GEOMETRIC BIN PACKING ALGORITHM FOR ARBITRARY SHAPES

By

ARFATH PASHA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

I dedicate this thesis to my sister Anberin.

ACKNOWLEDGMENTS

I would like to extend my thanks and appreciation to Dr. Meera Sitharam for her invaluable input into the intricacies of algorithms, for her scholarship in this field, her readiness and ability to impart it.

I thank Dr. Carl Crane for his unstinting support throughout the research phase, for extending facilities, and his knowledge in robotics which inspired this research. I would also like to express my appreciation to Dr. Sartaj Sahni for his assistance and kindly agreeing to be a part of the supervisory committee.

The Department of Energy for its support through the University Research Program in Robotics is also greatly appreciated.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

GEOMETRIC BIN PACKING ALGORITHM FOR ARBITRARY SHAPES

By

Arfath Pasha

August 2003

Chair: Meera Sitharam
Major Department: Computer and Information Science and Engineering

One waste remediation plan calls for repackaging hazardous waste into fifty-five gallon drums, which would then be melted in a plasma arc furnace. The objective of this effort was to develop a geometric bin-packing algorithm, which will fill a drum as completely as possible with arbitrary shaped objects while obeying certain physical constraints that make the final packing configuration realizable in practice. Some of these constraints are the effect of gravity on the objects being packed, minimization of center of gravity of the container, and allowable radiation dose levels for the container. The prevention of packing configurations that contain interlocking shapes may also be an important physical constraint as such configurations are difficult to achieve when the packing is performed with the help of a robotic arm. The proposed approach uses a genetic algorithm to optimize the packing order or sequence, and an online packing heuristic that is capable of finding near optimal placements for each object in the sequence. The problem is known to be strongly NP-hard and has many engineering

applications in areas such as container loading, stock cutting, layout optimization and rapid prototyping.

CHAPTER 1
INTRODUCTION

Motivation

This research was sponsored by the University Research Program in Robotics and was aimed towards finding an effective solution for the hazardous waste disposal problem being dealt with at Department of Energy sites. The objective of this task was to autonomously pack the hazardous waste into fifty-five gallon drums, which would then be melted in a plasma arc furnace. Central to the task was the development of a bin-packing algorithm that was capable of finding near optimal packing configurations for a set of irregular shaped objects.

In addition to the main objective of the algorithm, the autonomous nature of the packing process posed certain physical constraints such as the minimization of the center of gravity of the packed container, allowable radiation dose levels for the container, stability of the packed objects and the prevention of packing configurations that contain interlocking shapes, as such configurations are difficult to achieve when the packing is performed with the help of a robotic arm. Figure 1-1 shows a packing configuration that contains unstable placements. Figure 1-2 illustrates another packing configuration with interlocking shapes. A robotic arm cannot replicate these packing configurations.

This paper describes a methodology that may be used to meet the stated objectives. Although the method described is specific to the hazardous waste disposal problem, it may also be applicable to other packing applications such as container loading, stock cutting and rapid prototyping.

Figure 1-1. Packing configuration with unstable placement of objects. (Das97)

Figure 1-2. Packing configuration with interlocking placements.

Definitions

Optimization of the packing of arbitrary shaped objects into containers may be termed as a *geometric bin-packing problem*. The problem is essentially a *combinatorial optimization problem* that involves the *selection* and *arrangement* of items within a finite or discrete space, such that the resulting solution is integral in nature. Such problems are usually solved exactly and deterministically using integer-programming techniques. *Deterministic* algorithms give the same output for any particular input each and every time they are executed on that input. Polynomial time solvable problems usually employ deterministic algorithms to get the exact optimal solution. However, the geometric bin-packing problem is known to be strongly NP-hard and conventional integer programming methods such as branch and bound algorithms may take exponential time to solve the problem deterministically to optimality. In recent years, researchers have used

approximation and *randomized* algorithms and *heuristics* with varying degrees of success in trying to find near optimal solutions to the geometric bin-packing problem. For NP-complete problems where optimal solutions cannot be achieved in polynomial time, approximation algorithms are often used. An approximation algorithm is deterministic but inexact and it aims at obtaining a *near-optimal* solution in polynomial time. The level of approximation is given by an *approximation ratio* $\rho(n)$, which is the ratio of the cost of the solution provided by the algorithm to the cost of an optimal solution. This ratio is provably guaranteed. A randomized algorithm is one that makes some random choices. The behavior of randomized algorithms, depend not only on the input but also on the values produced by a random number generator. Although unproven, randomized algorithms are efficient in practice in finding near optimal solutions for NP-complete and NP-hard problems.

Deterministic heuristics and randomized heuristics such as simulated annealing and genetic algorithms have been used extensively for geometric bin packing problems. These approaches have been found to make a good trade-off between efficiency and quality of the results they provide, but offer no guarantees unlike approximation and randomized algorithms. These approaches are discussed here more detail.

Heuristics

As stated earlier, heuristics are also used to obtain good feasible solutions for NP-hard problems. Heuristics are usually deterministic algorithms that use a rule of thumb that is simple. The basic strategies for heuristics are divide and conquer and iterative improvement. Although heuristics work quickly and efficiently, the quality of their output may leave much to be desired. The performance of heuristics is usually tested by

running them on a set of inputs called benchmarks. The outputs of the heuristic for these benchmark inputs is then compared to the outputs of other heuristics run on the same set of benchmarks. This form of performance testing has some obvious drawbacks. The benchmarks represent only a small portion of the input universe and fail to describe the general problem instance. The quality of the outputs measured by the benchmark set may not help in improving the heuristic for general cases.

Throughout this paper, the word "pattern" is used for two-dimensional shapes and the words "object" and "part" are used interchangeably to denote shapes in three dimensions. The container for two-dimensional cases may be assumed to be rectangular in shape and that for a three-dimensional case may be assumed to be cuboids unless stated otherwise.

Randomized Heuristics

Simulated Annealing

Randomization has proved to be a very effective technique for finding near optimal solutions for highly combinatorial problems. The objective behind a randomization technique is to optimize a function using a random sampling of the solution space. Simulated annealing is one of the most popular randomized search methods being used in combinatorial optimization. It was first introduced by Kirkpatrick et al. (Kir83) when he combined statistical mechanics of multi-body systems with combinatorial optimization. The main idea behind simulated annealing is to reduce the overall energy of the system, which is defined by a cost function, in a gradual manner from a high-energy state to its ground state, which represents an optimal solution. This technique closely resembles the annealing process of metals where metals are heated to temperatures above their melting point and then cooled gradually to form uniform

crystalline structures. The important factors in the annealing process are the formulation of the cost function and the rate at which the energy of the system is reduced. If the energy of the system is reduced rapidly, the final solution is usually metastable. This is analogous to the quenching of metals which leads to a non-uniform crystalline structure with a higher than optimum energy state. The energy of the system is reduced by a random search that not only chooses solution points that reduce the objective function *f* but also accepts solution points that increase *f* with a probability *p*. A control parameter *T*, which is analogous to the temperature in the annealing of metals, is used to narrow the search down to a near optimal solution.

$$p = e^{-(\delta f / T)}$$

During the initial stage of the algorithm, the control parameter allows the algorithm to make large changes to its parameter values. This allows the algorithm to explore new regions in the parameter space. As the algorithm progresses, the control parameter is lowered slowly and this forces the algorithm to perform a neighborhood search that eventually yields to a near optimal solution.

Genetic Algorithms

Genetic algorithms, like simulated annealing, utilize a randomized search technique to find near optimal solutions for combinatorial problems. This method of optimization was first developed by John Holland in 1975 (Hol75) and was then made popular by one of his students David Goldberg in 1989 (Gol89). Over the last decade, genetic algorithms have received significant attention for their effectiveness and quality of solutions for problems that cannot be solved using conventional optimization techniques. Genetic algorithms or GAs are based on the natural evolution of living organisms. The process of adaptation in a changing environment is the key to survival.

This adaptation process takes place over a number of generations and tends to yield a highly fit set of individuals that are capable to producing off springs that have a higher chance of survival. The level of fitness of an organism may be determined from its genetic makeup. The genetic makeup of an organism contains information about the various attributes of the organism. This attribute information is stored in the genes of its chromosomes. The fitness may depend on the type of value that each gene takes on and the nature of interaction between genes. The structure usually leads to a highly nonlinear and epistatic solution space and despite its complexity, increasingly fit organisms are

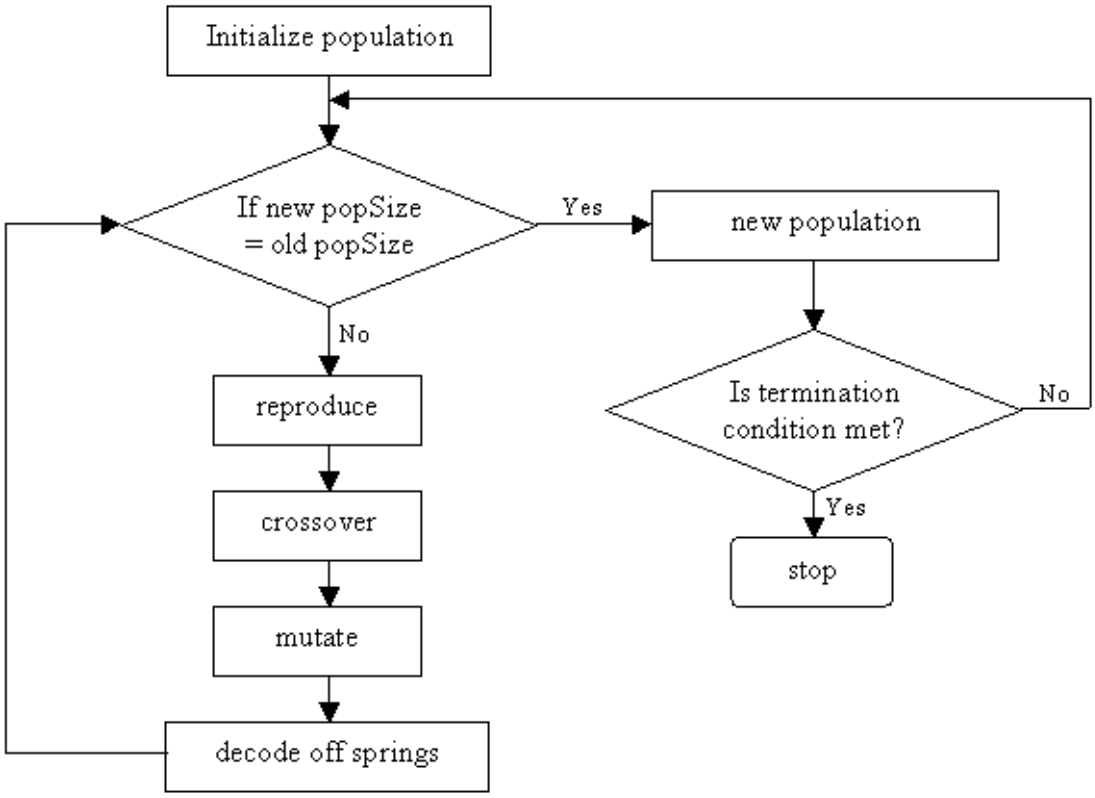


Figure 1-3. Structure of a typical genetic algorithm.

created as a result of the evolutionary process. Researchers have modeled various intractable problems such as the traveling sales person problem based on this technique and found that it can be used to find near optimal solutions.

The typical structure of a genetic algorithm consists of a population of a set of coded strings called chromosomes. These chromosomes are built up of smaller elements called genes. Each gene represents a certain attribute of the problem that the chromosome is modeled after. A gene may take on several different values called alleles. For example, the layout of a machine shop floor may be modeled by a chromosome whose genes represent the locations of the machines on the shop floor. The values that a gene can take on are called alleles. In the machine shop example, the finite set of locations that each gene can take on are the alleles for the gene. Each chromosome is associated with a fitness value that is calculated with the help of an evaluation function. The average fitness value of all the chromosomes in the population gives an idea of the fitness of the entire population. At the start of the algorithm, the genes are assigned random allele values and the fitness of this initial state is computed. Just as in nature, the refinement of the population occurs over several generations. During the course of each generation, the algorithm applies genetic operators such as reproduction, crossover and mutation to the chromosomes. Reproduction is the process of selecting chromosomes whose genes will be passed on to the next generation. The selection is usually based on the fitness of the chromosomes. Chromosomes that have higher fitness values have a greater probability of being selected. The crossover and mutation operators are applied to the selected chromosomes to produce the off springs that form the next generation. Crossover is the process of selecting genes from two parent chromosomes to form a new offspring chromosome. The crossover operator must ensure that some of the characteristics of the parent chromosomes that pertain to the problem are retained in the off springs. These characteristics are maintained in sub-strings of short defining length called schemata that

form the building blocks of chromosomes. The crossover operator helps in propagating highly fit schemata from generation to generation giving exponentially increasing samples to the observed best. The mutation operator randomly alters genes in the new population, usually with a very small probability. The purpose of the mutation operator is to get the algorithm to move out of local optimums by causing some random perturbation of the genes. A detailed view of the structure of genetic algorithms and their operators may be found in Goldberg (Gol89).

Geometric Bin Packing Problem Versions

Over the last three decades, the bin-packing problem has been studied by researchers in various forms. Research in this area began with the classical one-dimensional bin-packing problem, which served as a foundation for the analysis of approximation algorithms. It was one of the first combinatorial optimization problems for which performance guarantees were investigated. Since then, the problem has been broken down into several different versions based on various factors such as geometry of the objects, number of bins, nature of the problem and its constraints. All of these versions are very different from each other except for one common property - they all contain a *capacity constraint*. The bin or bins that need to be packed have a finite capacity that cannot be exceeded. Figure1-4 shows how the bin-packing problem can be broken up into different versions based on various factors. Different combinations of these factors yield different versions of the problem.

With the exception of the *single bin decision problem* with no input shape information, which is trivial, none of the versions of the problem are polynomial time solvable.

The *single bin optimization problem* with no shape information may be framed as a knapsack problem that uses the *size* of the objects as the profit. Such a problem may be framed as follows.

Given a set of objects L = {a1, a2, a3, ..., an}, $a_i \in \{0,1\}$, and each object having a certain size w_i, a container with size W,

$$\text{Max } \sum_{i=1..n} a_i w_i \text{ subject to } \sum_{i=1..n} a_i w_i \leq W$$

This version of the problem is known to be NP-hard.

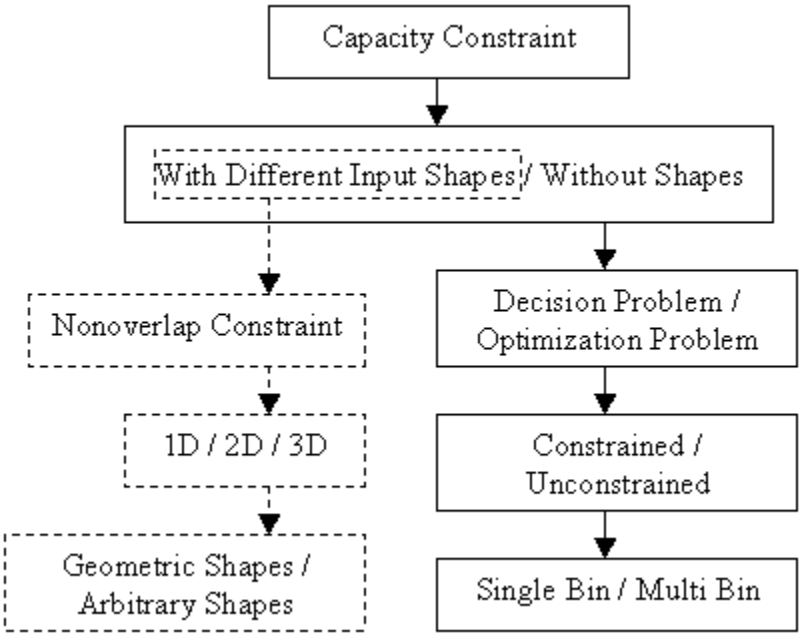


Figure 1-4. Factors that determine the version of the bin-packing problem

Single bin packing problems that consider the shape of the objects being packed are useful in applications like container loading and stock cutting. The decision version of the problem is NP-complete while the optimization version is strongly NP-hard even for simple geometric shapes. The decision version of the problem may be framed as follows.

Given a set of n three-dimensional (or two-dimensional) objects and a container with dimensions WHD (or WH), find if it is possible for all n objects to be packed into the container without overlapping.

In the optimization version of the problem, a finite set of n three-dimensional (or two-dimensional) objects must be packed into a container with dimensions WD (or W) and infinite H such that the packed height H' of the objects is minimized.

The single bin packing problem may be further categorized into *online* or *offline* packing. In the online packing process, the items must be packed in a predefined order or sequence, and information about the items is made available in that order. An online packing algorithm needs only to optimize the position and orientation of each item based on its shape and the space available in the bin after the previous items in the sequence have been packed. Since this method of packing performs only a local optimization, greedy algorithms and heuristics are best suited for it. Offline bin packing on the other hand is a global optimization process where there is no restriction on the order in which the items are packed. Most researchers that attempt to optimize a single bin taking shape into consideration resort to heuristic methods or randomized algorithms.

In addition to this, the algorithm may be further *constrained* or *unconstrained*. Additional constraints such as low center of gravity of the container, or order of removal of the objects may be used. Typically, heuristic search methods and randomized heuristics such as simulated annealing and genetic algorithms are used for this type of packing process.

Bin packing problems that involve the packing of multiple bins are known as *multi-bin packing* problems. The aim in such problems is usually to partition the items

into groups while ensuring that the net volume of each group is less than or equal to the capacity of each bin. Similar to the single bin packing problem, multi-bin packing problems may be framed as NP-complete decision problems where the objective is to find if it is possible to pack a finite set of items into a finite set of bins. They may also be framed as optimization problems where the objective is to minimize the number of bins that are required to pack a finite set of items. Also, just as in the single bin packing case that does not consider the shape constraint, the multi-bin packing problem with no shape constraint does nothing to find if the partitions that it created are actually feasible. The shapes of the items in each partition may prevent the items from being packed into the bin in spite of the volume constraint being met. This is illustrated in Figure 1-5. Although the grouping shown in the figure satisfy the capacity constraint, containers (a) and (b)

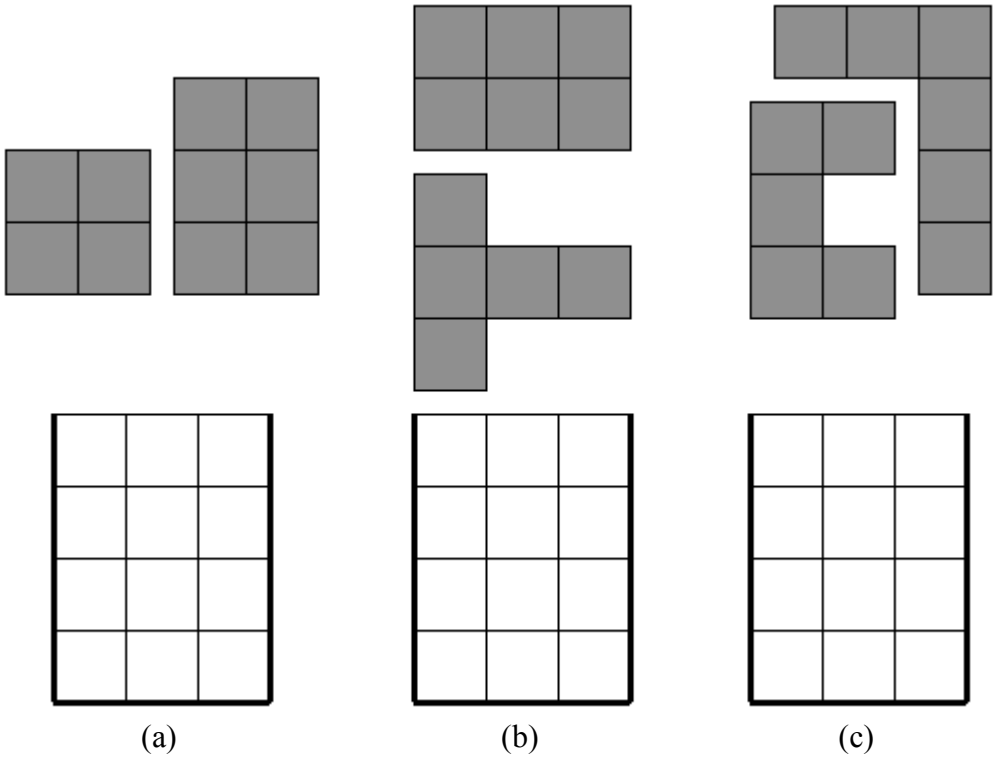


Figure 1-5. Multi-bin packing where capacity constraint is met but shape constraint is violated

cannot be packed without overlapping due to the violation of the non-overlap constraint. When the shapes of the items are taken into consideration, the problem becomes even more intractable. As a result, research for the multi-bin packing problem with non-overlap constraint has been limited mostly to rectangular shaped objects. Multi bin-packing problems have almost always been treated as offline bin-packing problems.

