In contrast, there is no way to distribute whitespace evenly for the randomly generated layout in Figure 15 (below) without re-ordering the objects themselves:

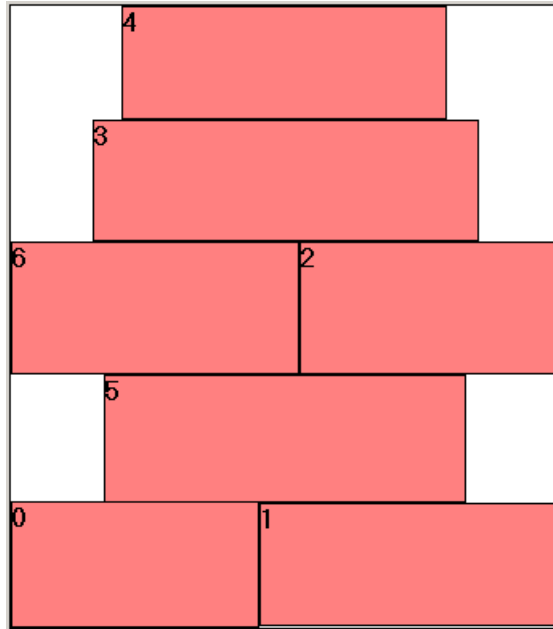


Figure 15: The most parsimonious of the 100 randomly generated layouts for the first generation of a GA run, for the same 7 module dataset as Figure 14

The large clumps of whitespace in this unoptimised layout would make it impossible to use the same simple algorithm to distribute the space evenly around a printed page as would work for Figure 14.

For the GA to be useful, it is important that it can find appealing layouts for a range of datasets, not just the simplest cases. Figure 16 shows results for 18 and 24 objects:

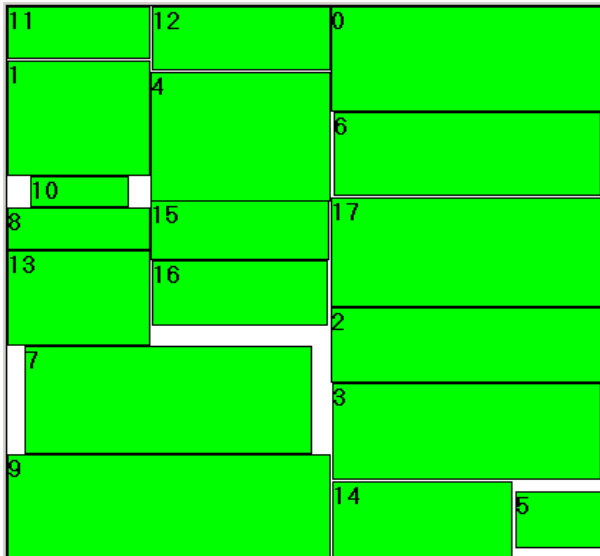


Figure 16a: a GA-generated layout for the 18 module dataset

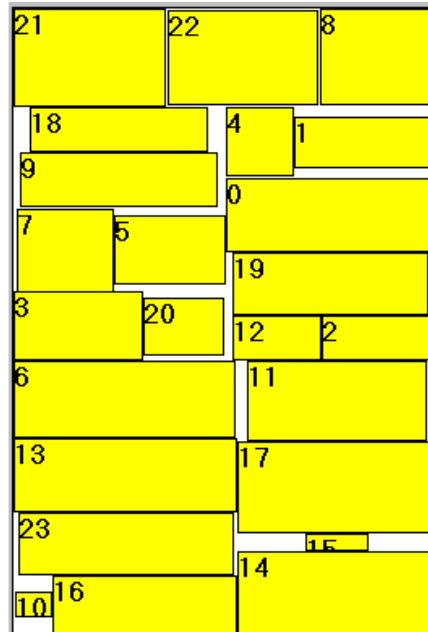


Figure 16b: a GA-generated layout for the 24 module dataset

Although these layouts are not as parsimonious as that for the 7 module instance, the GA has still managed to produce visually appealing layouts in both of these cases: It is striking that in both of

these cases a roughly columnar layout has appeared, but this is not specified anywhere as a goal of the GA. The GA simply optimises a cost function that seeks to minimise the total area required for the layout, and this arrangement into columns has emerged as a by-product of the algorithm. Columnar layouts are not necessarily produced by this GA, but very often the layout produced will either be divided into columns or rows (and Section 7.1 will consider the factors that determine whether rows or columns are likely), as in the layout shown in Figure 17, which is the result of another run of the GA on the same 18 module dataset as depicted in Figure 16b:

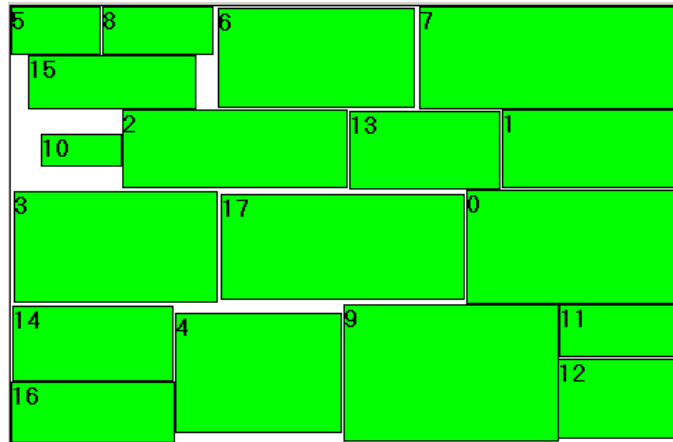


Figure 17: a second GA-generated layout for the same 18 module dataset as depicted in Figure 16b

As explained in Section 2.2.2.4, many common layout styles, including multiple-column pages, are guillotineable. The GA presented here looks exclusively at slicing layouts, which by definition are also guillotineable, which seems to have the side-effect of biasing the search towards layouts with multiple straight cuts across the page. This is convenient because such layouts are visually more coherent than layouts with less visible structure.