General Formal Problem Statement

Based on the categorization of the bin-packing problem given earlier, the problem of packing contaminated waste may be framed as a constrained three-dimensional multi-bin packing problem with a non-overlap constraint as follows.

> Given a finite set of three dimensional objects of arbitrary geometry, and an infinite number of containers with dimensions WHD (or RH if container is cylindrical), pack without overlapping or splitting, all the objects into the minimum number of containers subject to the following constraints.

- The center of gravity of each packed container must be below a certain threshold value.
- The cumulative dose of each packed container must be below a certain allowable value.
- The packing configuration of each container must not contain inter-locking shapes.
- The final position and orientation of each object in the container must result in a stable placement when the objects are placed in the order determined by the algorithm.

Complexity

The factors that make the bin-packing problem hard are the *packing order* (sequence in which the objects are packed) and the nature of the shape and size of the items. For a set of n items, there are n! different sequences in which the bin may be packed. For each such sequence, there may be several different ways of placing the items into the bin based on the nature of the shape and size of the items. Figure 1-6. shows an

instance of a packing problem that may not only have infinite solutions, but also infinite optimal solutions. The packing configuration shown may be rotated by small amounts about the vertical axis of the container to yield infinite similar solutions. Thus, the solution space for such a bin-packing problem is extremely large and multi-modal. For problems such as this, it is sufficient if an algorithm is capable of finding a "good" feasible solution that can be computed efficiently. Randomized algorithms such as simulated annealing and genetic algorithms are known to do just this.

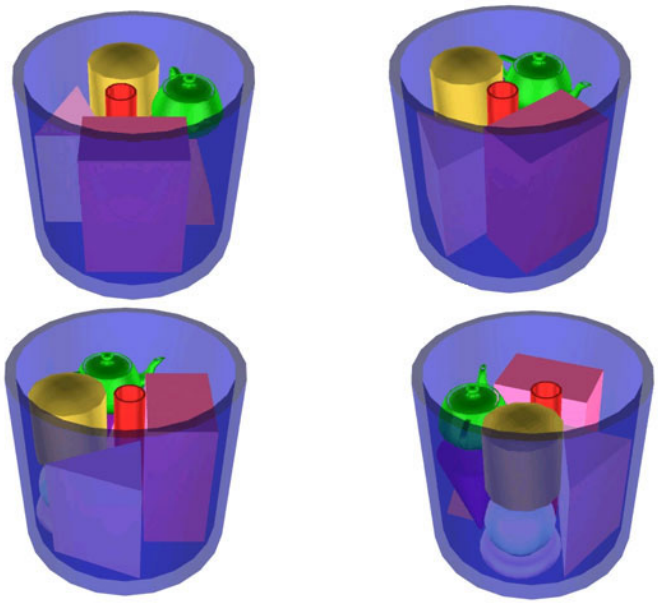


Figure 1-6. An infinite solution space with infinite feasible solutions may be possible in the three-dimensional bin-packing problem

In addition to the packing order, the size and shape of the items being packed play an important role in the design of a packing algorithm. Most often, all the items that need to be packed have specific geometric shapes such as rectangular, rectilinear or spherical shapes. Geometric shapes tend to reduce the solution space in terms of the number of optimal positions and orientations that each item may have. A sphere for example, may be placed at several positions in the bin but it has only one fixed orientation. An algorithm designed to pack 2D rectangular shapes may take advantage of the fact that the

optimal solutions for most input instances has the items oriented in one of two possible 90-degree orientations. This automatically reduces the feasible set of orientations, and therefore the solution space. The *strip-packing* algorithm developed by Lodi et al. (Lod99) makes use of size and shape information of the items to narrow down the search for a near optimal solution. Non-geometric or irregular shaped items may have extremely large or even infinite feasible positions and orientations.

CHAPTER 2
LITERATURE SURVEY

This section gives an overview of some of the approaches and optimization techniques that have been used in the past for the two and three-dimensional bin-packing problem.

Theoretical Work

Due to the nature of the complexity of the problem, relatively less work has been done in analyzing bounds when compared to the actual development of heuristics and algorithms for the problems in two and three dimensions. Among the theoreticians in this field, Silvano Martello and Daniel Vigo have been prominent researchers in the areas of numerical simulation and combinatorial optimization and have published several papers that describe exact and approximation algorithms for the bin-packing problem in two and three dimensions. Their work focused on packing rectangular shapes into the least number of bins. They have presented a lower bound for two-dimensional bin-packing problems with rectangular shapes that may be rotated by 90 degrees (Mar98). They have proved that the worst-case performance ratio of the sum of the area of the packed rectangles to the area of the container is 1/4. A branch and bound algorithm was used to test the effectiveness of the lower bound. The lower bound was later extended to three dimensions and verified with a similar branch and bound method in Martello et al. (Mar00). Experimental results have shown that smaller instances of the problem can be solved to optimality using exact algorithms, but for larger instances, approximation algorithms are required. Martello et al. (Mar98) explored *strip packing* and *tabu search*

methods for obtaining good approximations with larger instances. In the strip packing procedure, all the patterns are packed into a strip of infinite width and height equal to the bin height. The packed strip is then partitioned into slices of width equal to the bin width. The patterns that occupy each slice are packed into at most two bins. The tabu search method is a meta-heuristic that is applied to an underlying heuristic and its goal is to prevent the underlying heuristic from cycling in a local optimum by forbidding or penalizing moves that tend to guide the heuristic into a local optimum.

Heuristic Search Methods

Albano and Sapuppo (Alb80) resorted to heuristic search methods used in artificial intelligence to optimize the layout of irregular shaped two-dimensional patterns on large stock sheets. They developed a deterministic solution to the allocation problem using the A* heuristic search method in Nilsson (Nil71). A simple set of rules was used to place the patterns on the sheet metal. After a pattern was placed, a profile that separated the available space from the occupied and wasted space was generated. This profile aided in the placement of the next pattern. For the n remaining patterns, k orientations were sampled on the current profile and of the $n \times k$ possibilities, a fixed number of successors were chosen based on the least amount of wasted space. Information of the chosen successors was maintained in the form of a directed graph where the edges represented the amount of wasted space and the nodes represented the patterns in particular orientations. By decomposing the allocation problem into a graph problem, the author was able to apply the A* search heuristic and expand potentially good nodes based on current estimates of the total wasted space. The size of the graph was dynamically maintained in order to reduce the amount of time taken to find a good

solution. This created a trade-off between the quality of the solution and the time taken for the solution produced. Figure 2-1 shows an example output of the heuristic.

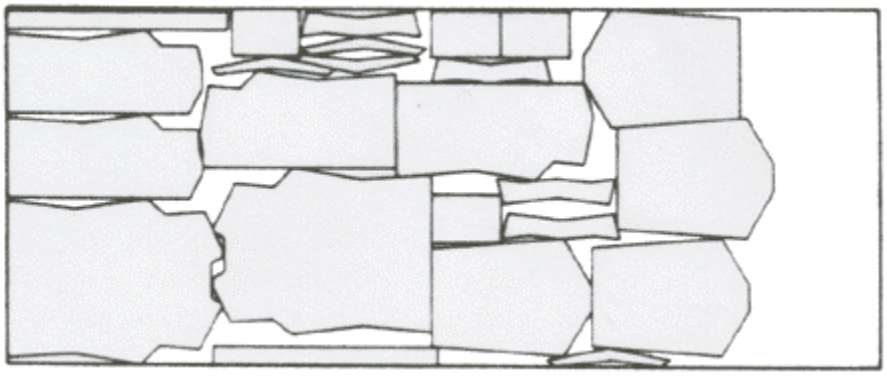


Figure 2-1. Albano and Sapuppo, A* heuristic output, 1980

Another heuristic search method was explored by Robert Mcgee (McG97) for the online packing of three-dimensional irregular shaped objects into a cylindrical drum. The parts and container were modeled with the help of a voxel data structure. There was no attempt made to optimize the packing order of the objects. The objective of the heuristic was only to minimize the void space and trapped space that was created from each placement of an object in the container. Void space was defined as the space directly below the object just placed, that was not already occupied by the previously placed objects. The space between the object just placed and the walls of the container was considered as trapped space if this space contained too few continuous voxels. In order to place the objects into the container with minimal computation, a data structure called a chain code matrix was used to keep track of the surface voxels of the parts in the container. The heuristic used a brute force method of checking for all possible placements of the object on the chain code matrix surface and with a one voxel translation resolution. For each position, an orientation resolution of *360/theta* was used about all three axes of rotation. *theta* was a user-preset parameter. For each placement, a quick surface

interference check was performed to check the feasibility of the placement. If a feasible placement was found, it was checked for stability. The void space and trapped space were then computed if the object was found to be stable. The best placement of all the feasible and stable placements was then chosen based on the least void space and trapped space that it created. The Figure 2-2 below shows the placement of six parts in the cylindrical container using this method.

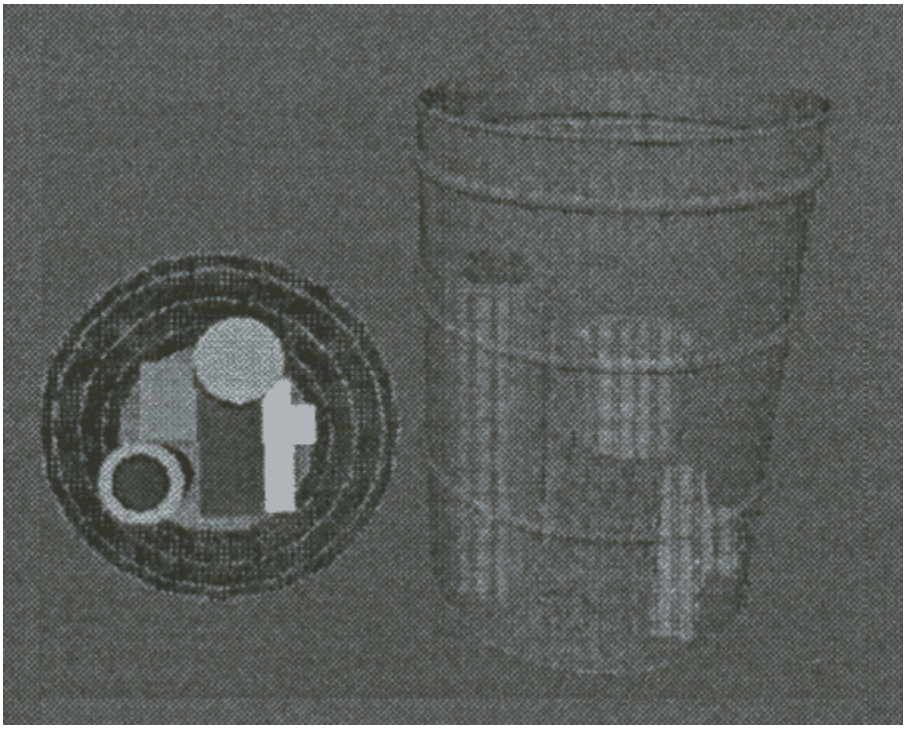


Figure 2-2. Robert McGee, online heuristic output of six packed shapes, 1997

Randomized Heuristics

Cagan et al. (Cag96) used simulated annealing to optimize a three-dimensional offline bin-packing problem for irregular shapes. The packing problem was formulated as a multi-objective optimization problem. Each item possessed an attractive force based on its distance to the centroid of the container. Penalty forces were given to volumes lying outside the container and to intersecting volumes. These individual forces were then summed up and weighted. The objective of the simulated annealing algorithm was to

minimize the weighted sum. An octree data structure was used to model the items and a multi-resolution modeling technique was implemented to reduce the amount of time taken for interference checking. At higher temperatures, low-resolution models of the items were used, and as the temperature was lowered, the accuracy of the octree models was increased. This time saving method was justified by the fact that at higher temperatures, the algorithm does a random walk in the solution space and does not require an accurate estimate of the objective function. But, at lower temperatures, the system performs a neighborhood search, which implies that the overall state of the system does not change much. This makes a more accurate evaluation fast with the multi-resolution modeling technique. Results of tests performed on benchmark problems containing four and sixty-four cubes were presented.

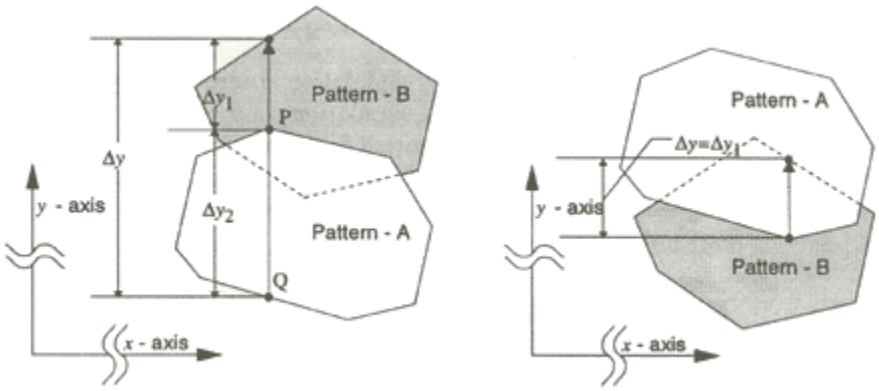


Figure 2-3. Computing the displacement along the y-axis in the heuristic by Ono and Wanatabe 1997 (cited in Das97).

Ono and Watanabe (cited in Das97) used a genetic algorithm to optimize the usage of sheet metal when arbitrary two-dimensional patterns had to be cut out of it. Their approach used the ordering of the patterns to model the chromosomes. This set the search space to n! where n is the number of patterns. The fitness of the chromosome was evaluated based on a heuristic called the Layout Determining Algorithm (LDA) that was

used to find the placement of the patterns on the sheet metal with no mutual overlap. The fitness was a measure of the sheet length used for a particular ordering or chromosome. The LDA moves the pattern along the sheet metal's height and width until it finds a position for which the pattern does not overlap the previously placed patterns. This is as shown in Figure 2-3. The increments with which the pattern is moved is based on simple interference checks along the X and Y directions for each vertex in the patterns. The first feasible position found for a pattern is its final position on the sheet. No attempt is made to optimize the position of a pattern for a particular ordering. In addition to this, the

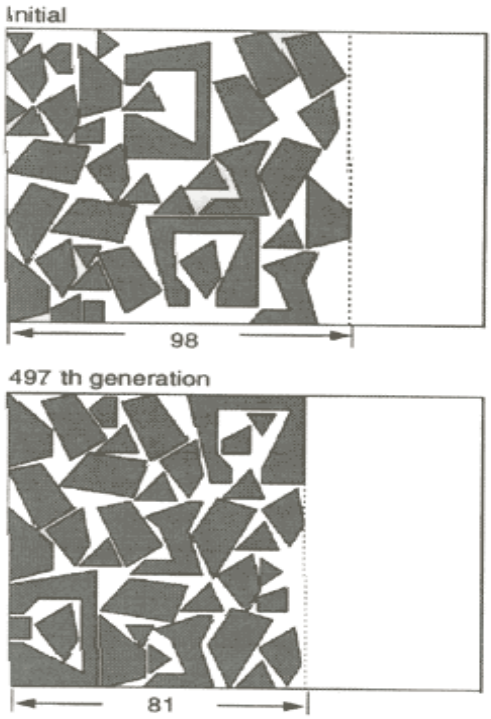


Figure 2-4. Results from a genetic algorithm, Ono and Watanabe (cited in Das97) 1997.

pattern are considered non-orientable by the LDA. Results show a convergence in the genetic algorithm as shown in Figure 2-4 below. The paper also shows the comparison of three popular crossover operators - CX (cyclic crossover), PMX (partially mapped crossover) and OX (order crossover) that were used for this problem.

CHAPTER 3
RESEARCH OBJECTIVES

As stated earlier, the problem of packing contaminated waste calls for the development of a three-dimensional bin-packing algorithm that is capable of producing near optimal packing configurations while obeying certain physical constraints. Due to the complex nature of this problem, the author chose to simplify the problem into a two dimensional packing problem as a first step. The focus of this research was limited to finding a good feasible solution rather than a theoretical analysis. Insight gained from handling the simpler form of the problem and from the results obtained from it may then be used to effectively come up with a solution for the problem in three dimensions. The final three-dimensional case was left for future work. In order not to loose the main properties of the original problem, some of the physical constraints that applied to the original three dimensional problem had to be incorporated into the two dimensional case. Also, the simplified two-dimensional problem had to be modeled in a way that could be easily extended into three dimensions. In order to accommodate all these changes, the generalized formal problem statement was reformulated into the simplified formal problem statement given below.

Simplified Formal Problem Statement

> Given a finite set of 2D polygonal patterns of arbitrary shape and a rectangular container with width W and infinite height, find a packing configuration that minimizes the packed height of the container such that none of the patterns violate the following physical constraints.

- Patterns do not overlap.

- Every pattern must be stable when placed in the order defined by the final packing configuration.
- There should be no interlocking patterns in the final packing configuration.

Other constraints such as allowable radiation dose levels of the container and the minimization of the center of gravity of the container were relaxed to further simplify the problem. Once the proposed solution has been described, it will be shown how these constraints may be added on at a later stage.

Our Preliminary Attempts

During the initial stages of the research, various approaches were analyzed and experimented with. The assessment of the advantages and drawbacks of these experiments aided in the formulation of the final solution to the 2D packing problem.

Attempt 1: A Simple Genetic Algorithm

Since the solution space was large, irregular and multimodal, the problem called for a randomized approach and genetic algorithms was chosen for this. At first, a simple genetic algorithm was implemented for rectangular patterns. The objective function was to minimize the intersecting area of the rectangles by moving them around within a container rectangle using random translations and 90-degree rotations. The chromosome for this problem was coded as a series of (x, y, theta) values, denoting the position of the patterns' centroids and orientations. A single point crossover operator with a simple mutation operator was used to model the algorithm. Despite the weak objective function and the poor modeling of the chromosome with respect to the problem objective, the algorithm showed convergence and the final results showed that the algorithm retained chromosomes that represented patterns that were separated out. It gave preference to larger patterns since more was to gain from separating them out. This exercise helped in

gaining insight into the working of genetic algorithms, their capabilities and pitfalls. The approach itself did not suit the problem at hand since it did not do anything to address the stability and interlocking shape issues. Besides, the task of interference checking can be very expensive when it is performed on arbitrary shapes. The Figure 3-1 below illustrates an output of this implementation.

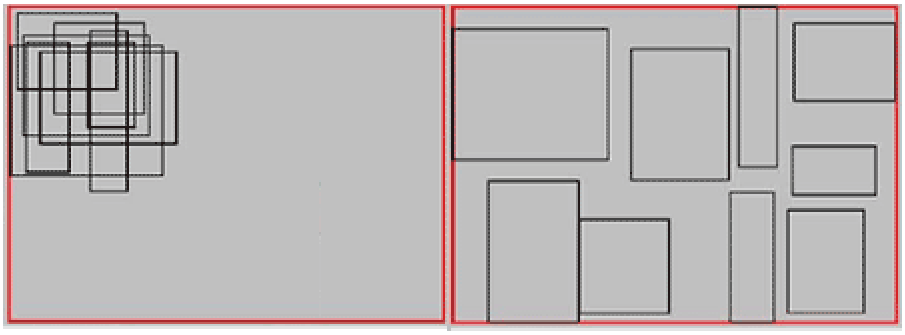


Figure 3-1. The simple genetic algorithm. (a) input (b) output after 989 generations

Attempt 2: Divide and Conquer Explored

The objective function of minimizing the intersecting areas of the patterns not only set the solution space as infinite (with mostly infeasible solutions), but also did not give the algorithm enough physical meaning. The result was a slow convergence. In an effort to find a stronger objective function, a divide and conquer approach was explored. The approach involved the partitioning of the container into two or more partitions and optimizing each partition separately while using the same random translation and rotation procedure as before for each partition. Although this method was more time efficient and gave the objective function more physical meaning, it was plagued with several problems. The partitions had to be at least as large as the largest pattern and this is hard to determine for arbitrary shaped patterns where the bounding box changes with the orientation of the pattern. The chromosomes could not be modeled without having

duplicates after the crossover operator was applied. Falkenauer (Fal96) suggests a procedure that can be used to eliminate duplicates for the multi-bin packing problem. The procedure is found to work well for the classical bin-packing problem, but it can be laborious and inefficient for the geometric single bin packing case that also takes placement of the patterns into account. Also, the space between partitions may not be utilized effectively and this would lead to poor final packing configurations. The issues with interlocking shapes and stable placement were still not addressed by the algorithm.

The main disadvantage in the first two attempts came from mixing the packing order and the placement aspects of the algorithm.

Idea of Main Contribution: A Hybrid Genetic Algorithm

At this point, a decision was made to further simplify the problem by breaking it down into two parts, ordering and placement, and tackling them separately. This idea was independently discovered and was later found in Dasgupta et al. (Das97). The order in which the patterns were placed in the container was known to affect the quality of the packing configuration. Also, for a given ordering, a mechanism was required to find the best position and orientation for each pattern as it was placed in the container. This breakup was then fit into the genetic algorithm structure by using a chromosome built up of pattern IDs as shown in the Figure 3-2 below. Each chromosome represents a particular ordering and is associated with a fitness value. The fitness value gave an idea of how good (or bad) the packing was when the patterns were placed in the order specified by the chromosome. The placement of the irregular shaped patterns is a non-trivial task that required a separate optimization procedure. Similar to the approach used in Dasgupta et al. (Das97), a heuristic was used for this task.

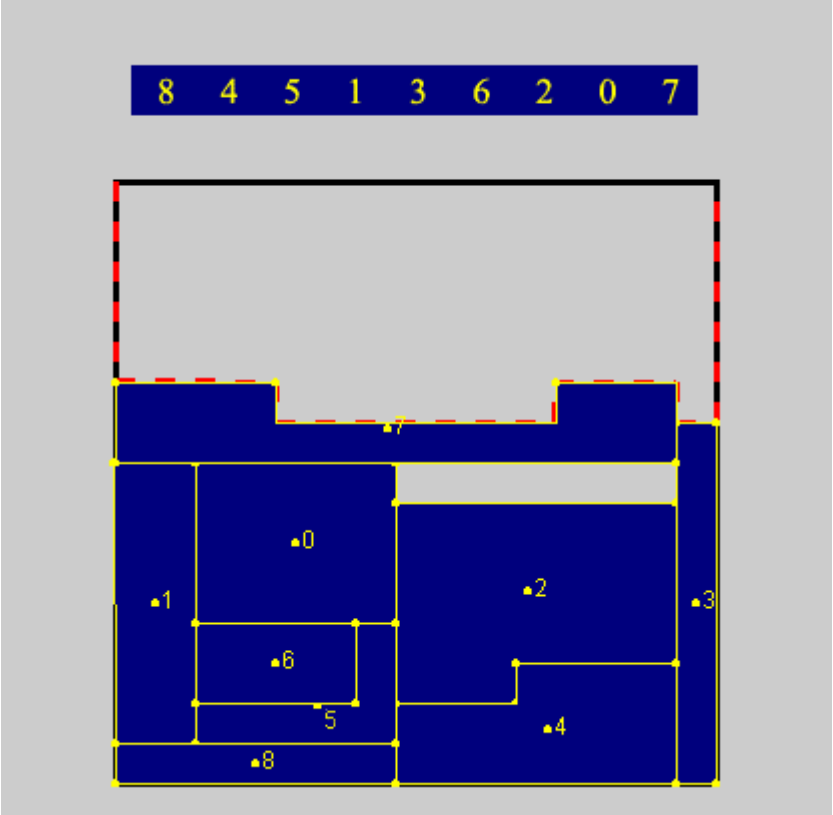


Figure 3-2. Chromosome containing pattern IDs and representing a packing configuration obtained by a heuristic

Packing problems in two dimensions can be modeled as either *tiling* problems or *stacking* problems. In the tiling approach, the patterns are packed in a horizontal plane similar to the laying of mosaic tiles on the floor. This method of packing may lead to packing configurations with interlocking patterns when arbitrary shapes are to be packed. Also, the effect of gravity and stability of the patterns cannot be incorporated into the tiling approach. The stacking problem is similar to the game of Tetris where patterns fall in a vertical plane and have to be positioned and oriented appropriately as they fall. This form of packing does not allow the formation of interlocking patterns even for arbitrary shaped objects. The notion of gravity and stability of the packed patterns may also be incorporated into this method. The stacking approach was thus found to resemble the contaminated waste disposal problem and was the basis for the placement heuristic. The

next section describes this new approach of breaking up ordering and placement in more detail. After having made the decision to simplify the problem to two dimensions and model it as a stacking problem, the problem was stated formally as follows.

The choice of breaking the problem up into two parts gave rise to a hybrid algorithm that used randomization to optimize the order in which the patterns were packed and a deterministic placement heuristic to optimize the placement of the patterns in the container. Not only did this approach make the problem more tractable, it gave the genetic algorithm the physical meaning it needed to perform the optimization effectively. In addition to this, the physical constraints of stability and the prevention of interlocking packing configurations could be addressed with this new approach. The design and implementation of this approach is described in detail in the following section. Later sections show how this approach can be extended into three dimensions.

CHAPTER 4
MAIN CONTRIBUTIONS OF CURRENT WORK

This work is an extension of the work done by Robert McGee in 1997 (McG97). The shortcomings of his algorithm were analyzed and several new features have been designed into the current algorithm to better meet the requirements of the contaminated waste disposal problem. McGee's implementation utilized an online heuristic, that is, it assumed an input order of objects to be placed and following this order, it optimized the placement of each object in the container. The approach in this paper not only optimizes the placement of the patterns in the container but also the order in which the patterns are placed into the container. The voxel data structure used in the earlier implementation has the tradeoff between resolution and efficiency in terms of both space and time complexity. The algorithm in this paper uses a polygonal data structure, which can easily be replaced with one of the commonly used boundary representation data structures for the 3D case. This form of representation of the objects is more efficient and will provide better approximations to the real shapes. McGee's implementation used a brute force method to find a good placement for each object by translating the object over every voxel on the surface profile for a discrete set of orientations. The placement heuristic that has been developed in this paper also finds "good" placements and has a linear average case running time. To make the algorithm more efficient, no interference checking is performed in the current implementation. Finally, stability and interlocking shape constraints that were incorporated in McGee's implementation have also been considered in the current implementation.

Assumptions

To further simplify the problem, the following assumptions have been made, without loss of generality, about the nature of the patterns being packed.

- The patterns do not have holes. This assumption was made since it is not possible to fill holes in the stacking method for the two dimensional case. When the stacking problem is extended to three dimensions though, holes may be filled with other objects using this method.
- The patterns are assumed to be of unit thickness and are all made of the same material. Therefore, the center of mass of each pattern coincides with its centroid. This assumption also implies that the center of gravity of the packed container is automatically minimized by the minimization of the packed height of the container since all the patterns have the same density.
- The container has a rectangular profile and is assumed to have unit depth too.

Genetic Algorithm

The model for the genetic algorithm chosen in this paper represents chromosomes as the packing order or sequence in which patterns are packed into the container. The number of patterns *n* that need to be packed therefore determine the length of each chromosome. Each chromosome consists of an array of integers in the range *0* to *n-1*, such that every element in the array holds a unique integer in that range. Integers in the array represent pattern IDs. An array element *i* containing pattern ID *j*, implies that the pattern with ID *j* is the *i'th* pattern to be packed into the container. The fitness values of chromosomes are computed with the help of the placement heuristic that actually performs the packing for each chromosome. Figure 4-1 illustrates the structure of the packing algorithm.

The algorithm begins with the initialization of a population of randomly generated set of chromosomes that are decoded with the help of the placement heuristic. In order to

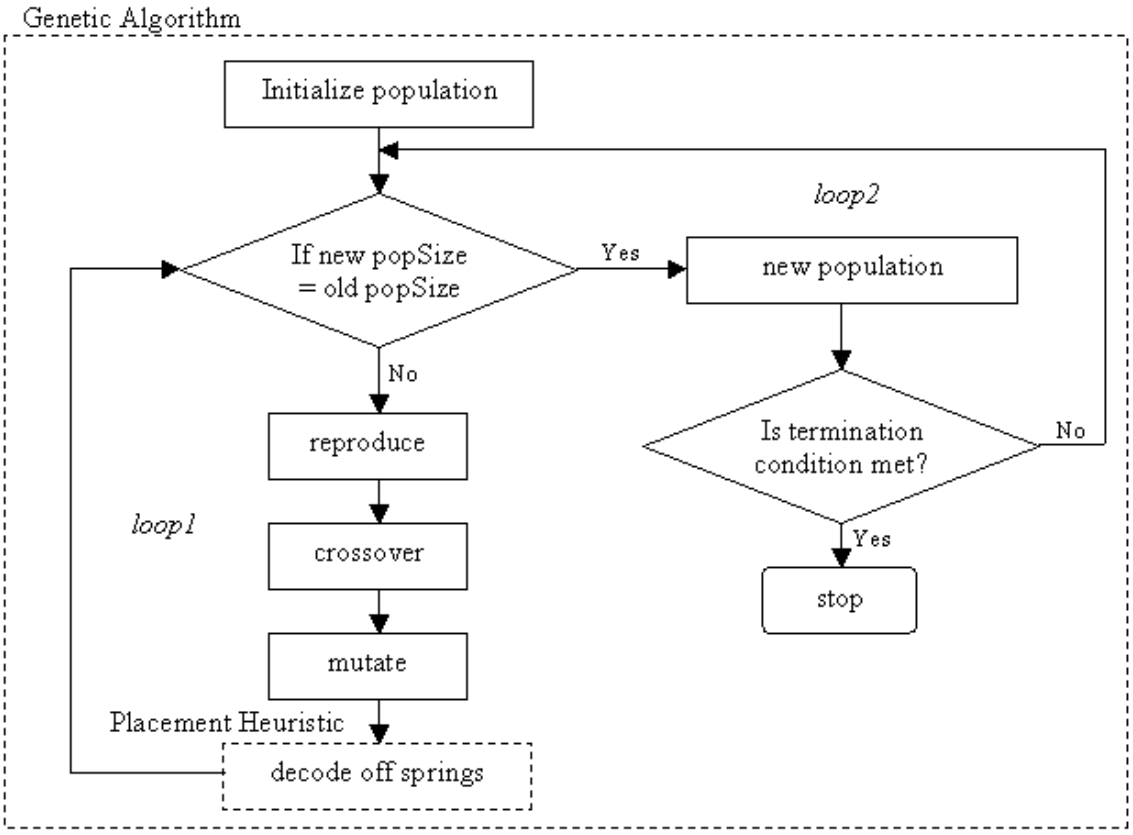


Figure 4-1. Flowchart of genetic algorithm

initialize the chromosomes with random alleles, a function that generates uniform random permutations of an array must be used. The function RANDOMIZE-IN-PLACE that is described in Corman et al. (Cor01) is used. The function runs in O(n) time and its pseudo-code is given below.

RANDOMIZE-IN-PLACE(*A*)

1. $n \leftarrow length[A]$
2. **for** $i \leftarrow 1$ **to** n
3. **do** swap $A[i] \leftrightarrow A[\text{Random}(i,n)]$

Once Initialization of the population has been performed, the algorithm is run for several generations until the termination condition is met. For each generation, loop1 in Figure 4-1 is run *p/2* times (*p* being the population size is always an even number) and

two off springs that are part of the next generation are created with each execution of loop1. Loop2 forms the outer loop and runs once for each generation.

Reproduction is performed with a simple roulette wheel selection procedure where pairs of chromosomes are randomly selected with a probability proportional to their fitness values. The roulette wheel selection method gives chromosomes that are more fit, a better chance of propagating their genes to future generations while not completely ignoring the weaker chromosomes. The pseudo-code for this function is presented below.

REPRODUCTION(*Pop*, *sumFitness*)

1. *jj* ← 0
2. *partSum* ← 0
3. *sRand* ← RANDOM(0,*sumFitness*)

4. **while** true
5. **if** *sRand* ≥ *partSum* and *sRand* ≤ *partSum*+*Pop*[++*jj*].*fitness*
6. **then return** *jj*
7. *partSum* += *pop*[*jj*].*fitness*

REPRODUCTION takes as input, the population and the sum of the individual fitness values of the chromosomes in the population and returns the index of the chromosome that is to be reproduced. At first, a random number *sRand* is chosen between *0* and *sumFitness* in line 3. The while-loop in lines 4-7 loops through all the fitness intervals from *0* to *sumFitness* until it finds an interval that *sRand* lies in. When an interval that bounds *sRand* is found, the index of the corresponding chromosome is returned. This operation needs to be performed twice to yield two parents that can be crossed and probably mutated to get two off springs. REPRODUCTION also runs in O(n) time.

The main objective of the crossover operator is to create off springs from parents that have something in common with the parent chromosomes in terms of their context. That is, the decoded offspring must exhibit some similar characteristics from both decoded parent solutions. The crossover operator must also yield children that are not too similar to each other or to the parents. If this happens, all the chromosomes in the population will begin to look similar after a few generations and this would result in a degenerate or incest population that is incapable of searching the solution space. In addition to these general requirements for a good crossover operator, the order based encoding scheme chosen for the packing problem calls for an operator that does not produce chromosomes with duplicate genes. Each gene in the chromosome must contain a unique integer in the range *0* to *n-1*. This added requirement cannot be fulfilled from the basic single point type crossover operators and calls for something more sophisticated. Michalewicz (Mic92) describes three operators - CX (cyclic crossover), PMX (partially mapped crossover) and OX (order crossover) that are capable of meeting this requirement. The PMX and OX operators are somewhat similar except that the OX operator gives more importance to the relative ordering of the genes while the PMX operator gives importance to both ordering and position information. The CX operator, like the OX, retains the relative ordering information of the genes. Of the three operators, CX and OX were found to be the least disruptive from the point of view of relative ordering. OX was finally chosen because of its simplicity in terms of implementation.

The OX operator is described with the help of the following example. Consider the two parents p1 and p2 below.

p1 = 7 4 3 5 2 1 6 9 0 8

p2 = 3 4 1 6 8 2 0 5 7 9

At first, two crossover points are chosen randomly. The crossover points are marked by '|'.

p1 = 7 4 3 5 | 2 1 6 | 9 0 8

p2 = 3 4 1 6 | 8 2 0 | 5 7 9

The portions of the chromosomes that are between the crossover points are copied into the offspring.

o1 = x x x x | 2 1 6 | x x x

o2 = x x x x | 8 2 0 | x x x

Beginning from the second crossover point of the second parent, the genes are copied in the same order with the exception of the genes that lie between the two crossover points of the first parent. We get the sequence 5 7 9 3 4 1 6 8 2 0 which reduces to 5 7 9 3 4 8 0 when 2 1 6 is removed from it. The reduced sequence is then used to fill the remaining placeholders in the first chromosome starting at the second crossover point to yield the completed first child,

o1 = 3 4 8 0 | 2 1 6 | 5 7 9

Similarly, the second child is,

o2 = 3 5 1 6 | 8 2 0 | 9 7 4

As seen from the above example, both off springs contain substrings whose relative ordering can be found in the parent chromosomes. The relative ordering of substring 3480 in *o1* is found in *p2*, and the relative ordering of substring 820 in *o2* is also found in *p2*. The off springs are also very different from their parents and contain no duplicates.

```
ORDER-CROSSOVER(P1, P2)

1.  for ii ← 1 to n       // n is the length of the chromosome
2.    do O[ii] ← P1[ii]

3.  if RANDOM(0,1) < 1-PCROSS
4.    then return O

5.  c1 ← RANDOM(0,n)
6.  c2 ← RANDOM(c1,n)

7.  for ii ← 0 to n
8.    do rem[ii] ← P2[ii]

9.  for ii ← c1 to c2
10.   do for jj ← 0 to n
11.     if  P1[ii] = rem[jj]
12.       then rem[jj] ← -1

13. jj ← c2
14. for ii ← 0 to n
15.   do if rem[ii] ≠ -1
16.     then O[jj%n] ← rem[ii]
17.          jj++

18. return O
```

The pseudo-code for the order-crossover is given above. The function takes two parents *P1* and *P2* as input and outputs a single offspring. In order to get two off springs from the same parents, the function must be called a second time with the order of the parents inverted in the function call. The function returns at line 4 (100-PCROSS)% of the time, where PCROSS is the probability of crossover. If the function returns at line 4, the returned offspring is similar to *P1*. Lines 5-6 choose random cutoff points *c1* and *c2*. Lines 9-12 filter out the alleles *rem,* that lie within the cutoff points of the offspring from the remaining alleles of *P2*. These remaining alleles are inserted into the offspring in

lines 13-17. The crossed over offspring is returned in line 17. ORDER-CROSSOVER also runs in linear time.

Once the chromosomes have been subject to the reproduction and crossover operators, a simple mutation operator is applied to them. The operator is applied to about 1% of the genes processed. When applied, it randomly chooses two genes in a chromosome and swaps them. The pseudo-code for the mutation operator is given below.

MUTATION(*O*)

1. **for** *ii* ← 1 **to** *n* // n is the length of the chromosome
2. **do if** RANDOM(0,1) < PMUTATION
3. **then** swap(*O*[*ii*], *O*[RANDOM(0,*n*)])
4. **return** *O*

Other parameters such as the size of the population and number of generations required, or the termination condition are based on experimental results and are discussed in the next section.

Optimal Placement Algorithm

General Approach

The placement algorithm formed the inner most loop in the main genetic algorithm. During every generation, it had to be executed once for each chromosome in the population. For inputs that contain patterns with identical shapes, the algorithm depended solely on the heuristic for good packing configurations. The placement algorithm therefore needed to be extremely efficient in terms of both quality of placements and time complexity. Also, the placement algorithm had to be deterministic so that the fitness value for each unique chromosome was always the same.