The GA does not, however, produce infallibly good results. The following two examples will show conditions in which it fails to produce visually pleasing layouts. Figure 18 shows the most parsimonious layout produced for the 35 module dataset, which was the dataset for which the GA found the least parsimonious layouts (as shown by Figure 10):

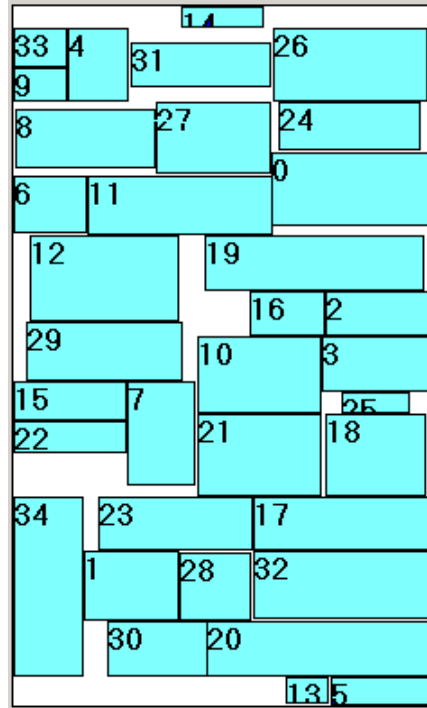


Figure 18: the best layout for the 35 module dataset

As Figures 10 & 11 in Section 4.3.2 show, the performance of the GA, both in terms of the space-efficiency of the best layout found, and the consistency with which it finds good layouts, decreases as the number of objects to be placed increases. A simple solution to this could be to divide both the dataset and the page, letting the GA lay out each (for example) half page with half of the objects.

However, it is also possible for the GA to produce ugly layouts for simpler datasets as well, because the cost function is an imperfect measure of aesthetic appeal. Figure 19 shows an example layout for 12 objects, which has a low cost because it has little whitespace, but looks unappealing because the GA has found a solution with an extreme aspect ratio:

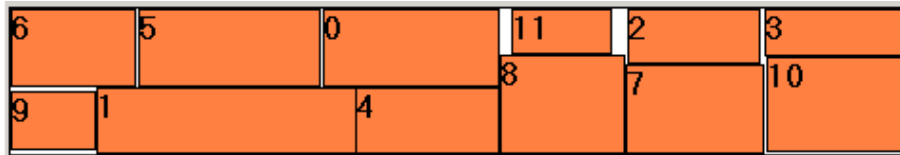


Figure 19: an efficient but unattractive layout for the 12 module dataset

Aspect ratio is not only a contributing factor in aesthetic appeal, it is also often a known constant for print layouts, so Section 6.2 considers ways of controlling the aspect ratio of layouts produced by the GA.

6.2. Controlling the aspect ratio

Because VLSI floorplans typically do not have to be restricted to a precise aspect ratio, Cohoon et al. [4] allowed the aspect ratio of their floorplans to vary between user-set extremes, usually 0.5 and 2. The results presented so far in this paper do not feature any aspect ratios constraints at all, but the aspect ratio will almost certainly have to be constrained for any real print layout.

In the context of producing printed pages, the aspect ratio of the page is known in advance, and any significant deviation from this aspect ratio will result in a large, ugly block of wasted space, not to mention extreme cases like the layout in Figure 19. To remedy this, various methods were tested for making the GA find solutions with an aspect ratio close to a value chosen by the user, with varying degrees of success.

6.2.1. Incorporating the aspect ratio into the cost function

As outlined in Section 3.2.2.3, the cost function for the GA is simply $A + \lambda W$, where A is the area of the smallest rectangle required to contain a layout, W is the total wirelength of connections between modules, and λ is a user-specified parameter to balance the two terms. For all of the data illustrated in Section 6.1 λ was set to 0, so in effect the cost of a layout was simply the area (A) of its container.

As a first attempt at constraining the aspect ratio of layouts found by the GA, the cost evaluation was changed to measure the area of the smallest rectangle *of the desired proportions* that could contain the layout produced, as illustrated in Figure 20:

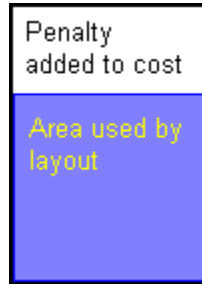


Figure 20: an example of the penalty added to a layout of aspect ratio 1, if the desired aspect ratio was 0.71

This was tested with an aspect ratio of $1/\sqrt{2}$, which is the aspect ratio of the DIN A_n series of paper sizes, in portrait alignment, and a common ratio for printed documents. The results tended to look clumsy, as illustrated in Figure 21:

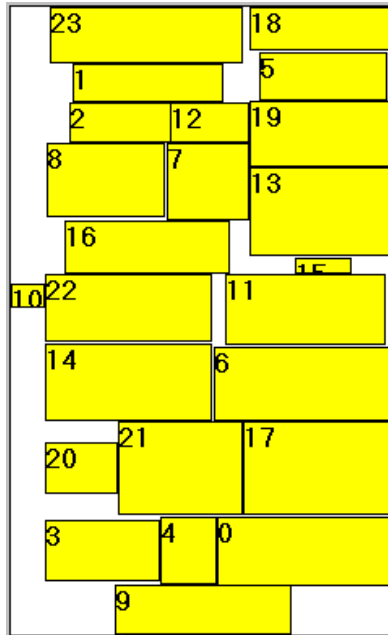


Figure 21: A layout for the same 24 module dataset as in Figure 15b, generated by the GA with the aspect ratio incorporated into the cost function

Not only is this layout ugly and inefficient, but its aspect ratio (0.62) also is not very close to the desired ratio ($1/\sqrt{2} = 0.71$). It is also easy to see a simple change that would reduce the area noticeably – moving module 10 into the space on one side of module 9 – and yet the GA did not discover this change even though it ran for 1000 generations (100,000 evaluations) after first discovering this layout. It is possible that this exact change required a series of mutations to the genotype, but there is certainly an area-reducing change that could be produced by a single mutation – changing the slice between 10 and the rest of the objects from a vertical slice to a horizontal one, which only requires one application of the M2 operator described in Section 3.2.3,

and yet this solution was also not found. This is because although moving module 10 in either way would reduce the solution area, it would not reduce the cost as calculated here, because it would not reduce the size of the smallest container of *aspect ratio 0.71* that could contain the layout; in fact changing the slice from vertical to horizontal would *increase* the cost calculated in this way.