The general structure for the placement algorithm comprised of placing patterns on top of a *profile* that was maintained in the container. The profile was made up of a list

of straight-line edges that marked the upper most edges of the patterns that were already packed. At the start of the placement algorithm, the profile was initialized to the two walls and floor of the container. This is as shown by the red dashed line in Figure 4-2(a). After each pattern was placed, the profile was updated in such a way as to blanket the pattern that was just packed. This is illustrated in Figure 4-2 (b,c,d). The search for a good

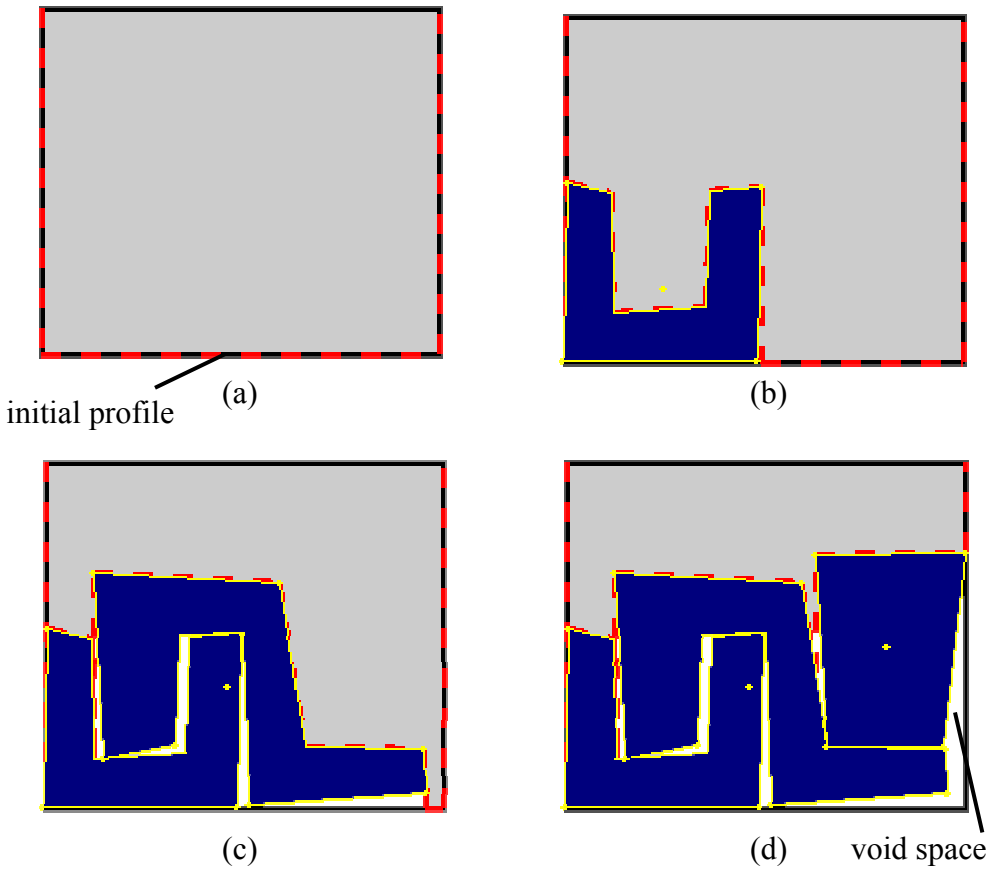


Figure 4-2. Placing objects on top of a profile V

placement involved finding a position and orientation for the pattern that resulted in the least *void space* and for which the physical constraints were not violated. The void space was defined as the space directly below a pattern that lay between the lower edges of the pattern and the profile. This too is illustrated in the Figure 4-2(c).

Geometric Conventions

A set of geometric and graphics conventions had to be devised before the placement algorithm was formulated. Two coordinate systems and a vertex numbering system were put together for this. These conventions are illustrated in Figure 4-3. The graphics coordinate system had its origin positioned at the upper left corner of the screen and oriented such that its positive y-axis was directed downwards and its positive x-axis directed from left to right. The graphics coordinate system was considered as the global coordinate system and was mainly used for graphics operations. Vertices of the container, profile and pattern were represented in the graphics coordinate system. The container coordinate system had its origin positioned at the lower left corner of the container and oriented such that its positive y-axis was directed upwards and the positive x-axis from left to right. This coordinate system was useful from the real world application point of view. The position and orientation of the packed patterns could be referenced from this coordinate system. The profile vertices were numbered from the top left to the top right corner of the container. The pattern vertices were ordered in the clockwise direction in the screen (or graphics) coordinates. A Clockwise ordering of the vertices in the graphics coordinate system results in a counter-clockwise ordering in the Cartesian coordinate system. This allows the use of basic polygon algorithms written for the conventional counter-clockwise ordering of vertices in the Cartesian coordinate system. The position of a pattern was defined by the position of its first vertex in the graphics coordinate system. Its orientation was defined as the angle subtended by the positive x-axis of the graphics coordinate system and the vector along the first edge of the pattern directed from vertex0 to vertex1.

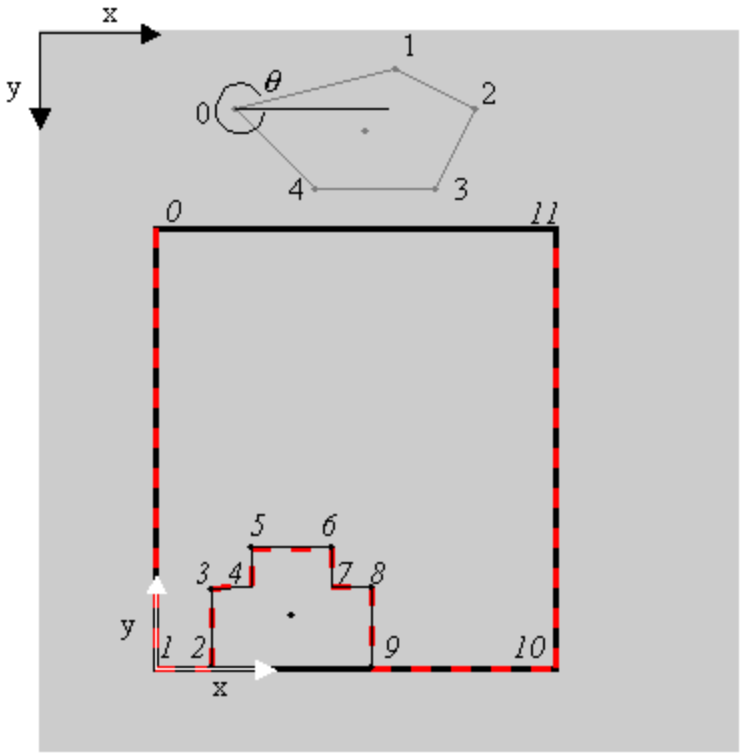


Figure 4-3. Geometric conventions used in the placement heuristic

Proof: Optimal Placement for the 2D Non-Oriented Case

Sitharam and Wu (Sit02), took the lower edges of the 2D pattern and the profile edges as two sets of piecewise linear functions that are monotonic along the x-axis and proved that it was possible to find an optimal placement in linear time when the pattern edges can be translated along the x and y axes but not rotated. The entire proof is presented below.

Given two continuous piecewise linear functions $f : (0,t) \rightarrow R$ and $g : (0,s) \rightarrow R$; where $s < t$, s and t positive; f has n linear pieces and g has m linear pieces. The goal is to design an efficient algorithm to find $0 < u < t\text{-}s$, and v in R such that the function $h_{u,v}(x) : (u, s+u) \rightarrow R$, defined as $g(x\text{-}u) + v$ satisfies two properties.

1. $h_{u.v}(x)$ is atleast $f(x)$ on h's support
2. $|| h\text{-}f ||_1$ (taken on h's support is minimized).

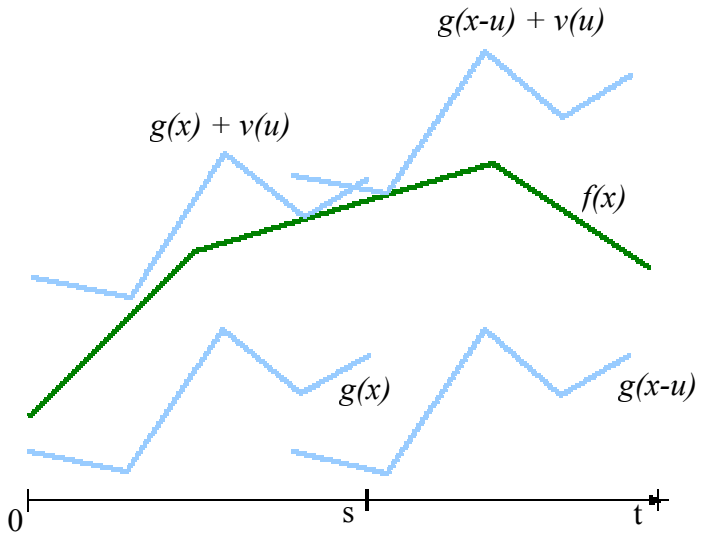


Figure 4-4. Illustration of the problem for proving the non-oriented case

As stated, $f : (0,t) \rightarrow R$ is a piecewise linear function with n linear pieces, and $g : (0,s) \rightarrow R$ is another piecewise linear function with m linear pieces, where $t > s > 0$.

Define $v : (0, s-t) \rightarrow R$ as $v(u) = max_{x \in (u,u+s)}\{f(x) - g(x-u)\}$. It follows that $g(x-u) + v(u) \geq f(x)$ for $\forall x \in (u, u+s)$. It is also clear that if u_0 and v_0 satisfy the two properties in the problem, $v_0 = v(u_0)$.

Let $h_u(x) = g(x-u) + v(u)$, now we need to look for u such that $|| h_u(x) - f(x) ||_1 = \int_{u, u+s} h_u(x) - f(x)\, dx$, is minimized.

Since both $f(x)$ and $g(x)$ are piecewise linear, for any fixed $u \in (0, t-s)$, the distance $d_u(x) = f(x) - g(x-u)$ is also piecewise linear. Therefore, the maximum of $d_u(x)$ can only be taken at the break points located in $[u, u+s]$. This is shown in Figure 4-4 as the vertical dotted lines representing the possible position where the maximum of $d_u(x)$ is taken. That means to compute $v(u)$, only finite number $(< n+m)$ of values of $d_u(x)$ is needed.

In other words, for any $u \in (0, t\text{-}s)$, there is some *break point* x_i of $f(x) : v(u) = f(x_i) - g(x_i - u)$ or there is some break point y_i of $g(x) : v(u) = f(y_i + u) - g(y_i)$. Now let $v_i^f(u) = f(x_i) - g(x_i - u)$, $u \in (x_i - s, x_i)$, $\forall$ break points x_i *of* $f(x)$;

$v_i^g(u) = f(y_i + u) - g(y_i)$, $u \in (0 - y_i, t - y_i) \cap (0 , s)$, $\forall$ break points y_i of $g(x)$,

then $v(u) = max_{i,l \in \{f,g\}}\{v_i^l(u)\}$.

All $v_i^l(u)$ are piecewise linear functions and can be computed quickly, in fact, they are all translations of $f(x)$ or $g(x)$. So, $v(u)$ is also piecewise linear and can be quickly computed.

Now it is clear that $h_u(x) - f(x) = g(x\text{-}u) + v(u) - f(x)$ is piecewise linear on both u and x. Therefore $|| h_u(x) - f(x) ||_1$ is a quadratic spline for u. To get its minimum, one would compute its local minimum on each polynomial piece and then compute the global minimum.

To maximize the max-norm, one can simply compute the max-norm of $h_{ui}(x) - f(x)$ for each break point u_i and compute the minimum of the max-norms.

To minimize the 2-norm of $h_u(x) - f(x)$, the final step becomes to compute the local minimum for each cubic polynomial piece and then compute the global minimum.

This method can also be generalized to solve similar problems of two piecewise linear functions defined in R^2. But extending this proof to the oriented case will involve more computations to find local minimums. This is an open problem.

Our Approach 1: Optimal Placement

At first, a placement technique was built out of a linear programming approach used for the design of molds that is described in deBerg et al. (deB00). The main idea behind the approach was to find the shape of a mold from which the object to be cast

could be extracted. Since different orientations of the object give rise to different molds, the objective was to find a suitable orientation for the object that would facilitate the removal of the object from its mold by a single translation along a direction vector $\vec{d}$. This is possible only if $\vec{d}$ makes an angle of at least 90^0 with the outward normal $\hat{f}$ of all the surfaces on the mold. This is as shown in Figure 4-5.

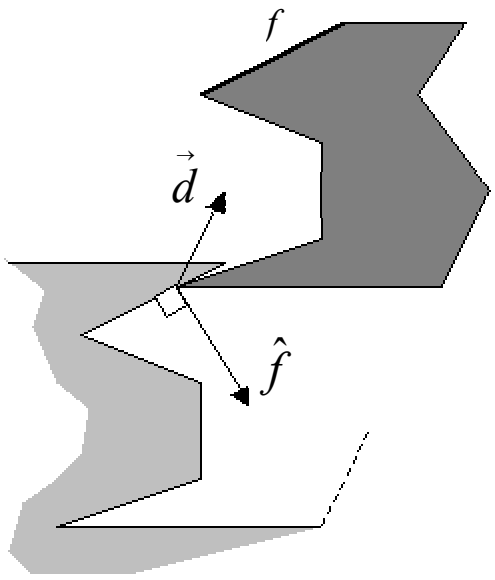


Figure 4-5 Geomtery of Casting (Deb00)

By treating the lower edges lying between the extreme vertices of the pattern as the surface to be cast, and a segment of the profile as a potential mold, the molding making approach was used to find the direction vectors $\vec{d}$ and $\vec{D}$ for the pattern and profile segment respectively as shown in Figure 4-6(b). The segment of the profile that was chosen had to be at least as wide as the lower edges of the pattern. Once these direction vectors were found, the pattern and profile segment were oriented such that their direction vectors were directed along the positive y-axis of the container coordinate system. The oriented pattern was positioned above the profile segment such that it's

lower set of vertices were between the extreme vertices of the profile segment. The pattern was then translated towards the profile segment until it made contact with it. If the pattern was unstable, it was rotated either clockwise or counter clockwise about the first point of contact until it made a second point of contact with the profile segment. The decision of whether to rotate it clockwise or counter clockwise was made based on the location of the centroid with respect to the first point of contact. If the centroid was to the left of the first point of contact, the pattern was rotated counter clockwise in an effort to get a second point of contact to the left of the centroid and vice versa. The pattern was then rotated back by an amount the profile segment was first rotated to get the final position and orientation as shown in Figure 4-6(f). To find a near optimal placement, this procedure had to be executed for n (n being the number of pattern vertices) orientations of the pattern. Each of these pattern orientations had to be sampled against atmost m-3 (m being the number of profile vertices) profile segments making it an O(mn) time algorithm.

The orientations obtained from the direction vectors helped minimize the area between the mating edges of the profile and pattern. This approach looked promising, but it could not be adapted for the packing problem because it had a few serious flaws. The profile segment had to be at least as wide as the lower edges of the pattern in order to cradle the pattern. A reasonable way to find a profile segment that was at least as wide as the pattern and not too wide could not be found as the profile in this case was not monotonic along the x-axis. Finally, the placement search required a considerable amount of computing and this slowed the packing algorithm tremendously. Although this

approach resulted in good placements, it had to be abandoned for a simpler and more efficient one.

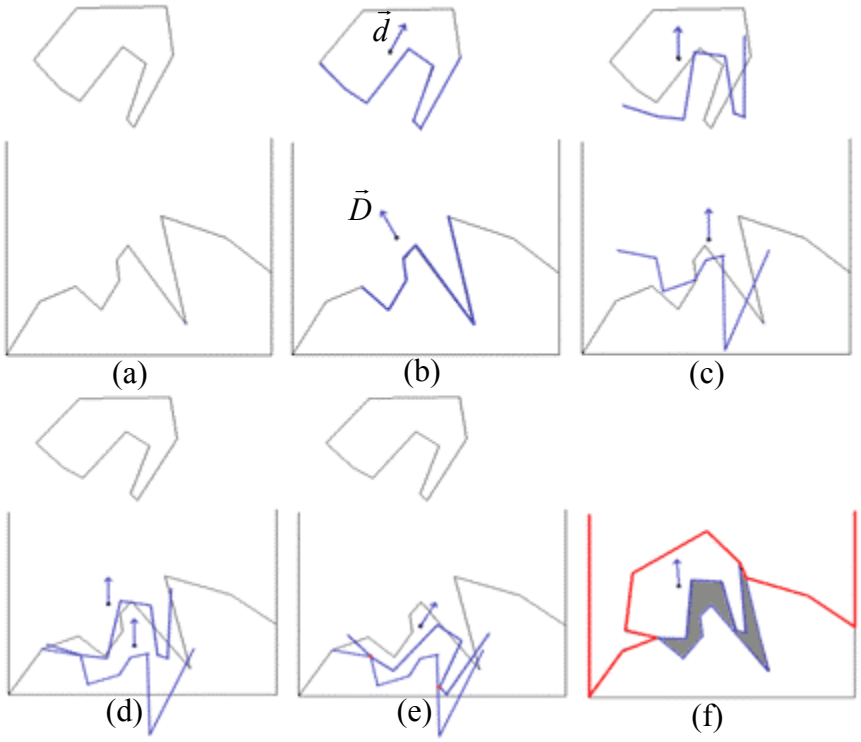


Figure 4-6. Placement heuristic based on the linear programming approach for mold making

Our Approach 2: Optimal Placement

To make the heuristic lightweight, a simple but effective rule had to be adopted. Since no one rule can fulfill all the possible cases that can be encountered for arbitrary shapes, two simple rules that were very different from each other, provided good placements and complemented each other were chosen. The heuristic used these two rules to actually place the patterns in the container and the best placement was then picked from several sample placements.

For each pattern to be placed in the container, PLACE-PATTERN was called once with the pattern *Ptn* and the current profile *Pr* as input. Before the function began sampling placements for the pattern, it ran a profile smoothing routine SMOOTH-

PROFILE that eliminated profile vertices that were coincident, collinear, or subtended a concave angle that was less than half the smallest convex angle on the pattern. This helped speed up the algorithm. SMOOTH-PROFILE ran in linear time. Once the profile was smoothed, several placements were sampled in lines 5-12 using PLACEMENT-RULE1 and PLACEMENT-RULE2. Placements that are contained within the container and are

```
PLACE-PATTERN( Ptn , Pr )

1.  pl ← 0   // initialize placement
2.  bestPl ← 0   // initialize best placement
3.  PE ← ∞   // initialize potential energy

4.  Pr ← SMOOTH-PROFILE( Ptn, Pr )

5.  for i ← 0 to n-1   // n = number of pattern vertices
6.    do for j ←0 to m-1   // m = number of profile vertices
7.      do pl ← PLACEMENT-RULE1( Ptn.vertex(i), Pr.vertex(j) )
8.         if pl.stable() and pl.contained() and pl.energy() < PE
9.           then bestPl ← pl
10.                PE ← pl.energy()

11.        pl ← PLACEMENT-RULE2( Ptn, pr.vertex(j) )
12.        if pl.stable() and pl.contained() and pl.energy() < PE
13.          then bestPl ← pl
14.               PE ← pl.energy()

15. if bestPl ≠ 0
16.   then PLACE-PATTERN(Ptn, Pr, bestPl)
17.        Pr ← GENERATE-NEW-PROFILE(Ptn, Pr)
18. return Pr
```

stable, are recorded if they are found to have a potential energy that is less than that of the best placement *bestPl*. This selection is done in lines 8 and 12.

The quality of a placement is judged by virtue of the pattern's potential energy in that placement. The best placement is one that has the least potential energy among all

the sampled placements. Potential energy *PE* of a placement is given by the following equation.

PE = (pattern area + void area)* height of pattern centroid above container floor

Since the potential energy of a placement increases with void area and the height of the placement, the chosen placement is one that has close to minimum void area and height above the container floor. The inclusion of the pattern area prevents *PE* from going to 0 in cases when there is no void area in the placement. Also, the impact of the void area in the equation is relative to the area of the pattern. When the height of the placement is constant, the growth of PE with the increase in void area is proportional to the pattern area.

If a good placement is found, the pattern in placed using the function PLACE-PATTERN and a new profile that encapsulated the placed pattern was computed using the function GENERATE-NEW-PROFILE. The new profile was generated in linear time and used to place the next pattern.

PLACEMENT-RULE1 paired up convex vertices belonging to the pattern with concave vertices on the profile and vice versa. For each convex vertex on the pattern, the pattern was positioned and oriented above each concave vertex such that the convex pattern vertex was directly above the concave profile vertex and formed the bottom most vertex in the pattern. The pattern was then oriented such that the vector along the inner bisector of the convex angle was parallel to and pointed in the same direction as the vector along the inner bisector of the concave vertex of the profile. This is shown in Figure 4-7(a). The pattern was translated vertically downwards until it made contact with the profile. Just as in the linear programming approach, the pattern was rotated to make a second point of contact with the profile (Figure 4-7(b)(c)). Once the pattern was placed,

the placement was checked for stability. The pattern was considered stable if the two extreme points of contact were on either side of the centroid. PLACEMENT-RULE1 worked well only when the angle of the convex vertex was less than or equal to the angle of the mating concave vertex. Also, placements near the walls of the container tended to intersect the container walls. Figure 4-8 illustrates a case where a concave vertex of the pattern is aligned with a convex profile vertex. Although this placement looks stable, it will be considered unstable by the heuristic because the two points of contact are not on either side of the pattern's centroid.

To over come these drawbacks, PLACEMENT-RULE2 that aligned the patterns with vertical edges in the profile was used. The patterns were rotated such that each edge connecting two adjacent convex hull vertices was made parallel to a vertical edge on the profile and formed the leftmost or rightmost convex hull edge based on whether the vertical edge of the profile was right facing or left facing. This is illustrated in Figure 4-9.

The two rules were found to complement each other well, and if a perfect fit was available for a pattern, the heuristic was capable of finding it. The time that the heuristic took for each placement, depended on the nature of the shape in terms of the number of convex, concave and hull vertices, and the nature of the profile in terms of the number of convex and concave vertices, and number of vertical edges. The worst case running time of the heuristic is O(mn) (m being the number of profile vertices and n being the number of pattern vertices).

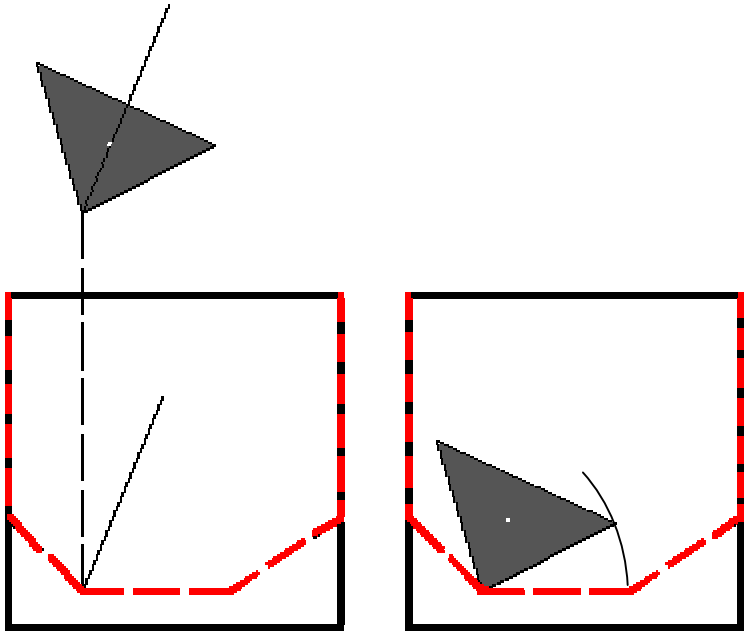


Figure 4-7. First rule in the placement heuristic for a convex vertex in the pattern

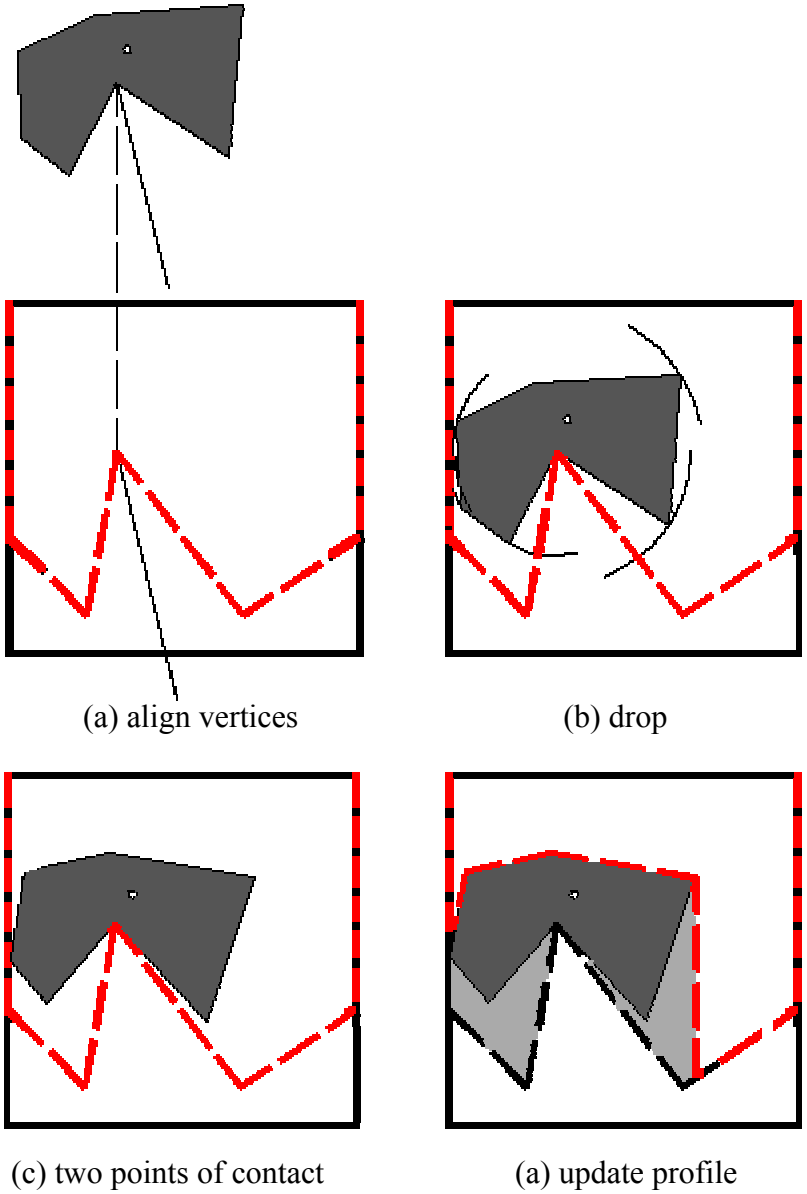


Figure 4-8. First rule in the placement heuristic for a concave vertex in the pattern

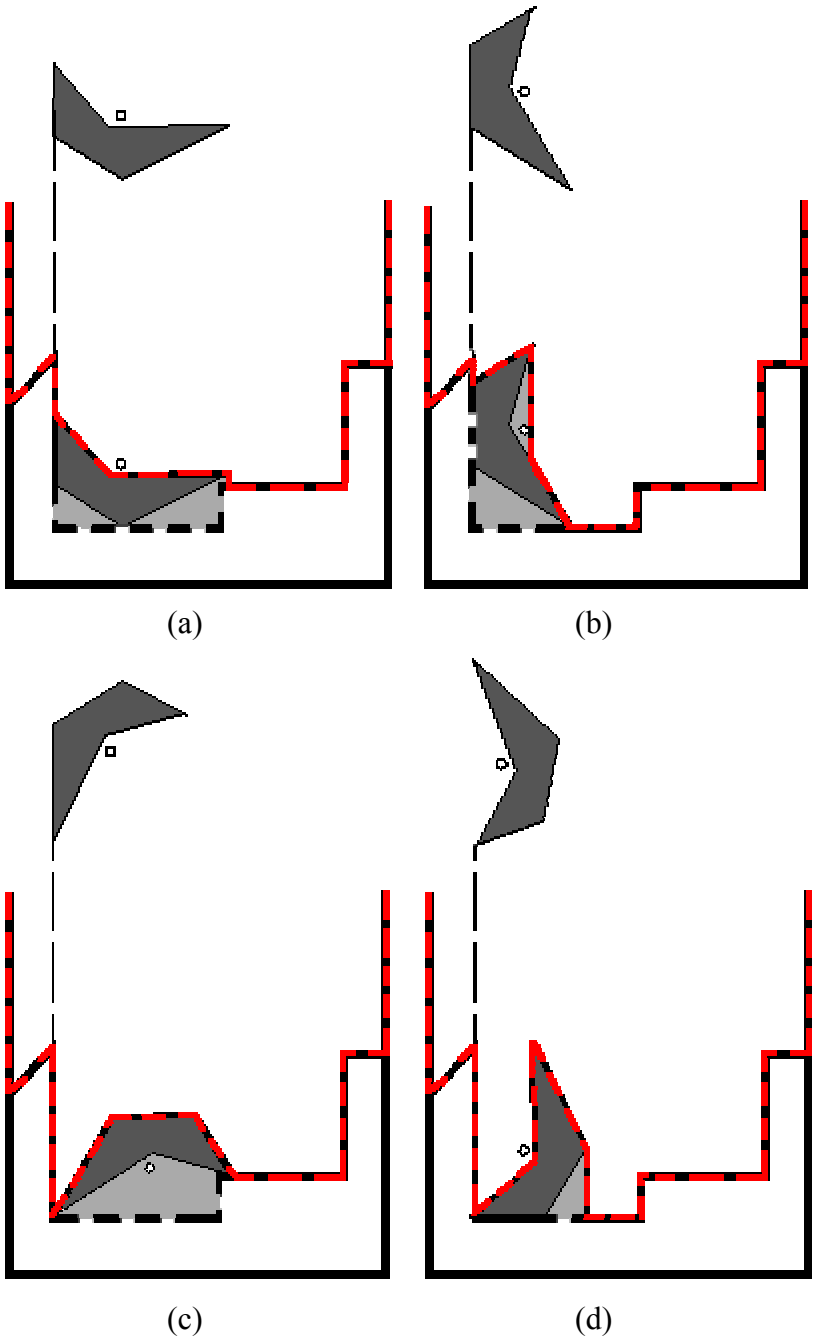


Figure 4-9. Second rule in the placement heuristic

CHAPTER 5
EXPERIMENTAL RESULTS

Choice of GA Parameters

The performance of the algorithm can be greatly enhanced by choosing the right values for the population size, probability of crossover pCross and the probability of mutation pMutation. But, setting these parameters is known to be a difficult task. Haupt (Hau98) suggests various methods for refining these parameters and also states that there is no best way to do it, and that the behavior of the genetic algorithm depends on the problem being solved. A simple iterative procedure was used by running the algorithm on three problem instances that contained shapes for which the order was important in achieving good packing configurations. The population size, was first varied as a function of the input size while keeping pCross and pMutation fixed. The best convergence was obtained when the population size was varied linearly with the input size. pCross and pMutation were then altered in turn and a pCross of 0.6 and pMutation of 0.05 were found to further improve the convergence of the algorithm.

Empirical Analysis

The algorithm was executed for a number of problem instances, some of which were taken from previous papers. Each problem instance was executed 4 times and for 200 generations in each run. The outputs of all four runs for each instance were found to be consistent with each other in terms for the rate of convergence and the quality of the output. Figures 5-1 to 5-5 illustrate the comparisons in the outputs of the algorithm with the outputs presented in previous papers. The shapes of the pattern were copied by hand

and are only approximations of the shapes presented in previous papers. The time that the algorithm took was found to be proportional to the number of patterns being packed, the number of vertices in each pattern and the width of the container. For a set of 10 patterns of comparable size and with 6 vertices per pattern, a container width capable of holding three patterns in a row, the algorithm averaged about 1 second per generation on a Pentium-4 1.8GHz machine.

Irregular Shapes

For input instances containing irregular shaped patterns, the algorithm gave tight packing configurations and this is seen in Figure 5-3 to 5-7. Figures 5-3 to 5-5 show comparisons with outputs of previous papers that contained irregular shapes. The algorithm gave packing configurations that were as optimal as previous algorithms while adhering to all the constraints. Figure 5-6 and 5-7 show the convergence of instances containing only convex patterns and only non-convex patterns. The algorithm performed equally well for both cases.

Geometric Shapes

The algorithm was also run with inputs containing only geometric shapes and was compared to the outputs presented in petridis et al. (Pet98) and Dasgupta et al. (Das97). Figure 5-1 and 5-2 show comparisons for geometric shapes. The outputs of our algorithm did not fair well in comparison to the previous algorithms for two reasons. Firstly, the implementations in Petridis et al. (Pet98) and Dasgupta et al. (Das97) did not consider rotations and thus reduced the problem complexity significantly. Secondly, the heuristic is unable to make a global decision between two or more locally optimal placements. There is a possibility of two or more distinct chromosomes encoding the same solution

when the difference between them is small and restricted to neighboring genes as shown below.

0 1 2 3 4 5 6

0 1 3 2 4 5 6

The two chromosomes are similar except for the third and fourth genes that are swapped. The placement heuristic may output the same packing configuration for both chromosomes. This form of redundancy can grow exponentially with the size of the input and reduce the effectiveness of the algorithm. But, the possibility of this happening for a irregular and unique set of shapes is rare.

Genetic Algorithm Drawbacks

When the input set contains patterns of identical shape, the problem of redundancy becomes more significant. As the percentage of identical or duplicate shapes increase in the input set, the genetic algorithm gets increasingly ineffective and is actually rendered completely ineffective when all the patterns in the input set have a similar shape. For such input cases, the algorithm depends solely on the power of the placement heuristic as the order of placement ceases to play a role in the optimization process. This is seen for problem instances shown in Figures 5-1, 5-2 and 5-3. These instances contain duplicate shapes and the algorithm takes relatively more time to converge.

From the experimental runs that were performed, it was found that the algorithm does not give better results beyond a certain point. Therefore, a good termination condition for the algorithm would be to stop when there is no improvement for a user defined number of generations. Since the algorithm updates the graphics with each improvement, the user may also terminate it once a satisfactory result is obtained.

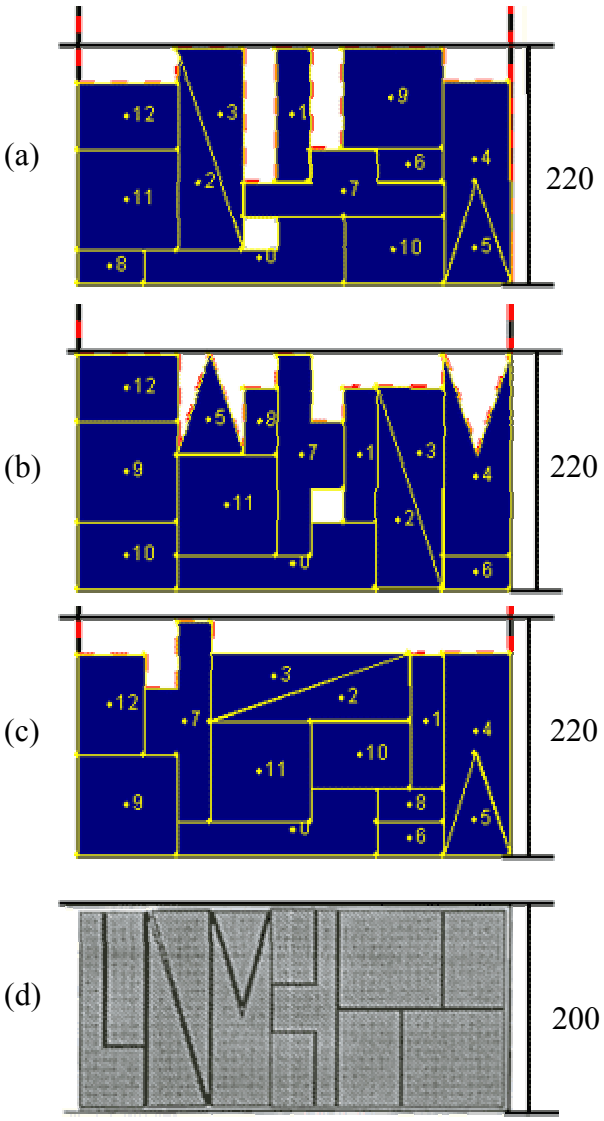


Figure 5-1. Arrangement of 13 geometric patterns (a) 12th generation, (b) 39th generation, (c) 47th generation, (d) Output from Petridis et al. (Pet98)

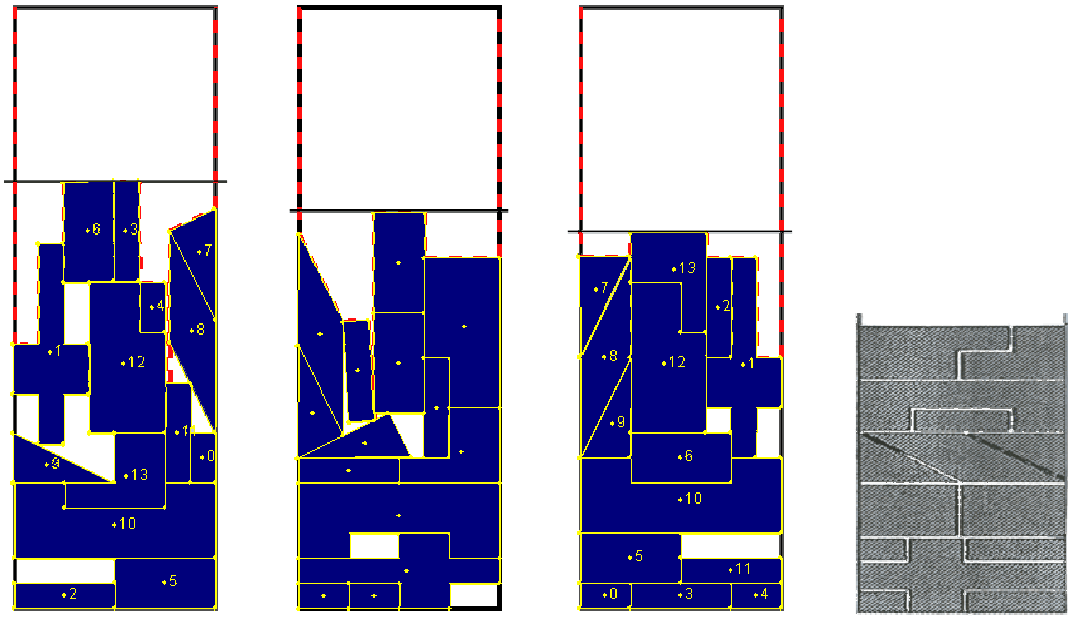


Figure 5-2. Arrangement of 14 geometric patterns (a) 2nd generation, (b) 36th generation, (c) 84th generation, (d) output from Watanabe and Ono (cited in Das97)