The layout shown in Figure 21 is typical of results with this version of the GA, so it seems likely that the altered cost function has distorted the search space in a way which makes it difficult for the GA to find high quality solutions. Section 6.2.2 describes a slightly different approach to the same problem.

6.2.2. Incorporating the aspect ratio into the area calculation

Instead of simply assigning an area to a layout oblivious to aspect ratio constraints, and then penalising solutions that diverge from the desired aspect ratio, the second attempt at controlling aspect ratio incorporated the aspect ratio constraint directly into the area calculation. As described in Section 3.2.2.1, the computation of area actually involves choosing a point from a bounding curve for the layout, which contains every combination of width and height that can contain the layout as specified. Instead of simply choosing the width & height pair with the lowest product (and therefore smallest area), the algorithm was modified to choose the width & height pair that fit into the smallest container of the desired aspect ratio. The penalty term was removed from the cost function, because this version of the algorithm will always produce a result with the desired aspect ratio, so the penalty term became redundant. Figure 22 illustrates how this version of the algorithm chooses dimensions from a bounding curve:

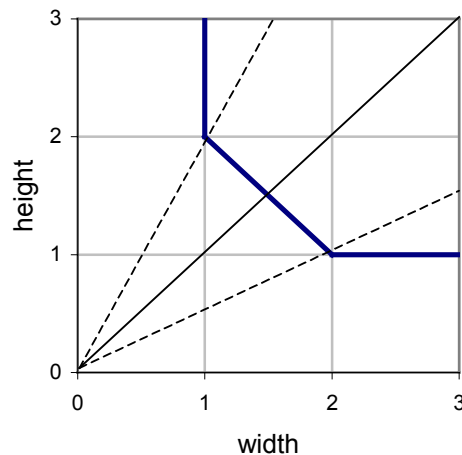


Figure 22: area computation taking aspect ratio constraints into account. If no aspect ratio were specified, the algorithm would choose dimensions along either of the dashed lines, giving an area of 2 and an aspect ratio of 0.5 or 2. If an aspect ratio of 1 is specified, the algorithm must instead choose the point where the straight line crosses the bounding curve, at (1.5, 1.5), giving an area of 2.25

A typical result from this version of the algorithm, using the same 24 module dataset as in Figure 21, is shown in Figure 23:

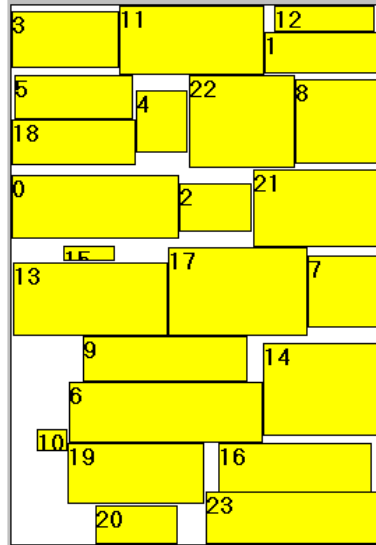


Figure 23: A layout for the same 24 module dataset as in Figures 21 & 16b, this time generated by the GA with the aspect ratio constraint as part of the area computation

Although the results with this version of the algorithm are less parsimonious than with no aspect ratio constraints (as illustrated in Figure 16b), they do represent an improvement over the version described in Section 6.2.1. The aspect ratio is very close to that desired, at 0.70 when the desired ratio was 0.71, and the whitespace is more evenly balanced. This is typical of layouts produced by this modification to the algorithm, but still represents a decrease in quality from the layouts shown in Section 6.2.1, without aspect ratio constraints.

Section 6.2.3 presents a refinement to the aspect ratio constraint, by letting the user specify how strictly the constraint should be applied.

6.2.3. Allowing relaxation of the aspect ratio constraints

Although control over the aspect ratio of layouts is important to keeping the results attractive, this need not entail an absolutely rigid constraint. Documents printed on the same size paper have a variety of margin sizes, and slight unevenness in aspect ratio can be compensated for by adding different amounts of space between objects. Exactly how rigidly the aspect ratio has to be specified will also probably vary from document to document; if a single page document is being prepared it may be allowed more flexibility than if a selection of pages will be bound together, because the pages of a larger document must match each other.

To allow user control over this, two parameters were added to the cost evaluation. A weight could be selected to determine the relative importance of the aspect ratio penalty term, and a range could be specified as a percentage deviation from the ideal aspect ratio for which solutions would not be penalised at all. The new cost function was therefore:

$$\text{cost} = A + \lambda W + kP$$

Where k is the user-specified weight and P is the penalty term, calculated as the difference in area between the layout itself and the smallest rectangle within the specified range of aspect ratios that could contain the layout. In effect, setting k to zero is equivalent to removing the aspect ratio constraint altogether, and setting k to 1 and the allowed range to zero is equivalent to the cost function used in Section 6.2.2.

Experiments with a range of values for these new parameters found that varying the range gives more fine-grained control over the layouts produced than varying the weight, as illustrated by Figure 24:

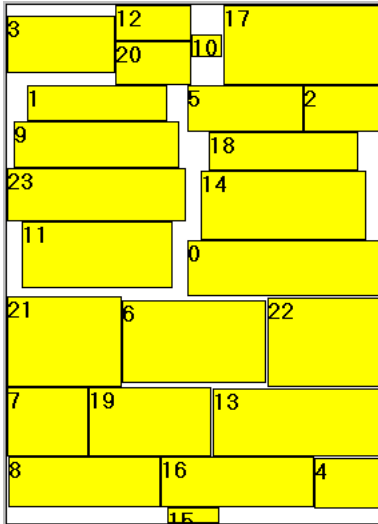


Figure 24a: a layout produced with an aspect ratio weighting of 0.5, and an actual aspect ratio of 0.72

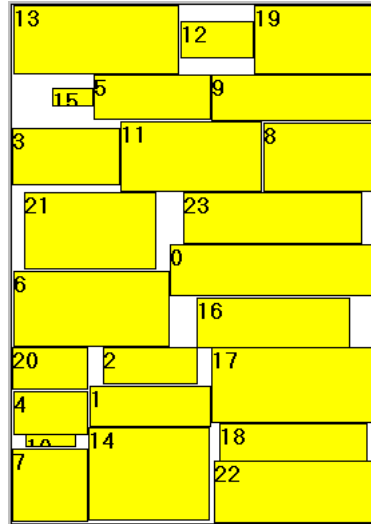


Figure 24b: a layout produced with an aspect ratio range of $\pm 10\%$, and an actual aspect ratio of 0.71

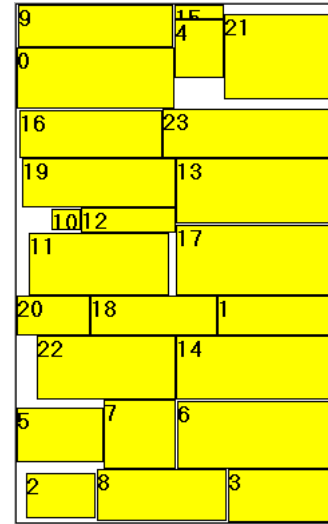


Figure 24c: a layout produced with an aspect ratio range of $\pm 20\%$, and an actual aspect ratio of 0.62

As these examples show, there is a trade-off between achieving tight control over aspect ratio and producing a parsimonious layout. This gives a user of the system a helpful level of control over the sort of layouts produced, by allowing them to choose exactly how to balance the compromise between these two objectives.

6.3. Grouping items together

So far the GA has simply treated each print object as a box, regardless of its contents. In a real print context, it is highly unlikely that simply placing items in an arbitrary order will produce useful output, for instance in a catalogue there is often a picture, a tagline and a body of text all describing the same product, which must all be placed together to make sense. The concept of wirelength from the VLSI context (as described in Section 3.2.1) was adapted to organise items into groups, as outlined in this section.

In the original VLSI placement algorithms reported by Wong & Liu [24] and Cohoon et al. [25], modules could be connected to each other by variable strength connections, to represent the wiring between them, but the layouts produced so far have simply been produced with λ (the wirelength weight) set to zero, thus ignoring any connections between modules. Group membership was indicated by setting the connection strength between members of a group to 1, and the connection strength between modules in different groups to zero. Setting λ in the cost function could therefore determine the relative importance of keeping groups together and of minimising layout area.

6.3.1. Division into two groups

This was first tested with the 20 module dataset, divided into 2 groups, with modules 0-9 assigned to one group, and 10-19 to the other. Figure 25 compares the effect of setting λ (which can now be considered the *grouping weight*) to zero or 1:

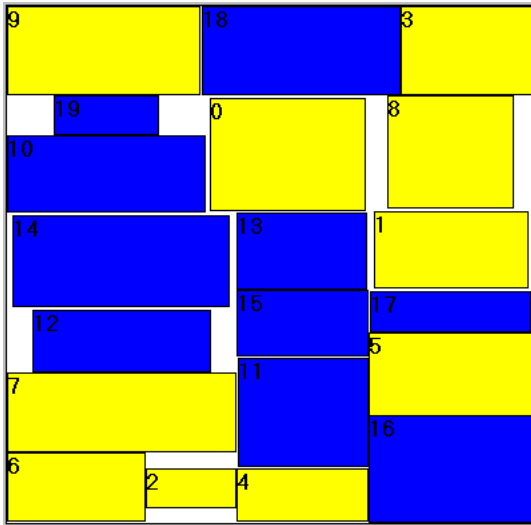


Figure 25a: the 20 module dataset with $\lambda=0$ and area 259.9

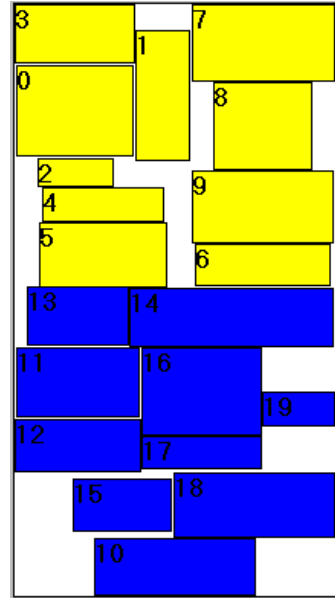


Figure 25b: the same 20 module dataset with $\lambda=1$ and area 313.7

These settings for λ represent extremes either side of a desirable balance. With the grouping weight set to 0 it is not surprising that there is no evidence of grouping – any that did appear would be solely coincidental – and with a weight of 1 the grouping is perfect, but the layout substantially less efficient. A range of values for λ between 0 and 1 was tested, and 0.2 was found to provide the best results, with reliable grouping and efficient packing; a typical layout is shown in Figure 26:

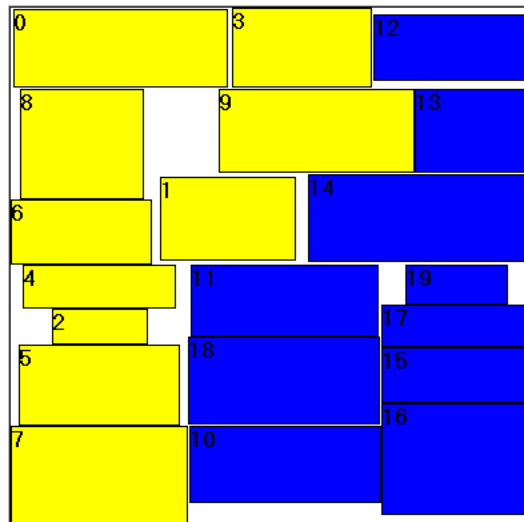


Figure 26: the same 20 module dataset as in Figure 25, with $\lambda=0.2$ and area 267.6