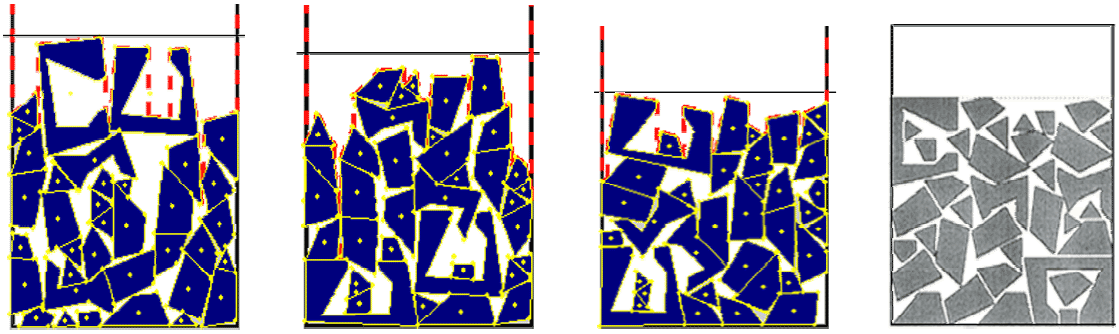
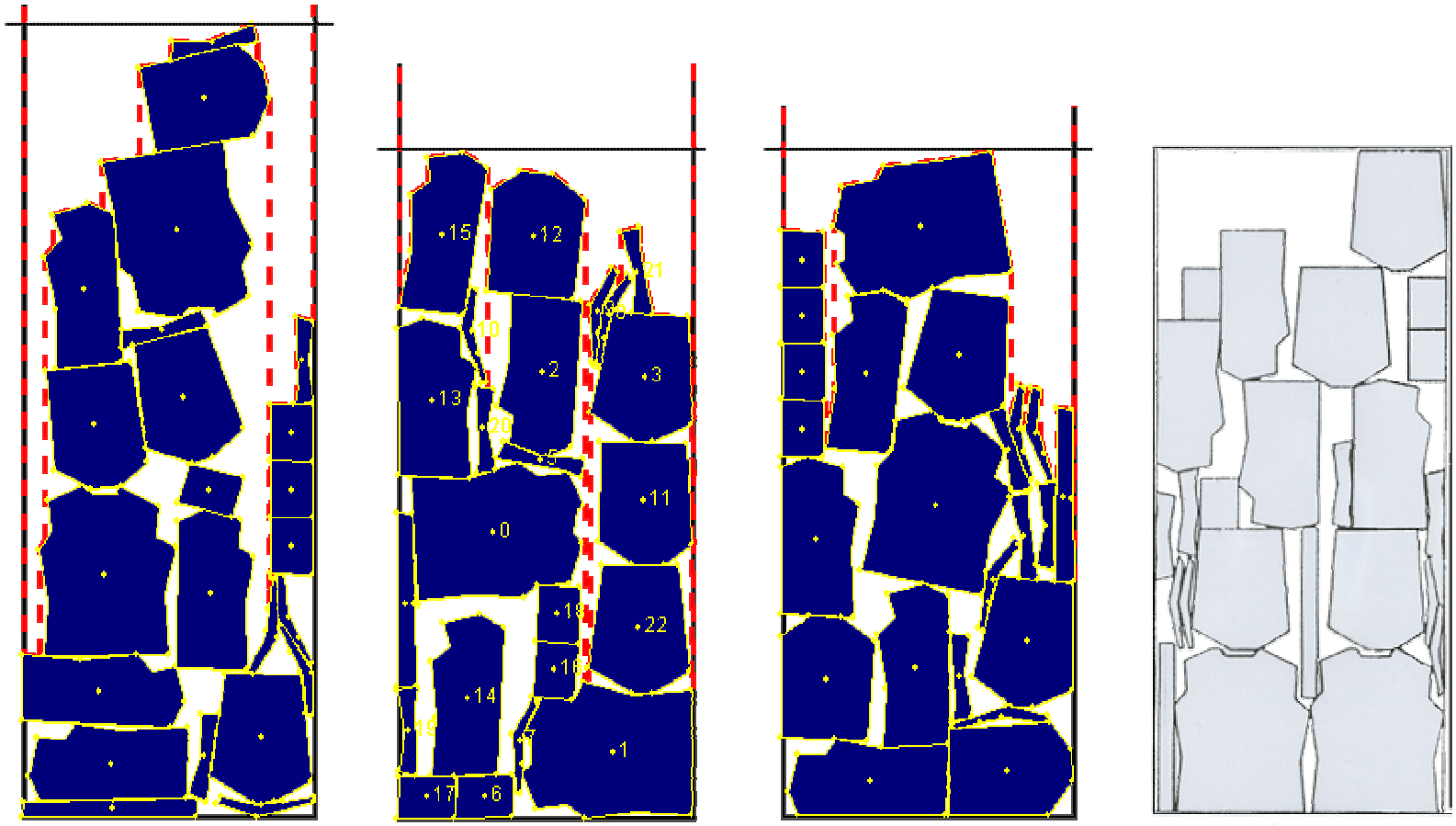
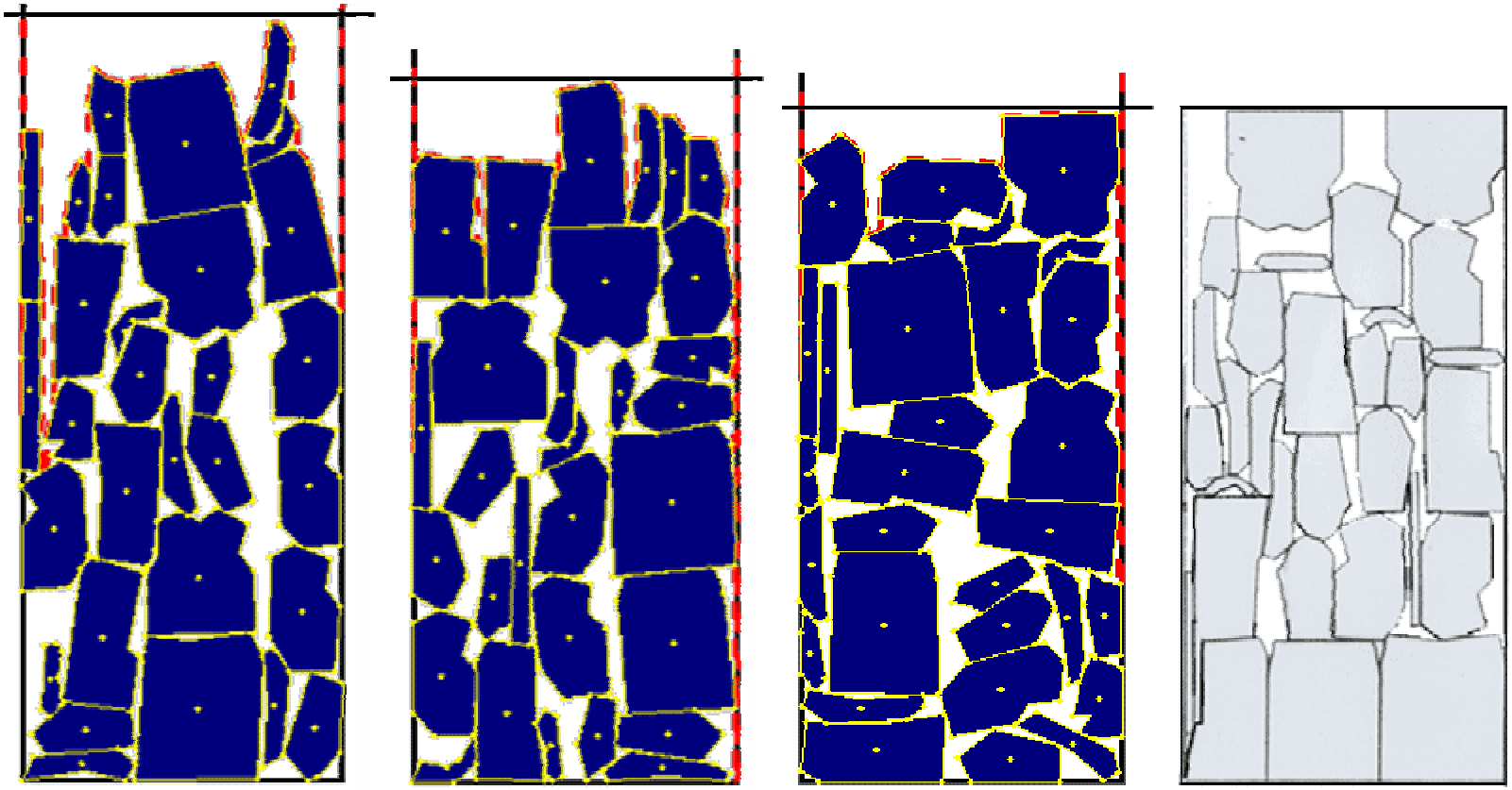


Figure 5-3. Arrangement of 36 irregular patterns (a) 1st generation, (b) 16th generation, (c) 40th generation, (d) output from Watanabe and Ono (cited in Das97)



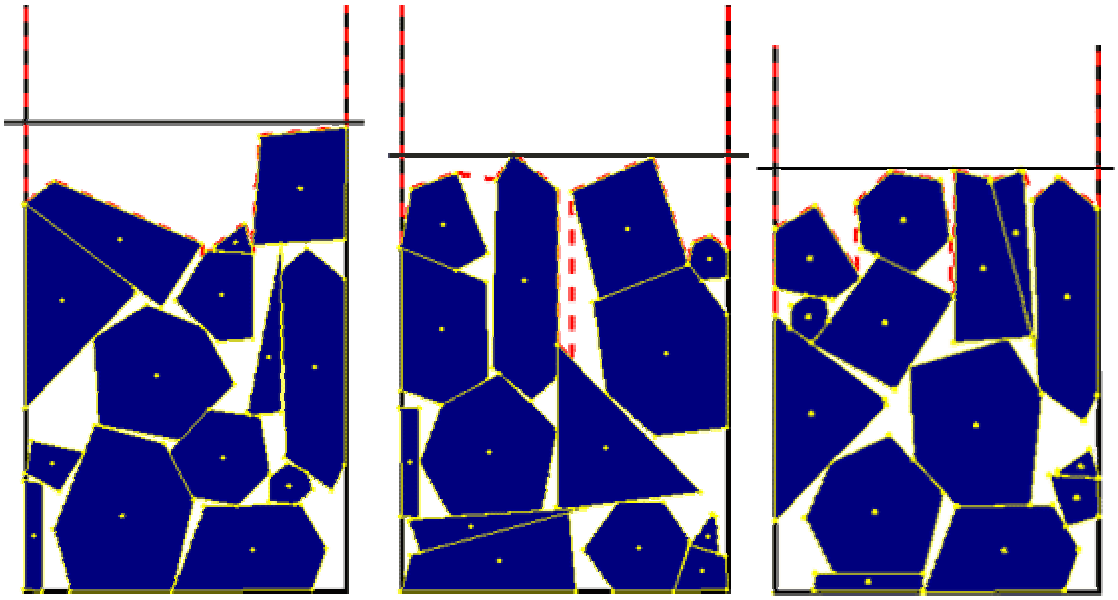
(a) (b) (c) (d)

Figure 5-4. Arrangement of 24 irregular patterns (a) 1st generation, (b) 5th generation, (c) 26th generation, (d) output from Albano and Supoppo (Alb80)



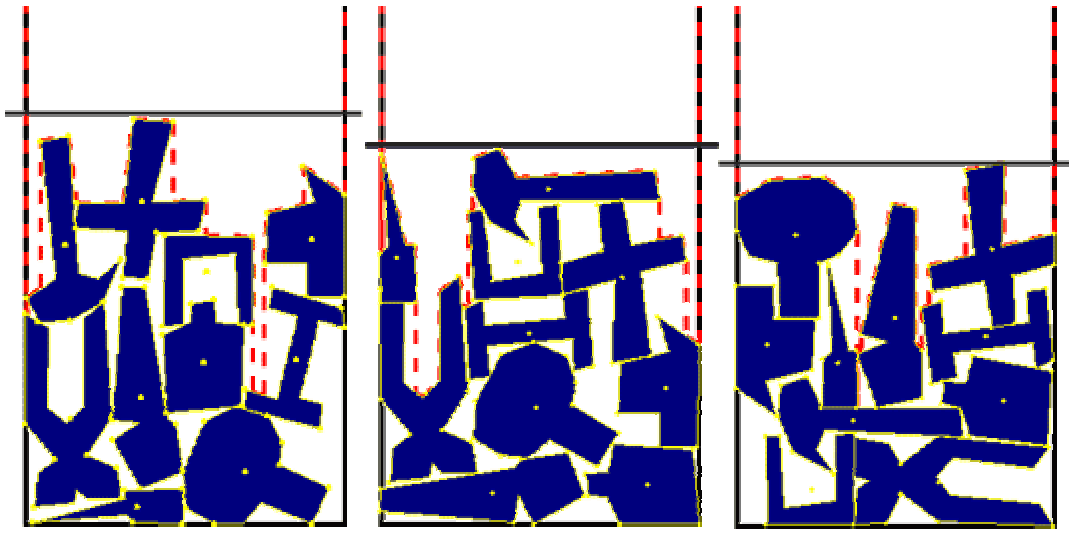
(a) (b) (c) (d)

Figure 5-5. Arrangement of 30 irregular patterns (a) 2nd generation, (b) 9th generation, (c) 22nd generation, (d) output from Albano and Supoppo (Alb80)



(a) (b) (a)

Figure 5-6. Arrangement of 14 irregular convex patterns (a) 2nd generation (b) 7th generation (c) 25th generation



(a) (b) (a)

Figure 5-7. Arrangement of 10 irregular non-convex patterns (a) 1st generation (b) 8th generation (c) 39th generation

Placement Heuristic Drawbacks

The heuristic implemented in this paper runs in linear time and has some amount of redundancy built into it. This redundancy helps in finding near optimal local placements regardless of the shape of the patterns. Placements that are sampled by the heuristic are found to be at or near the break points present along the profile. But, it suffers from one drawback. As the patterns are packed, the profile begins to develop narrow valley like regions. The heuristic does not find the placements directly above these regions to be good placement even though they may be better in terms of reducing the container height. This is because the heuristic settles for a placement whose potential energy is the least among all the sampled placements, and the void area created by each placement is considered in the calculation of the potential energy. The area in the valley region becomes part of the void area and reduces the potential energy of any placement above the valley. Figure 5-8 shows an instance of this problem. The alternate placement in the figure is not considered as a good placement because its potential energy is reduced due to the void area created in the valley. The effect of this problem can also be seen in the results shown in Figure 5-4 and Figure 5-5 where the patterns seem to be stacked directly on top of each other with longitudinal gaps between them. Not considering the void area in the calculation of the potential energy of a placement is not the solution to the problem because it results in a lot of wasted space below the placements. The problem can be eliminated by improving the profile smoothing function.

The algorithm only minimizes the effect of slippage by searching for placements that are stable. Slippage may occur when objects are placed into the container due to factors such as low coefficient of friction at contact points. Checking for slippage is computationally too expensive for the heuristic. These drawbacks may be fixed with the

help of a vision system that can detect a change in the real profile when compared to its virtual counterpart after each part is packed. If a change occurs, the packing algorithm may be executed once again for the unpacked parts and the real profile. The flexibility provided in this manner makes the algorithm more adaptable to changes in the real environment.

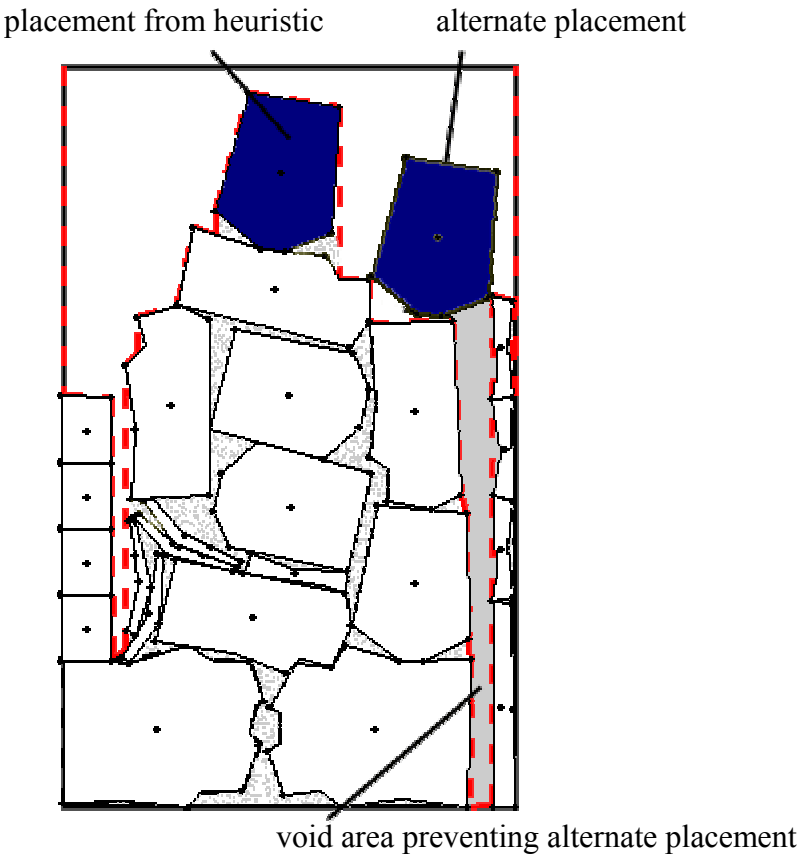


Figure 5-8. A drawback in the placement heuristic

Conclusion

There is a lot of scope in further improving the heuristic to produce better local placements. The approach used in this paper is simple and may be mapped into three dimensions. The hybrid nature of the algorithm allows it to be optimized against multiple constraints such as minimization of the center of gravity of the container, order of

removal and pressure constraints of the objects. It also has the flexibility of being used for both online and offline packing.

The algorithm, therefore is capable of finding near optimal packing configurations for a set of arbitrary shapes, capable of optimizing against multiple constraints, is flexible enough to adapt to real world changes, is general enough to be applied to other packing applications and integrates well with an autonomous packing system.

CHAPTER 6
FUTURE WORK

From the inference gained by implementing the packing algorithm in two dimensions, a three dimensional packing algorithm may be implemented more effectively. The algorithm presented in this paper may be extended to three dimensions with the help of an edge based boundary representation data structure for the objects and the profile. The two dimensional profile would map onto a surface. A container with a fixed width and depth but with an infinite height may be used with the same heuristic and genetic algorithm objective function to yield similar results. The methodology used in searching for near optimal placements may also be used in three dimensions.

There is a lot of scope for improving the heuristic either to make it find local optimal placements, or placements that will lead to better global solutions. The genetic algorithm may also be improved by making it adaptive. Petridis et al. (Pet98) suggests a method that varies the fitness function dynamically and shows how it can improve the convergence time for the algorithm.

Multi-bin packing may be achieved by making the heuristic pack multiple bins for each chromosome. The objective function of the genetic algorithm would then have to be changed from minimizing the packed height of the bin to minimizing the number of bins used.

The algorithm finds tight packing configurations for the patterns without considering the center of gravity of the packed container. If the patterns (or objects in three dimensions) are made of different materials that vary in density, a tight packing

configuration may not mean that the center of gravity has been minimized. This is a result of the second assumption that was made about the nature of the patterns being packed. The algorithm though, may be changed to overcome this drawback by considering the density of the patterns when the potential energy of each placement is calculated by the placement heuristic.

APPENDIX A
DOCUMENTATION

Java 1.3 was chosen for the implementation of the algorithm since it is well suited for creating rapid prototypes. The language also provides extensive support for 2D graphics, utilities such as random number generators and basic data structures.

The code was split into three packages – genAlgthm, gui and packingDataStruct described in Table 1 below. Each package and their contents are detailed in the tables that follow.

Table 1. Packages contained in the implementation of the packing algorithm

Package	Descriptiom
gui	Contains classes that build the graphical user interface.
packingDataStruct	Contains the data structure used to describe the patterns and container.
genAlgthm	Contains code for the genetic algorithm.

Table 2. Classes contained in package genAlgthm

Class	Description
Chromosome	Implements a chromosome as an array of integers that contains the packing order of the list of patterns. Each chromosome is associated with a fitness value and holds the pose information of the patterns in the part list when the patterns are packed in the order defined by the chromosome.
Population	Implements a population of chromosomes and the genetic algorithm.

Table 3. Classes contained in the package gui

Class	Description
DataPanel	A panel to the east of the main window that contains global data of the packing algorithm. The data panel is updated frequently during the execution of the algorithm. If the mouse pointer is moved over a pattern, data associated with the pattern in displayed in the data panel.
DrawPanel	A panel in the center of the main window that can be used to draw two-dimensional shapes for the patterns and the container. The panel is equipped all the available mouse handlers. Patterns are drawn by single clicking the mouse at the desired vertices of the pattern. Clicking near the first vertex closes the pattern. The container is drawn by single clicking the mouse either of the two diagonal ends of the container. A pattern that is being drawn may be canceled by clicking the right mouse button. The draw panel is refreshed frequently during the execution of the algorithm to show the most recent optimal solution obtained by the algorithm.
FileHandler	This class contains code that handles the file input/output operations for the implementation.
Main	Main contains the main method for running the packing program. Typing “java gui.Main” from outside the gui directory on a console or shell window starts the program. Main extends JApplet and can be run as a standalone program or as an applet.
MenuBar	This class contains code for the menu bar situation to the north of the main window.
FileFilter	A convenience implementation provided by Sun Microsystems, that filters out all files except for those type extensions that it knows about from the FileChooserDialog box. Extensions are of the type ".dpk" for the packing algorithm. Case is ignored.
StatusPanel	A panel to the south of the main window that gives the user status information, tool tips and error/warning messages while the user interacts with the program.

Table 4. Classes contained in the package packingDataStruct

Class	Description
BasicTests	A set of class methods containing code for basic computational geometry operations.
Container	A top level data structure that describes the container. Container has an inner class Profile that describes the profile of the container and the operations associated with the profile.
ConvexHull	Contains the implementation of Graham's algorithm for finding the convex hull of a polygon. This class implements the comparable interface in Java.
Heuristic	Implementation of the Placement Heuristic.
Part	Data structure that describes the pattern, its attributes and the operations associated with it. Part extends Vertex.
PartList	Data structure used to hold a list of parts or patterns.
Pose	Defines the position *(x,y)* and orientation *theta* of a geometric entity in the graphic coordinate system.
Vertex	Describes a vertex, its attributes and all the operations performed on it.

Table 5. Attributes and methods contained in genAlgthm.Chromosome

Field Summary	
public double *fitness*	Fitness value of the chromosome.
protected int [] *gene*	Set of genes making up the chromosome.
protected Pose [] *pose*	Array containing pose information for the list of patterns packed in the order defined by the chromosome.
Constructor Summary	
Chromosome (Chromosome *chromo*)	Creates a new chromosome that is identical to the input chromosomes *chromo*.
Chromosome (int *size*)	Creates a new chromosome of size *size* and initializes it with a random set of genes.
Method Summary	
public int [] getChromosome()	Returns the chromosome as an array of integers.
public int getGene(int *index*)	Returns the gene with the specified index.
public void setGene(int *index*, int *value*)	Sets the value of a gene with the specified index.
public void setPose(int *index*, Pose *pose*)	Sets the pose of the pattern with the specified index.
public int size()	Returns the size of the chromosome.
public void swap (int *index1*, int *index2*)	Swaps two genes in the chromosome.
public java.lang.String toString()	Returns the string representation of the chromosome.

Table 6. Attributes and methods contained in genAlgthm.Population

Field Summary	
protected double *avgFitness*	Average fitness of the population.
private Container *cc*	Container.
private java.util.Random *cRand*	Random number generator for the crossover operator.
Private DrawPanel *drawPanel*	Reference to the drawing panel of the graphical user interface.
protected Chromosome *fittestChromo*	Deep copy of the fittest chromosome.
private int *genCtr*	Generation counter.
private Heuristic *ht*	Reference to an instance of the placement heuristic.
protected double *maxFitness*	Maximum fitness value in the entire population for all generations.
private static final int *MAXGEN*	Maximum number of generations that the genetic algorithm will run before terminating automatically.
private int *popSize*	Maximum size of the population.
protected double *minFitness*	Minimum fitness value in the entire population for all generations.
private java.util.Random *mRand*	Random number generator for the mutation operator.
private int *numCross*	Holds the number of crossovers performed.
private int *numGenes*	Number of genes in a Chromosome.
private int *numMutatation*	Holds the number of mutations performed.
private static final int *PCROSS*	Probability of crossover.
private static final int *PMUTATION*	Probability of mutation.
protected Chromosome [] *pop*	Population of chromosomes.
private java.util.Random *rand*	Random number generator for generating a Random sequence of genes in each Chromosome.
protected double *sumFitness*	Sum of the fitness values of all the chromosomes in the population.
Constructor Summary	
Population (DrawPanel *drawPanel*)	Creates a new population of chromosomes based in the input information obtained from the reference to the draw panel. Initializes each chromosome with a random set of genes. The size of the population is equal to the number of patterns that need to be packed into the container.

Table 6. Continued

Method Summary	
private void crossover(int [] *p1*, int [] *p2*)	Performs a crossover operator on the two parents *p1* and *p2* with a probability PCROSS. The function then replaces the parents with the off springs.
private void decode(Chromosome *chromo*)	Computes the fitness value of a chromosome by running the placement heuristic on the chromosome.
public void draw()	Draws the packing configuration of the fittest chromosome.
private void generation()	Executes one complete generation of the genetic algorithm.
private static int indexOf(int [] *pp*, int *elem*)	Returns the index of *elem* in the array *pp* or –1 if *elem* is not found in the array.
private void mutation(Chromosome *chromo*)	Performs a random mutation on the input chromosome with the probability PMUTATION.
public void run()	Begins the execution of the genetic algorithm.
private void select()	Selects two parents randomly from the population of chromosomes for mating. The selection is done on the roulette wheel method.
public java.lang.String toString()	Returns a string representation of the population.

Table 7. Attributes and methods contained in gui.DataPanel

Constructor Summary	
DataPanel()	Creates an instance of the data panel.
Method Summary	
public static void writeGlobalData(DrawPanel *drawPanel*)	Writes global data to the draw panel. The data written includes max fitness and average fitness of the population, a measure of time complexity of the heuristic, number of crossovers, number of mutations and the time taken by the algorithm.
public static void writePartData(Part *part*)	Writes part data of the given part. Data includes the area of the part, location of centroid and index of the part.

Table 8. Attributes and methods contained in gui.DrawPanel

Field Summary	
protected Container *container*	Reference to the container to be packed.
private Part *copyPart*	Placeholder for a copied part when the edit | copy command in the menu bar is used.
private int *drawMode*	Drawing mode of the draw panel. The following modes are defined for *drawMode,* 0 draw container 1 draw pattern 2 delete selected pattern 3 move selected pattern 4 copy selected pattern 5 select a pattern
private Vertex *moveFrom*	Location from which the selected pattern must be moved. Used when the edit | move command is used from the menu bar.
private boolean *pauseGA*	Set to true is the used pauses the genetic algorithm.
private java.util.Vector *polygon*	A list of vertices that are temporarily stored as a pattern is drawn.
private boolean *showGrid*	A grid is displayed when *showGrid* is set to true. s*howGrid* is set to true when the user selected the edit | show grid option from the menu bar.
private boolean *snap*	The mouse pointer snaps to the closest grid point when this option is switched on from the edit | snap option in the menu bar.
private java.awt.Rectangle *snapRect*	A square with dimensions 20x20 pixels that is used to close the polygon.
private static final int *SNAPSIZE*	Resolution of the snap grid.
private boolean *stopGA*	Set to true if the user chooses to stop the genetic algorithm from run | stop GA in the menu bar.
Constructor Summary	
DrawPanel()	Creates an instance of the draw panel.
Method Summary	
private void init()	Initializes the draw panel.
public void paint(java.awt.Graphics *g*)	Paints graphic entities such as container, patterns etc onto the draw panel.
public void repaint()	Repaints the draw panel.
public void reset()	Initializes the draw panel.
protected void setDrawMode(int *mode*)	Sets the draw mode of the draw panel.
private void this_mouseClicked (java.awt.event.MouseEvent *e*)	Does nothing.

Table 8. Continued

Method Summary	
private void this_mouseDragged (java.awt.event.MouseEvent *e*)	Does nothing.
private void this_mouseEntered (java.awt.event.MouseEvent *e*)	Records the coordinates of the mouse when the draw mode is 5.
private void this_mouseExited (java.awt.event.MouseEvent *e*)	Does nothing.
private void this_mouseMoved (java.awt.event.MouseEvent *e*)	Does nothing.
private void this_mouseReleased (java.awt.event.MouseEvent *e*)	Registers the coordinates of the mouse click as either a selection coordinate or a vertex belonging to the pattern or container based on the drawing mode.
private void update(java.awt.Graphics *g*)	Updates the draw panel.

Table 9. Attributes and methods contained in gui.FileHandler

Field Summary	
private java.lang.String *filename*	File name.
Constructor Summary	
FileHandler()	Creates a file handler for an unspecified file.
FileHandler(java.lang.String *fileName*)	Creates a file handler for the file with name *fileName*.
Method Summary	
protected java.lang.String open()	Opens the file associated with this file handler and returns the contents of the file. Returns null if the file is not found or the file is not of the right format.
protected boolean save(java.lang.String *contents*)	Saves the given contents to the file with name *fileName.* If file name is not specified, the file chooser dialog box is displayed. Returns true is the save operations was successful.
protected boolean saveAs(java.lang.String *contents,* java.lang.String *fileName*)	Similar to the method save, but saves the given contents to the given file name.

Table 10. Attributes and methods contained in gui.Main

Field Summary	
private static boolean *isStandAlone*	Set to true is the program in being run as a stand-alone program. Set to false is the program is being run as an applet.
Constructor Summary	
Main()	Creates an instance of the program when the program is being run as a standalone program.
Method Summary	
public void destroy()	Overrides the destroy method in the super class.
public java.lang.String getParameter(java.lang.String *key*, java.lang.String *def*)	Gets applet information from the param tags in the HTML file that contains the applet.
public void init()	Initializes the applet.
public static boolean isStandalone()	Returns true is the program is being run as a stand alone program.
public static void main(java.lang.String [] *args*)	Main method of the packing program.
public void start()	Overrides the start method in the super class.
public void stop()	Overrides the stop method in the super class.

Table 11. Attributes and methods contained in gui.MenuBar

Field Summary	
private javax.swing.JRadioButtonMenuItem *container*	Set to true if the program is being run as a stand-alone program. Set to false if the program is being run as an applet.
private javax.swing.JMenu *display*	Pull down menu for display options.
private javax.swing.JMenu *draw*	Pull down menu for drawing options.
private DrawPanel *drawPanel*	A reference to the draw panel.
private javax.swing.JMenu *edit*	Pull down menu for editing options.
private Population *pop*	A reference to the population.
private javax.swing.JMenu *file*	Pull down menu for file handling options.
private FileHandler *fileHandler*	A reference to the file handler.
private javax.swing.JRadioButtonMenuItem *grid*	A radio button option in the edit menu for displaying a grid on the draw panel. The grid has a resolution of 20x20 pixels.
private javax.swing.JMenuBar *menuBar*	Menu bar.
private javax.swing.JMenuItem *menuItem*	A handle for a menu item.
private javax.swing.JRadioButtonMenuItem *parts*	A radio button option in the draw menu for toggling between drawing modes for drawing the container and parts.
private javax.swing.JMenu *run*	Pull down menu with various options for executing the packing program.
private javax.swing.JRadioButtonMenuItem *snap*	A radio button option in the edit menu. When snap is switched on, the each mouse click is set to the nearest grid point.
Constructor Summary	
MenuBar(DrawPanel *drawPanel*)	Creates an instance of the program when the program is being run as a standalone program.
Method Summary	
public void actionPerformed(java.awt.event.ActionEvent *e*)	Event handler for the menu bar.
public DrawPanel drawPanel()	Returns the menu bar's handle to the draw panel.
public javax.swing.JMenuBar getMenuBar()	Returns a reference to the menu bar.
public void itemStateChanged(java.awt.event.ItemEvent *e*)	Event handler for the radio button menu items in the menu bar.

Table 12. Attributes and methods contained in gui.FileFilter

Field Summary	
private static java.lang.String TYPE_UNKNOWN	
private static java.lang.String HIDDEN_FILE	
private java.util.HashTable *filters*	
private java.lang.String *description*	
private java.lang.String *fullDescription*	
private boolean *useExtensionsInBoolean*	
Constructor Summary	
FileFilter()	Creates a file filter. If no filters are added, then all files are accepted.
FileFilter(java.lang.String *extension*)	Creates a file filter that accepts files with the given extension. Example: new FileFilter("dpk")
FileFilter(java.lang.String *extension*, java.lang.String *description*)	Creates a file filter that accepts the given file type. Example: new FileFilter("dpk", "bin packing files ") Note that the "." before the extension is not needed. If provided, it will be ignored.
Method Summary	
public boolean accept(File *f*)	Return true if this file should be shown in the directory pane, false if it shouldn't.
public void addExtension(java.lang.String *extension*)	Adds a filetype "dot" extension to filter against. For example: the following code will create a filter that filters out all files except those that end in ".dpk" : FileFilter filter = new FleFilter(); filter.addExtension("dpk"); Note that the "." before the extension is not needed and will be ignored.
public java.lang.String getDescription()	Returns the human readable description of this filter. For example: "Bin Packing files (*.dpk)"
public java.lang.String getExtension(File *f*)	Return the extension portion of the file's name.
public boolean isExtensionListInDescription()	Returns whether the extension list (.jpg, .gif, etc) should show up in the human readable description. Only relevent if a description was provided in the constructor or using setDescription().

Table 12. Continued

Method Summary	
public void setExtensionListInDescription(boolean *b*)	Determines whether the extension list (.jpg, .gif, etc) should show up in the human readable description. Only relevent if a description was provided in the constructor or using setDescription().
public void setExtensionListInDescription(boolean *b*)	Determines whether the extension list (.jpg, .gif, etc) should show up in the human readable description. Only relevent if a description was provided in the constructor or using setDescription().

Table 13. Attributes and methods contained in gui.StatusPanel

Field Summary	
private DrawPanel *drawPanel*	A reference to the draw panel.
private static javax.swing.JtextArea *tArea*	Text area where status messages are written.
private static javax.swing.Jlabel *xCood*	Label to display the x-coordinate of the mouse pointer on the draw panel.
private static javax.swing.Jlabel *yCood*	Label to display the y-coordinate of the mouse pointer on the draw panel.
Constructor Summary	
StatusPanel(DrawPanel *drawPanel*)	Creates an instance of the status panel.
Method Summary	
protected static void setCoordinates(int *x*, int *y*)	Sets the given coordinates on the status panel.
protected static void write(java.lang.String *status*)	Writes the given status message on the status panel.

Table 14. Attributes and methods contained in packingDataStruct.BasicTests

Field Summary	
private static double *TOL*	Tolerance used for floating point inequality checks.
Method Summary	
public static double angle(Vertex *v0*, Vertex *v1*)	returns the angle in radians subtended at *v0* by a line passing through the two points and the positive x-axis. Note: CCW angle returned if origin is in the top left corner of the screen. CW angle returned if origin is in the bottom left corner of the screen.
public static double angle(Vertex *v0*, Vertex *v1*, Vertex *v2*)	Returns the acute angle (in radians) subtended by the three vertices at the middle vertex *v1*.
public static double area(java.util.Vector *vertexList*)	Returns the area bounded by the list of vertices that define a closed polygon.
private static int areaSign(Vertex *v0*, Vertex *v1*, Vertex *v2*)	Returns the signed area of the triangle defined by the three vertices in the order *v0,v1,*and *v2*.
public static boolean between(Vertex *v0*, Vertex *v1*, Vertex *v2*)	Returns true if the vertex *v1* lies within or along the edges of the bounding box defined by *v0* and *v2*.
public static int circleLineIntersect(Vertex *center*, double *radius*, Vertex *v0*, Vertex *v1*, Vertex [] *intersect*)	Finds the intersection of a circle with the given center and radius, and a line segment defined by *v0v1*. The point(s) of intersection are returned through a 2-element array of vertices that is passed to the function through the parameter list. Returns, 0 if the line segment does not intersect the circle. 1 if the line segment is tangential to the circle. 2 if the line segment intersects the circle once. 3 if the line segment intersects the circle twice.
public static boolean collinear(Vertex *v0*, Vertex *v1*, Vertex *v2*)	Returns true if the vertex v2 is collinear to the straight line.
public static Vertex computeCentroid(Part *part*)	Computes the centroid of a part.
private static double crossProduct (Vertex *v0*, Vertex *v1*)	Returns the cross product of two vertices.
public static boolean equalsTo(double *a*, double *b*)	Returns true if a == b within a tolerance of TOL. Else it returns false.

Table 14. Continued

Method Summary	
public static double getDistance(Vertex *v0*, Vertex *v1*)	Returns the absolute distance between the given vertices.
public static boolean isClockwiseOriented (java.util.Vector *vertexList*)	Returns true if the vertex list is oriented clockwise. Else it returns false.
public static char linesIntersect(Vertex *v0*, Vertex *v1*, Vertex *v2*, Vertex *v3*, Vertex *intersect*)	Checks to see if two line segments defined by *v0v1* and *v2v3* intersect. If they intersect, the function returns the point of intersection through *intersect* in the parameter list. Returns, 'e': The segments collinearly overlap, sharing a point 'v': An endpoint (vertex) of one segment is on the other segment, but 'e' doesn't hold '1': The segments intersect properly (i.e., they share a point and neither 'v' nor 'e' holds) '0': The segments do not intersect (i.e., they share no points)
public static char pointInPoly(Part *part*, Vertex *v0*)	Checks to see if the given vertex lies within or along the edges of the part. Returns, 'i' : v0 is strictly interior to part 'o' : v0 is strictly exterior to part 'v' : v0 is a vertex of part 'e' : v0 lies on the relative interior of an edge of part
public static double pointLineDistance(Vertex *v0*, Vertex *v1*, Vertex *v2*, Vertex *intersect*)	Returns the perpendicular distance between the line defined by vertices *v0v1* and the vertex *v2*. The point of contact of the perpendicular with the line is returned through *intersect* in the parameter list.
public static void reverseVector(java.util.Vector *vertexList*)	Reverses the order of the given list of vertices.
public static double [] sortWithoutDup(double [] *array*, int *begin*, int *end*)	Sorts and array of doubles and returns the sorted array after removing duplicates between the limits *begin* and *end*.
public static int turns (Vertex *v0*, Vertex *v1*, Vertex *v2*)	Determines whether two consecutive line segments *v0v1* and *v1v2* form a left turn, a right turn or they are collinear. Returns, 1 if it is a left turn2 if it is a right turn 0 if the lines are collinear

Table 15. Attributes and methods contained in packingDataStruct.Container

Field Summary	
private double *area*	Area of the container.
private double available *area*	Area enclosed by the profile and the top edge of the container.
private Vertex *centroid*	Centroid of the container.
private java.awt.geom.Rectangle2D.Double *container*	Container.
private DrawPanel *drawPanel*	A reference to the draw panel.
private Vertex *origin*	Origin of the container. The lower left corner of the container in graphic coordinates.
private partList *partList*	List of parts that need to be packed.
private Container.Profile *profile*	Profile of the container.
Constructor Summary	
Container(Vertex *v0*, Vertex *v1*)	Creates a container with the given diagonal coordinates *v0* and *v1*.
Container(DrawPanel *drawPanel*)	Creates a container object with the information input by the user through the draw panel.
Container(java.lang.String *container*)	Creates a container object from a string representation of the container.
Method Summary	
public void addPart(Part *part*)	Adds a part to the container list of parts.
public Vertex origin()	Returns the origin of the container, i.e. the bottom left corner in graphic coordinates.
public double area()	Returns the area of the container.
public double availableArea()	Returns the area enclosed by the container profile and the top edge of the container.
public boolean contains(Rectangle2D.Double *bb*)	Returns true if the given bounding box *bb* is fully contained within the container (or at least touching the walls).
public boolean contains(Vertex *v0*)	Returns true if the given vertex is container within the container or is along the walls of the container. Else returns false.
public void draw(java.awt.Graphics2D *g2*)	Draws the container onto the given graphics context.
public Rectangle2D.Double getContainer()	Returns a reference to the container.
public PartList getPartList()	Returns a reference to the part list.
public java.util.Vector getProfile()	Returns a reference to the container profile.
public double height()	Returns the height of the container.
public synchronized Vertex origin()	Returns the origin of the container.