6.3.2. Division into more than two groups

The same paradigm was used to divide the 21 module dataset into three groups (0-6, 7-13 and 14-20). This attempt was unsuccessful however, as it proved impossible to find a weighting that balanced the requirements of both grouping and area reduction. As Figure 27 illustrates, even with $\lambda=1$ the grouping is imperfect, and the packing is highly inefficient when grouping is given that high a weighting:

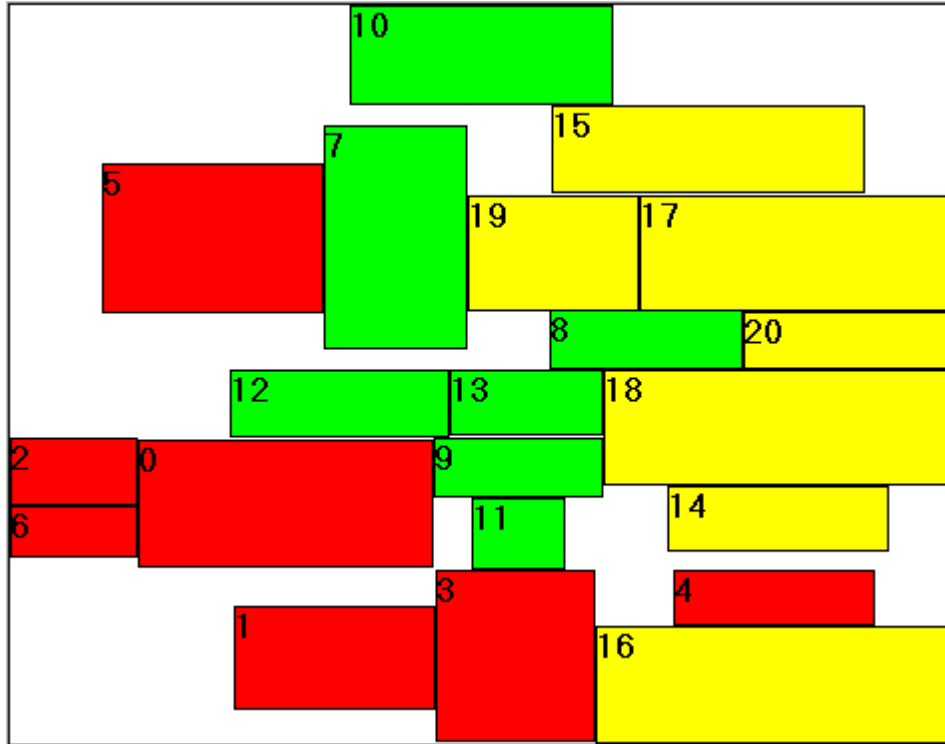


Figure 27: with 3 groups it becomes impossible to achieve reliable grouping without sacrificing layout efficiency

Although automatic grouping in this way fails for more than 2 groups, it is worth remembering that the GA performs well with small datasets. Therefore grouping could also be achieved by dividing the page *a priori* into regions, and then applying the GA separately to the objects to be placed in each region. This has the drawback (compared with automatic grouping) of not allowing flexible division of the page (as in the example of Figure 26), but it will generalise to larger numbers of groups.

6.4. Summary

Section 6 has presented visual representations of the layouts produced by the GA, to demonstrate that the simple heuristic of favouring layouts with minimal whitespace does tend to produce aesthetically pleasing layouts. It also described two extensions to the basic algorithm, to allow the user to control the aspect ratio of generated layouts, and to automatically divide the layout into two groups. An attempt to divide the layout into more than two groups was also described, but this was unsuccessful.

7. Discussion & further extensions

This section will consider the conditions under which the GA described above can be expected to be a useful layout tool, and the further work that will be required to make this into a complete layout generation system.

7.1. Constraints on the selection of objects

Although this study has looked at layout generation in isolation, selection and layout of print objects are not independent problems. As suggested in Section 2.2.3, the nature of the objects that have been selected will impact upon the quality of layouts that can be generated. All of the test data illustrated in Section 6 had similar random ranges of area and aspect ratio, but experience with this random data does provide some insight into the sort of objects that should be provided for the layout engine to work well. Sets of objects that fit the following criteria are likely to be laid out more attractively by this GA than sets that do not:

- Between 10 and 30 items, because with too many the GA becomes less good at finding parsimonious layouts (see Section 4.3), but with too few even a very space-efficient layout will not look as good as for numbers within this range (as in the 7 module example in Section 6.1)
- If the aspect ratios are fixed then giving several modules the same width will encourage columnar layouts, and giving several modules the same height will encourage layouts in rows
- If the aspect ratios are variable then they should allow enough flexibility for modules to have the possibility of matching each others' widths or heights, as this too will encourage layouts in columns or rows respectively
- The total area of the objects to be placed must be less than the area of the page, because for data that has any irregularities at all the optimal packings still incorporate some whitespace. However, the total area of objects must also not be too small, because then the layout produced would be smaller than the page.

If a large (>30) number of items must be placed, dividing both the page and the dataset into subgroups is likely to yield better results than simply trying to place the whole large group simultaneously. This division could in itself be informed by the requirements listed above; for example if 20 objects out of a set of 40 all have a similar width, they could be placed in one half of the page, thereby increasing the likely quality of the layout.

7.2. Strategies for adding whitespace

Whitespace is an important element of graphic design. As alluded to in Section 6.1, the most attractive looking pages are not the most densely packed, yet the GA as described here aims to minimise the whitespace on the page. This is a productive approach, however, because by minimising whitespace large gaps are generally removed, leaving an even distribution that could be 'finished' by simple heuristics. Once the uneven distribution of whitespace has been removed by the GA, simply adding space distributed evenly between all of the objects will usually produce aesthetically pleasing results.

It is also important to consider the placement of modules within their containers in the slicing structure. So far they have simply been centred, but a more sophisticated heuristic would improve the aesthetics of the page. An effective method would have to consider the balancing of whitespace and the creation of straight lines down and/or across the page, both of which are constraints to which most documents laid out by human paste-up artists conform.

A good strategy for adding whitespace can also help to fit GA-generated layouts to the aspect ratio required. As discussed in Section 6.2, the GA can work with aspect ratio constraints, but it produces far better results if there is some flexibility in the aspect ratio of the layout. If a layout diverges slightly from its desired aspect ratio this could be fixed by adding different amounts of space across the two dimensions; for example if a layout is slightly narrower than the page more space could be added horizontally between objects than vertically.

7.3. Taking account of content

The GA as described so far does not take account of what is actually contained within each print object, simply treating each item as a ‘black box’ with certain constraints on its shape. The system would be improved by taking the actual content of each object into account.

A simple consideration would be that changing the aspect ratio of images seldom makes sense, because although it might improve the overall appearance of the page it will usually make the image itself look odd. Instead, upper and lower bounds for the *area* of images could be set, with the aspect ratio being fixed.

For a more complex extension, word wrapping could be worked out for objects that contain text. The heuristic used thus far – that the area of an object must remain constant and its aspect ratio is continuously variable – is only an approximation of what really happens if a text box is resized. The increase in length of a column of text is only roughly proportional to a decrease in width that causes it, because shortening a line will not necessarily cause it to wrap earlier.

For example, Figure 28 plots the length of a sample paragraph of text as a function of its width:

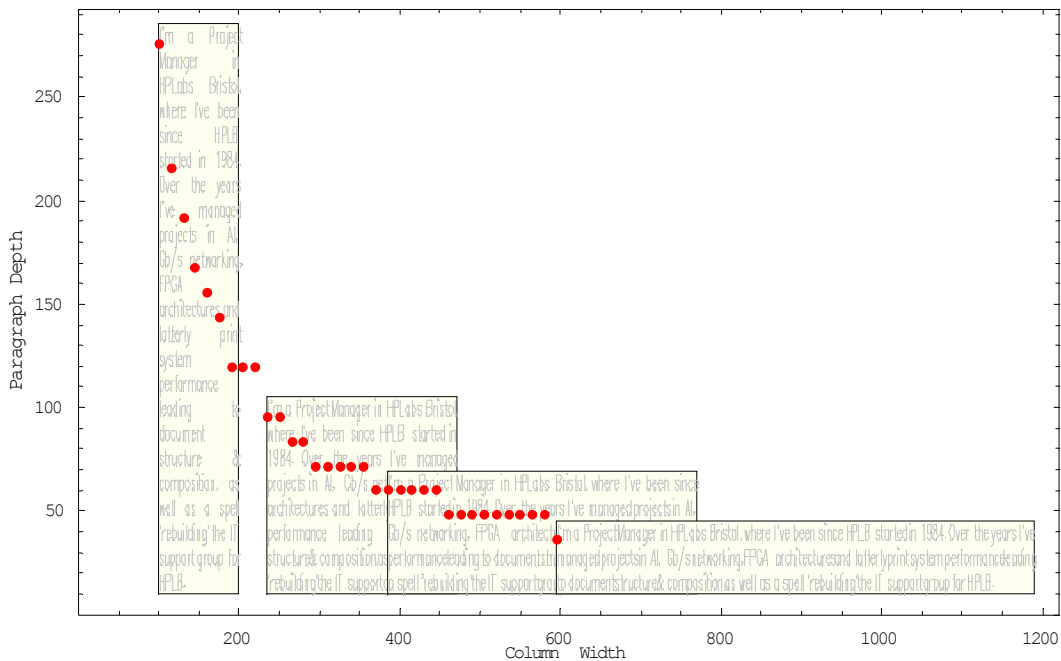


Figure 28: changing the width of a text box does not always increase its height (diagram produced by John Lumley of HP Labs, Bristol)

There is no simple way to model this accurately without processing the actual text within each box, so for accurate results the program would have to be given the contents of each box as part of its input. There is a risk, however, that doing this would slow the system down to the point of uselessness, because it adds so much complexity to each fitness evaluation, so it would be important to monitor the effect of this on system speed. It may well be possible to find an approximation method that is close enough to the detailed case to work, but less computationally costly, just as using a first-order approximation of modules’ bounding curves (as outlined in Section 3.2.2.1) in this work has proved accurate enough to find good results.

7.4. Summary

The GA described in this report is best seen as achieving part of the task of automating document layout. This section has described ways in which the system could be made more complete by adding a method for selection of objects that helps the GA to produce attractive layouts, finishing the layouts attractively by the use of carefully chosen heuristics to add whitespace, and making the data more realistic by considering the actual content of print objects.

8. Conclusion

The work presented here has demonstrated that automatically generating aesthetically pleasing page layouts is possible, and that Genetic Algorithms are a useful approach to this task. The page layout problem has enough in common with the better-studied problem of VLSI floorplanning that techniques can be usefully adapted, but modifications to floorplanning algorithms, as tested here, can increase the utility of the system by taking into account concerns specific to the domain of producing printed layouts.

While the area consideration in VLSI floorplanning is simply to minimise the area of a rectangular container into which a circuit must fit, a page layout algorithm must aim to improve the subjective quality of the layout of a selection of objects on a page of known fixed dimensions. Quality is a subjective assessment, which is difficult to encapsulate in an objective cost function, but the simple heuristic used here has proved able to generate attractive layouts reliably.

The only explicit goal of the GA was to minimise the wasted space between print objects, but as the selection of layouts illustrated in Section 6 show, this is enough to produce aesthetically pleasing output. The relatively even distribution of whitespace, which is in itself desirable, is clearly a direct effect of the requirement to minimise whitespace, but the tendency to arrange items roughly into rows or columns is a more surprising effect. It is clearly a beneficial side-effect, because the arrangement of items into straight lines across a page gives it a degree of visual unity that is attractive, and the self-organisation into rows or columns presents great advantages over an explicit division of the page by increasing the flexibility of the results. The structures that emerge from this layout method, as compared with pre-set rows or columns, are able to adapt to the size of the contents, and the freedom to dynamically change the number of rows or columns (as, for example, in Figure 16) allows a wider range of data to fit than a rigid structure would.