Table 15. Continued

Method Summary	
public void reset()	Resets the dimensions of the container to the dimensions specified through the constructor.
public void reset(Vertex *v0*, Vertex *v1*)	Resets the dimensions of the container to the given dimensions. Vertex *v0* and *v2* define any one diagonal of the container.
public void setProfile(java.util.Vector *newPr*)	Sets the profile of the container to the new profile.
public java.lang.String toString()	Returns the string representation of the container.
public double width()	Returns the width of the container.

Table 16. Attributes and methods contained in packingDataStruct.Container.Profile

Field Summary	
private java.util.Vector *rawProfile*	Raw profile of the container. The raw profile contains the exact profile of the top edges of the packed parts.
Constructor Summary	
Profile(Rectangle2D.Double *container*)	Creates a profile for the given container and initializes it.
Method Summary	
public java.util.Vector computeProfile(Part *part*)	Computes a new raw profile after the given part has been placed in the container.
public void draw(java.awt.Graphics2D *g2*)	Draws the profile to the given graphics context.
public Vertex get(int *index*)	Returns a profile vertex by index. Returns null if the vertex does not exist.
public void reset(Rectangle2D.Double *container*)	Re-initializes the profile for the given container.
public int size()	Returns the number of vertices in the container.
public java.util.Vector smoothProfile(java.util.Vector *nPr*, double *minAng*)	Smoothes the raw profile by removing vertices that are coincident, collinear and edges that subtend a concave angle less than *minAng*. The function returns the smoothed profile.
private void monotonize(java.util.Vector *pr*)	Makes the given vertex list monotonic along the x-axis.

Table 17. Attributes and methods contained in packingDataStruct.ConvexHull

Field Summary	
private Vertex v0	Reference to the bottommost rightmost vertex in the polygon.
private HullElement [] hullArray	Defines an element on the convex hull.
Constructor Summary	
public ConvexHull(Part parts)	Computes the convex hull of the given part and updates the part attribute with the hull information.
Method Summary	
private void grahamScan(HullElement [] hullArray)	Performs the graham scan.

Table 18. Attributes and methods contained in packingDataStruct.ConvexHull.HullElement

Field Summary	
private int *index*	Index of the hull element.
private Vertex *v0*	Vertex represented by the hull element.
private boolean *isOnHull*	Set to true if *v0* is a hull vertex.
private boolean *ang*	Angle subtended at the bottom most left most vertex by *v0* and the horizontal axis.
Constructor Summary	
HullElement(Vertex *v0*, int *index*)	Creates a hull element for the give vertex with index *index.*
Method Summary	
public int compareTo(Object o)	Implements the compareTo method that is part of the *comparable* interface in Java. The method compares two hull elements from the point of view of the convex hull algorithm. Returns, 1 if vertex in object o is to the left of the line joining *v0* (bottom most right most vertex of the part) and this vertex -1 if the vertex in object 0 is to the left of the line joining *v0* to this vertex. 0 if the vertex in object o collinear with the line joining *v0* to this vertex.

Table 19. Attributes and methods contained in packingDataStruct.Heuristic

Field Summary	
private Container *container*	Reference to the container.
private boolean *stability*	Set to true if stable placements are required. Else set to false.
private int *totalIterations*	Used to compute the average complexity of the heuristic.
private int *totalVerts*	Used to compute the average complexity of the heuristic.
Constructor Summary	
public Heuristic (DrawPanel *drawPanel*, boolean *stability*)	Creates an instance of the heuristic with container input taken from the draw panel. s*tability* is set to true if stable placements are required.
Method Summary	
public double getComplexity()	Returns the average case complexity of the placement heuristic.
public void packByOrder(Chromosome *chromo*)	Packs the contents of the container in the order specified by the given chromosome.
public boolean packPart(Part *part*)	Packs the given part into the container.
protected static Vertex dropPart(java.util.Vector *profile*, Part *part*)	Drops the given part on to the give profile such that the part makes at least one point of contact with the profile. This point of contact is returned by the function. The method assumes that the part is initially positioned above the profile.
protected boolean rotateToSecondContact(Part *part*, java.util.Vector *profile*, Vertex *ct1*)	Rotates the given part that has been dropped onto the given profile such that the part makes at least two points of contact with the profile. *ct1* is the first point of contact about which the part is rotated to get a second point of contact. Returns true if the placement is admissible. Else, returns false.
private static double getSecondContact(Vertex *v0*, Vertex *v1*, Vertex *rad*, Vertex *cen*, Vertex *ct2*, int *turn*)	Returns the angle by which the line segment *v0v1* must be rotated about *cen* in order to make a point of contact *ct2* with the edge joining the vertices *rad* and *cen*. The point of contact is returned though the reference *ct2* in the parameter list. The direction of rotation is specified by *turn*. If *turn* is equal to 1, find angle for counter clockwise rotation in graphics coordinates. If *turn* equals 2, find angle for clockwise rotation.

Table 20. Attributes and methods contained in packingDataStruct.Part

Field Summary	
private double *area*	Area of the part.
private java.awt.geom.Rectangle2D.Double *bbox*	Bounding box of the part.
private int *bVert*	Index to bottommost leftmost vertex.
private Vertex *centroid*	Centroid of the part.
public boolean *isPacked*	Set to true if part is packed. Else set to false.
private int *lVert*	Index to leftmost topmost vertex.
private double *minAng*	Minimum angle subtended at a convex vertex of the part.
private int *numHullVerts*	Number of convex hull vertices in the part.
public Pose *pose*	Pose information of the part after it has been placed in the container. Null if part is not packed.
Constructor Summary	
public Part()	Creates an empty part.
public Part (Part *part*)	Duplicates a given part.
public Part(java.lang.String *part*)	Creates a part from its string representation.
public Part(java.util.Vector *vertexList*)	Creates a part from the given list of vertices.
Method Summary	
public double area()	Returns the area of the part.
public int bVert()	Returns the index of the leftmost bottommost vertex.
public Vertex centroid()	Returns the centroid of the part.
public Object clone()	Returns a shallow copy of the part.
private void computeVertexConvexity()	Tags part vertices as convex or concave.
public void draw(java.awt.Graphics2D *g2*)	Draws the part onto the given graphics context.
public Vertex get(int *index*)	Returns the vertex with the specified index. Returns null if vertex does not exist.
public Rectangle2D.Double getBbox()	Returns the bounding box of the part.
public java.util.Vector getLowerVertices()	Returns the lower vertices that lie between the leftmost and rightmost vertices inclusive.
public java.awt.Polygon getPolygon()	Returns the part as a polygon.
public java.util.Vector getVertexList()	Returns the vertex list of the part.
public int lVert()	Returns the leftmost topmost vertex of the part.

Table 20. Continued

Method Summary	
public double minAng()	Returns the minimum convex angle in the part.
public int numHullVerts()	Returns the number of convex hull vertices in the part.
public void rotate(Vertex *refVert*, double *angle*)	Rotates the part about *refVert* by the given angle.
public void rotateCCWToNextHullVertex()	Rotates the part such that the convex hull vertex that lies on the lower side and nearest to the rightmost bottommost vertex is made the rightmost bottom most vertex.
public void rotateCWToNextHullVertex()	Rotates the part such that the convex hull vertex that lies on the lower side and nearest to the leftmost topmost vertex is made the leftmost top most vertex.
public static void rotateVertexList(java.util.Vector *vertexList*,Vertex *refVert*, double *angle*)	Rotates the given vertex list about *refVert* by the specified angle.
public int rVert()	Returns the index of the rightmost bottommost vertex.
public boolean selfIntersects()	Returns true if the part geometry is found to be self-intersecting.
public void set(Part *pp*)	Sets the part attributes to the given part.
public int size()	Returns the number of vertices in the part.
private void switchOrientation()	Makes a clockwise oriented part counterclockwise and wise versa.
public java.lang.String toString()	Returns the string representation of the part.
public void translate(Vertex *fromVert*, Vertex *toVert*)	Translates the part relative to *fromVert* and *toVert*.
public static void translateVertexList(java.util.Vector vList,Vertex *fromVert*, Vertex *toVert*)	Translates the given vertex list relative to *fromVert* and *toVert*.
public int tVert()	Returns the index of the topmost right most vertex.
public void unpack()	Unpacks the part if it is packed.
protected void updateBbox()	Updates the bounding box of the part.
public java.util.Vector vertexList()	Returns the vertex list of the part.

Table 21. Attributes and methods contained in packingDataStruct.PartList

Field Summary	
private java.util.Vector *masterList*	Original part list.
private java.util.Vector *pList*	Part list on which the algorithm is executed.
Constructor Summary	
public PartList()	Creates an empty part list.
public PartList(PartList *partList*)	Creates a part list from the list of parts given.
public PartList(java.lang.String *partList*)	Creates a part list from a string representation of that part list.
Method Summary	
public boolean add(Part *part*)	Adds a part to the list.
public synchronized void draw(java.awt.Graphics2D *g2*)	Draws all the parts in the list to the given graphics context.
public Part get(int *index*)	Gets a part by index. Returns null if requested part is not found.
public Part remove(int *index*)	Removes a part with the given index from the part list.
public void reset()	Copies the contents of *masterList* into *pList.*
public int size()	Returns the number of parts in the list.
public java.lang.String toString()	Returns the string representation of the part list.

Table 22. Attributes and methods contained in packingDataStruct.Pose

Field Summary	
private double *x*	x-coordinate of the pose.
private double *y*	y-coordinate of the pose.
private double *ang*	Defines the orientation of the pose.
Constructor Summary	
public Pose(double *x*, double *y*, double *ang*)	Creates a pose with position *x,y* and orientation *ang*.
public Pose(Pose *pose*)	Create a pose from another pose.
public Pose(Vertex *v0*, double *ang*)	Creates a pose with position *v0* and orientation *ang*.
Method Summary	
public Object clone()	Returns a shallow copy of the pose.
public double getOrientation()	Returns the orientation of the pose.
public double getX()	Returns the x-coordinate of the pose.
public double getY()	Returns the y-coordinate of the pose.
public Vertex position()	Returns the position of the pose.

Table 22. Continued

Method Summary	
public void set(Pose *pose*)	Sets the pose to a new pose.
public java.lang.String toString()	Returns the string representation of the pose.

Table 23. Attributes and methods contained in packingDataStruct.Vertex

Field Summary	
public double *angle*	Angle made by the bisector of the two edges connected to this vertex and the positive x-axis.
public boolean *isConvex*	Set to true if the vertex is a convex vertex in a list of vertices. Else set to false.
protected boolean *isOnHull*	Set to true if the vertex belongs to a polygon and is a convex hull vertex. Else set to false.
Constructor Summary	
public Vertex()	Creates a vertex with default coordinates (0,0).
public Vertex(double *x*, double *y*)	Creates a vertex with coordinates (*x*,*y*).
public Vertex(java.lang.String *vertex*)	Creates a vertex from the given string representation of the vertex.
public Vertex (Vertex *vertex*)	Creates a vertex from another vertex.
Method Summary	
public static Vertex add (Vertex *v0*, Vertex *v1*)	Returns the sum of two vertices, i.e. the sum of the two vectors geometrically equivalent to that vertex.
public Object clone()	Returns a shallow copy of a vertex.
public boolean coincident(Vertex *vert*)	Returns true if the given vertex is coincident to this vertex. Else returns false.
public synchronized void draw(Graphics *g2*)	Draws this vertex on to the given graphics context.
public void rotate(Vertex *refVert*, double *angle*)	Rotates this vertex about *refVert* by the specified angle.
public void set(Vertex *vert*)	Sets this vertex data to the data of the given vertex.
public static Vertex sub (Vertex *v0*, Vertex *v1*)	Returns the difference between two vertices, i.e. the difference of the two vectors geometrically equivalent to these vertices.
public java.lang.String toString()	Returns the string representation of this vertex.
public synchronized void translate(Vertex *fromVert*, Vertex *toVert*)	Translates this vertex relative to *fromVert* and *toVert*.

APPENDIX B
USER INTERFACE

Figure B.1 Shows the graphical user interface that was used to input data and for the visualization of the output. The Swing classes in Java1.3 were used to build the interface. The user interface was built in a way that allowed the program to be run as an applet or a stand-alone program.

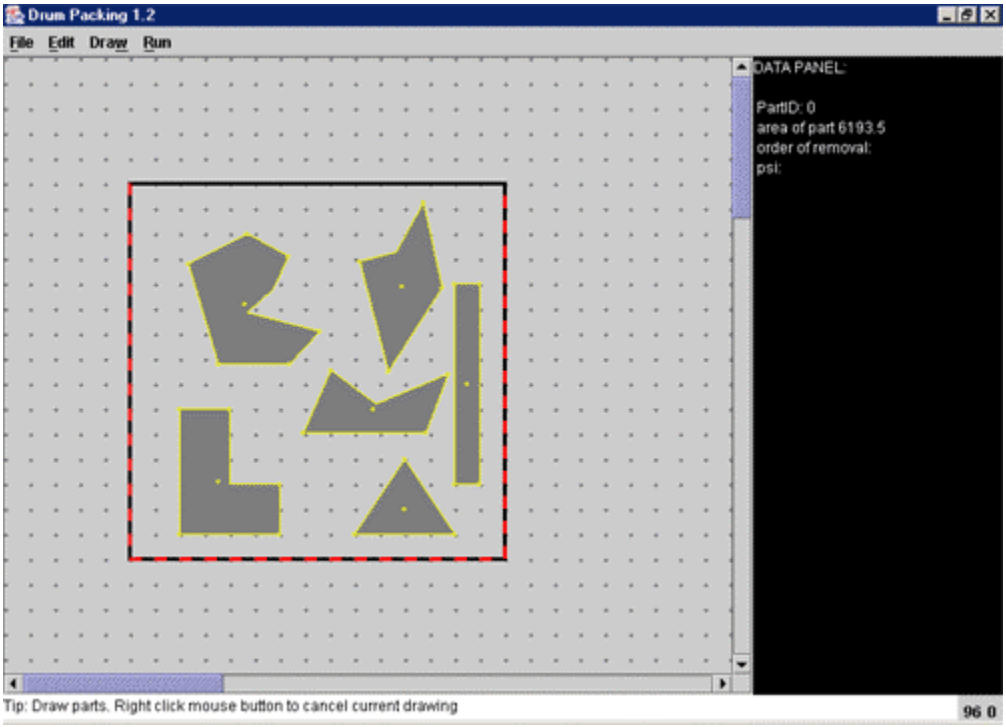


Figure B-1 Arbitrary and geometric shaped patterns drawn on the user interface with the grid switched on.

The draw panel allowed the user to draw arbitrary shapes as well as geometric shapes with the help of the mouse. Geometric shapes could be drawn by switching on the snap mode from the “edit” menu. When in snap mode, the location of a mouse click was

set to the grid point that was nearest to the actual mouse click. Grid points had a resolution of 20x20 pixels. The edit menu also provided features that could be used to move, delete or make copies of patterns.

A data panel was used to output global data such as the fitness of the best chromosome, the number of generations that the program has run, average time complexity of the heuristic etc. When the mouse was moved over a pattern, the data panel displayed data associated with the pattern. The data panel was updated in real time by using a separate thread for the execution of the genetic algorithm. The draw panel was updated each time a better solution was found.

The pattern and container data on the draw panel could be read into and out of persistent memory in the form of a formatted ASCII file. The “run” menu gave the user the option to run either the genetic algorithm or just the online placement heuristic on the input in the draw panel.

A status panel was used to display error messages and tool tips. It also showed the coordinates of the mouse pointer in the graphic coordinate system when the mouse pointer was on the draw panel.

LIST OF REFERENCES

Alb80 Albano A, Sapuppo G. Optimal Allocation of Two Dimensional Irregular Shapes Using Heuristic Search Methods. IEEE Transactions on systems, Man, and Cybernetics May 1980; SMC 10(5).

Cag96 Cagan J, Kolli A, Rutenbar R. Packing of Generic Three Dimensional Components Based On Multi-Resolution Modeling. Proceedings of The 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference; 1996 August 18-22; Irvine, CA.

Cor01 Corman HC, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*, Cambridge, Massachusetts, MIT Press, 2001.

Das97 Dasgupta D, Michalewicz Z. *Evolutionary Algorithms in Engineering Applications*, Berlin, Springer, 1997, pp. 515-530.

deB00 de Berg M, van Krevald M, Overmars M, Schwarzkopf O. *Computational Geometry Algorithms and Applications*, Berlin, Springer, 2000.

Fal96 Falkenauer E, A Hybrid Grouping Genetic Algorithm for Bin Packing. In Journal of Heuristics, 2(1), Kluwer Academic Publishers 1996; pp. 5-30.

Gol89 Goldberg DE. *Genetic Algorithms in Search, Optimization and Machine Learning*, New York, Adison Wesley, 1996.

Hau98 Haupt RL, Haupt SE. *Practical Genetic Algorithms*, New York, John Wiley, 1998.

Hoc95 Hochbaum DS. *Approximation Algorithms for NP-Hard Problems*, Boston, Massachusetts, PWS Publishing Company, 1995.

Hol75 Holland JH. *Adaptation in Natural and Artificial Systems*, Ann Arbor, The University of Michigan Press, 1975.

Kir83 Kirkpatrick S, Gelatt CD, Vecchi MP. Optimization by Simulated Annealing. Science, 1983; May 13; 220(4598).

Lod99 Lodi A, Martello M, Vigo D. Approximation Algorithms for the Oriented Two-Dimensional Bin Packing Problem. European Journal of Operations Research 1999; 112: 158-166.

Mar90 Martello S, Toth P. Lower Bounds and Reduction Procedures for the Bin Packing Problem. Discrete Applied Mathematics 1990; 22: 59-70.

Mar98 Martello S, Vigo D. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. Management Science 1998; March; 44(3).

Mar00 Martello S, Pisinger D, Vigo D. The Three Dimensional Bin Packing Problem. Operations Research 2000; March-April; 48(2): 256-267.

McG97 McGee RJ. Three-Dimensional Packing Algorithm Using Voxel Modeling [thesis]. Gainesville (FL): University of Florida; 1997.

Mic92 Michalewicz Z. *Genetic Algorithms + Data Structures = Evolution Programs*, Berlin, Springer-Verlag, 1992.

Nil71 Nilsson NJ. *Problem-Solving Methods in Artificial Intelligence*, New York, McGraw Hill, 1971.

Pet98 Petridis V, Kazarlis S, Bakirtzis A. Varying Fitness Functions in Genetic Algorithm Constrained Optimization: The Cutting Stock and Unit Commitment Problems. IEEE Transactions on Systems, Man, and Cybernetics part b: Cybernetics 1998; October; 28(5).

Rou98 O'Rourke J. *Computational Geometry in C*, Second Edition, Cambridge, Massachusetts, University Press, 1998.

Sit02 Sitharam M, Wu X. Optimal Placement for 2D Non-Oriented Geometric Bin-Packing [Manuscript]. Department of Computer and Information Science and Engineering, University of Florida, Gainesville (FL), 2002.

Szy95 Szykman S, Cagan J. A Simulated Annealing Approach to Three-Dimensional Component Packing. ASME Journal of Mechanical Design 1995; 117(2(A)): 308-314.

BIOGRAPHICAL SKETCH

Arfath Pasha was born in Bangalore, India, on October 28, 1973. Shortly after completing his bachelor's degree at the University of Mysore, India, in mechanical engineering he attended the University of Florida, Gainesville. Becoming interested in robotics, he pursued a concurrent master's degree in mechanical engineering and computer information science and engineering under the guidance of Dr. Carl D. Crane III and Dr. Meera Seetharam. During this time he also worked as a graduate research assistant at the Center for Intelligent Machines and Robotics.