The extensions from a simple whitespace-reducing algorithm bring the GA closer to being a complete page-layout system. While a VLSI designer need only worry about keeping the floorplan within a broad range, a print designer has to produce a layout knowing in advance what size media it will be printed onto. The addition of aspect ratio constraints to the cost evaluation allows a potential user of this system to simply let the program deal with this part of the problem, adding more automation. The ability to arrange items into groups automatically and flexibly, meanwhile, increases the meaningfulness of documents prepared in this way, which could easily be garbled if closely related items are placed far apart. It is significant, however, that the addition of these extra constraints introduces a trade-off between them, because it is highly unlikely that the layout with minimal whitespace will happen to also fit the desired aspect ratio perfectly and provide good semantic grouping. The GA presented above allows the user to specify the relative importance of these three goals, which may well vary according to the specific type of document being produced.

The performance of the GA does deteriorate if there are too many objects to be placed, or if a division into more than two groups is attempted. However, this is easily worked around by simply dividing the page up into smaller areas, each of which contains a small enough number of items in no more than two groups.

The quality of layout produced by this system does seem to be related to the type and quantity of objects being placed. For instance, if a document consists of a small number of items of rigidly fixed dimensions, none of which match each other, any possible layout (whether found by the GA or another method) will inevitably look messy, with an uneven distribution of content and whitespace and no structure. On the other hand, layouts of high quality are produced if the selection of items to be placed fits the broad guidelines in Section 7.1 – having some range of flexibility of aspect ratio, some regularity of dimensions, and a total area which is smaller than that of the page but not too small – and further development as suggested in Sections 7.2 & 7.3 should improve the aesthetics of generated layouts and increase the range of document types for which this system works well.

The hardware required for commercial custom printing already exists, but at present documents must either be designed by skilled professionals or by fitting highly constrained information into templates; the ability to use software to automatically and flexibly generate customised pages will allow such hardware to be used to its full potential.

References

- [1] Amazon.com inc. (2002). *Your Recommendations*. Only available online:
<http://www.amazon.co.uk/exec/obidos/tg/stores/browse/-/help/924452/026-5848469-2310813>
- [2] Borning, A., Lin, R. K-H., & Marriott, K. (2000). *Constraint-Based Document Layout for the Web*. *Multimedia Systems*, **8**:177-189. Available online:
<http://www.cs.washington.edu/research/constraints/web/mmJournal.html>
- [3] Canon Inc. (2002). *Card Photo Printer CP-10*. Only available online:
http://www.canon.co.jp/Imaging/CP10/CP10_3-e.html
- [4] Cohoon, J. P., Hegde, S. U., Martin, W. N. & Richards, S. (1988). *Floorplan design using distributed Genetic Algorithms*. IEEE National Aerospace and Electronics Conference (NAECON 88), Santa Clara CA, pp488-491.
- [5] Csirik, J., Johnson, D. S., Kenyon, C., Shor, P. W. & Weber, R. R. (1999). *A Self Organizing Bin Packing Heuristic*. Workshop on Algorithm Engineering and Experimentation (ALENEX '99), Baltimore, MD, pp246-265. Available online:
<http://citeseer.nj.nec.com/csirik99self.html>
- [6] Dyckhoff, H. (1990). *A typology of cutting and packing problems*. *European Journal of Operational Research*, **44**:145-159.
- [7] Elmaghraby, A. S., Abdelhafiz, E. & Hassan, M. F. (2000). *An intelligent approach to stock cutting optimization*. University of Louisville Multimedia Research Lab, Louisville, KY. Available online:
http://www.louisville.edu/speed/emacs/mrl/publications/pdf/folder/caine00_ehab.pdf
- [8] Geigel, J., & Loui, A. C. (2002). *Automatic page layout using genetic algorithms for electronic albuming*. *Proceedings of The International Society for Optical Engineering* **4311**:79-90. Available online:
<http://www.jogle.com/Research/publications/spieFinal.pdf>
- [9] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, Boston, MA.
- [10] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
- [11] Jayaram, V. (2002). *Photo Paper Saver v1.5.3.7*. Only available online:
<http://www.fpdactor.com/pps/main/main-j-0-0.html>
- [12] Kahng, A. B. (2000). *Classical Floorplanning Harmful?* ACM/IEEE International Symposium on Physical Design, San Diego, CA, pp207-213. Available online:
<http://citeseer.nj.nec.com/kahng00classical.html>
- [13] Kodak Limited. (2001). *Kodak Releases New Software For EasyShare Digital Cameras*. Only available online: <http://www.kodak.co.uk/UK/en/corp/press/archive/2001/20010801.shtml>
- [14] Laakso, J., Jäntti, R., Rinne, M., & Salonaho, O. (1998). *Radio Resource Knapsack Packing For WCDMA Air Interface*. 9th IEEE Symposium on Personal, Indoor and Mobile Radio Communications, **1**: 183-187.
- [15] Lieberman, H. (1995). *The Visual Language of Experts in Graphic Design*. IEEE Symposium on Visual Languages, Darmstadt, Germany. Available online:
<http://lieber.www.media.mit.edu/people/lieber/Lieberary/Graphic-Design/Expert-Design/>
- [16] Mani, N., & Srinivasan, B. (1997). *Using Genetic Algorithm for Slicing Floorplan Area Optimization in Circuit Design*. IEEE Conference on Systems, Men & Cybernetics, Orlando, FL, **3**:2888-2892.

- [17] Martello, S., Pisinger, D., & Vigo, D. (1997). *The Three-Dimensional Bin Packing Problem*. University of Bologna Technical Report DEIS-OR-97-6. Available online: <http://citeseer.nj.nec.com/133900.html>
- [18] Newhouse, N. (2001). *Practical CSS Layout Tips, Tricks & Techniques*. A List Apart, issue 119. Only available online: <http://www alistapart.com/stories/practicalcss/>
- [19] Otten, R. H. J. M. (1982). *Layout Structures*. IEEE Large Scale Systems Symposium.
- [20] Otten, R. H. J. M. (1983). *Efficient Floorplan Optimization*. International Conference on Computer Design.
- [21] Pimpawat, C., & Chaiyaratana, N. (2001). *Using a co-operative co-evolutionary genetic algorithm to solve a three-dimensional container loading problem*. Congress on Evolutionary Computation, Seoul, Republic of Korea, pp1197-1204. Available online: <http://rcis.kmitnb.ac.th/publication/CEC2001%20-%20Container%20Loading.pdf>
- [22] Rebaudengo, M., & Sonza Reorda, M. (1996). *GALLO: A Genetic Algorithm for Floorplan Area Optimization*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **15**:943-951. Available online: <http://www.cad.polito.it/FullDB/exact/gallo.html>
- [23] Sandnes, F. E., & Megson G. M. (1996). *A Hybrid Genetic Algorithm Applied to Automatic Parallel Controller Code Generation*. European Workshop on Real-Time Systems (EURWRTS'96), L'Aquila, Italy.
- [24] Smith, J. E. & Fogarty, T. C. (1996). *Self adaptation of mutation rates in a steady state genetic algorithm*. IEEE Conference on Evolutionary Computation, New York City, NY, pp318-323.
- [25] Tarasova, T. D., & Rozanova, L. F. (2000). *Methods of Calculating Rational Using of Material Resources*. Workshop on Computer Science and Information Technologies. Available online: http://csit.ugatu.ac.ru/pdf_ed/v2_113-115.pdf
- [26] Thompson, A. (2002). *Notes on design through artificial evolution: Opportunities and algorithms*. pp17-26 in: Parmee, I. C. (ed.). *Adaptive Computing in design and manufacture (V)*. Springer-Verlag, Heidelberg, Germany. Available online: <http://www.cogs.susx.ac.uk/users/adrianth/acdm2002/paper.pdf>
- [27] Kong Tianming, Hong Xianlong, & Qiao Changge. (1997). *VEAP: Global Optimization based Efficient Algorithm for VLSI Placement*. Asia and South Pacific Design Automation Conference (ASP-DAC'97), Chiba, Japan, pp277-280.
- [28] University of Washington Constraints Research Group. (2002). *UW Constraint-Based Systems*. Only available online: <http://www.cs.washington.edu/research/constraints/>
- [29] Wong, D. F., & Liu, C. L. (1986). *A New Algorithm for Floorplan Design*. 23rd IEEE Design Automation Conference, Las Vegas, NV, pp101-107.
- [30] World Wide Web Consortium. (1998). *Cascading Style Sheets, level 2*. Only available online: <http://www.w3.org/TR/REC-CSS2/>
- [31] World Wide Web Consortium. (1998). *Paged media*. Only available online: <http://www.w3.org/TR/REC-CSS2/page.html>
- [32] Zitzler, E., & Thiele, L. (1998). *Multiobjective Optimization Using Evolutionary Algorithms: A Comparative Case Study*. pp 292-301 in: Eiben, A. E., Bäck, T., Schoenhauer, M., & Schwefel, H.-P. (eds.). *Parallel Problem Solving from Nature*. Springer-Verlag, Berlin, Germany. Downloadable from: <http://citeseer.nj.nec.com/zitzler98multiobjective.html>

Appendix A: Table of results

The performance of the GA on random test data:

Number of modules	Mean area of solution	Standard deviation of solution area	Total area of modules	Percentage whitespace	Mean number of evaluations to find solution	Standard deviation of run length
4	34.36	0*	32.06	6.702%	230.0	176.7
5	46.41	0*	43.85	5.512%	2180	2348
6	57.06	0.4420	54.68	4.182%	60010	52059
7	91.36	0.4060	88.78	2.826%	89010	69776
8	73.81	1.216	69.77	5.480%	77160	43505
9	95.13	1.121	89.69	5.714%	51990	45424
10	88.32	1.847	81.97	7.193%	110540	109703
11	131.0	2.401	123.1	6.081%	130300	88730
12	102.3	2.401	95.55	6.618%	179890	77841
13	136.9	3.966	125.4	8.337%	136800	66019
14	137.6	5.485	122.5	10.99%	102710	63721
15	188.6	2.893	170.5	9.588%	114070	97993
16	188.9	5.110	174.0	7.835%	187190	188699
17	184.0	6.321	162.1	11.86%	146880	90977
18	202.9	6.208	183.4	9.636%	156790	74680
19	219.8	7.758	199.3	9.324%	167150	96996
20	271.3	10.52	237.6	12.41%	122950	89451
21	226.4	8.363	198.2	12.45%	121210	75714
22	290.5	9.932	252.2	13.19%	213490	192437
23	232.0	8.958	199.9	13.85%	217690	75036
24	304.4	8.991	263.2	13.51%	183530	120337
25	347.3	18.06	292.1	15.90%	209380	147108
26	352.3	13.53	304.2	13.66%	234410	183473
27	352.3	23.89	305.0	13.44%	298010	173599
28	354.4	17.36	293.9	17.05%	164730	112963
29	405.6	18.46	352.9	12.99%	218260	154094
30	419.0	24.53	348.2	16.91%	181270	103595
31	435.5	15.33	357.2	17.97%	226830	105926
32	425.2	17.40	359.7	15.41%	169410	86767
33	432.9	33.04	364.8	15.74%	251130	101700
34	381.4	15.29	319.2	16.31%	187330	70071
35	477.2	37.74	372.4	21.97%	206080	125963
36	464.1	35.38	369.5	20.38%	198850	85289
37	541.9	35.87	439.8	18.83%	284570	173051
38	525.2	32.82	418.0	20.41%	180780	67595
39	503.5	32.29	405.8	19.41%	258840	141277
40	471.5	30.25	380.0	19.41%	310810	99021

Entries marked with a * are those for which the GA is known to consistently find a globally optimal solution. See Section 4.3.2 for a discussion of these results.