Implementation and Performance Improvement of the Evaluation of a Two Dimensional Bin Packing Problem using the No Fit Polygon

Edmund Burke
University of Nottingham,
University Park
NG7 2RD, UK
ekb@cs.nott.ac.uk

Graham Kendall
University of Nottingham,
University Park
NG7 2RD, UK
gxk@cs.nott.ac.uk

Abstract : When using a meta-heuristic algorithm (e.g. simulated annealing) it is often the case that the evaluation function is the most computationally expensive phase of the algorithm. This is the case in the research in which we are currently conducting.
For the 2D stock cutting problem we calculate the NFP (No Fit Polygon) for two polygons. Next we calculate the convex hull, as many times as there are vertices on the NFP, for these two polygons. This process is repeated whilst we consider the set of polygons that need to be packed.
Due to the nature of meta-heuristic algorithms this procedure is repeated many times. As the manipulation of polygons is computational expensive, the algorithm shows a bottleneck at this evaluation stage.
In fact, many of the evaluations are simply re-evaluating (complete or partial) solutions that have been seen earlier in the algorithm. In order to use these previous evaluation we introduce a cache which holds previous (complete and partial) solutions. We demonstrate that by increasing the cache size we can significantly increase the speed of the algorithm. We also introduce the concept of polygon types which allow us to make much better use of the cache.
However, in some circumstances, it may not be beneficial to use the cached evaluation as this would simply keep returning the same evaluation value. There may be better solutions available if we re-evaluate earlier polygons in the set. Therefore, we show that by introducing a re-evaluation parameter that, we can stop the algorithm always using a cached solution and potentially find a better solution by re-evaluating the set of polygons.
This paper also describes the data structures that we use to support the operations described above, as well as general computational geometry algorithms we use. We also present our computational results.
Please note that this paper only presents results from cutting problems in order to show the effectiveness of our optimization algorithms.

Keywords : stock cutting, simulated annealing, packing, computational geometry

1 Introduction

Our research is concerned with the 2D Stock Cutting problem. This involves placing a series of shapes onto a stock sheet(s) such that the shapes do not overlap, they lie within the confines of the stock sheet and the waste is minimised. Specifically, our research is looking at producing good quality solutions to these problems using meta-heuristic algorithms (such as simulated annealing). In addition we want to be able to place any arbitrary shapes onto stock sheets not, for example, just rectangles.

Subsequent papers to this will report on the findings of our research with regards to 2D stock cutting problem. The purpose of this paper is to outline some of the fundamental data structures and design decisions we have made in implementing our algorithms. The principles presented in this report will form the basis for our future research.

1.1 Problem Complexity

To give some idea of the complexity involved in dealing with certain types of shapes we present here a brief analysis of the computational complexity of three shape categories with respect to calculating their intersection.

The 2D stock cutting problem can be viewed as a problem that places three types of shapes onto a stock sheet. In increasing order of complexity these shapes are.

- **Rectangular shapes**. A rectangle can be represented simply by defining two coordinates. Operations such as detecting if two rectangles intersect is easy and computationally inexpensive.
- **Convex Polygons**. These data structures can be represented as a number of vertices. When joined, the vertices create a shape such that no internal angle is greater than 180° (later in the paper we discuss in more detail how a polygon can be represented). The complexity of an operation such as calculating the intersection of two convex polygons can be done in linear time, O(n + m) (where n and m are the number of vertices of the polygons) (O'Rourke, 1994). It can also be shown that the intersection of two convex polygons will only ever return another (single) convex polygon with the number of vertices being n + m.
- **Non-Convex Polygons**. Non-Convex polygons allow internal angles that are greater than 180°. This increases the complexity of operations such as computing overlap as you can now potentially have intersections which return more than one polygon. Due to this, the operation is of quadratic complexity (O(nm)) (O'Rourke, 1994).

It is also possible to classify polygons as *simple* and *non-simple*. Simple polygons have no edges crossing, whilst non-simple polygons have one or more edges crossing each other. For the purposes of this paper, and our future research, we only deal with simple polygons.

As our research is concerned with packing arbitrary shapes, we work with polygons (as opposed to rectangles). For the reasons outlined above, this adds computational complexity to this problem and it is the aim of this paper to outline how we have reduced the running times of the algorithms, whilst maintaining good quality solutions.

2 Computational Geometry

In order to discuss the packing of polygons it is necessary to consider some aspects of computational geometry.

Firstly, we describe the abstract data types (ADT's) that we have implemented that allow us to manipulate polygons. Next we consider the No Fit Polygon (NFP) which is fundamental to our research.

2.1 Abstract Data Type - Point

Much of our research relies on the fact that polygon data structures can be manipulated.
The literature (Laszlo, 1996, O'Rourke, 1994, Sedgewick, 1992) is in agreement that polygons should be represented by a set of points, with each point representing a vertex of the polygon.
Following the convention of representing a point using cartesian co-ordinates it would be beneficial if integers could be used to represent the x and y co-ordinates. This would guarantee numerical accuracy when performing operations on a point. At least, accuracy within the limits of the integer representation of the computer on which the point has been implemented. If we stored integers in 32 bits, unsigned, this would allow a range for the x and y co-ordinates of 0 to 4,294,967,295.
However, using integers does present problems. When carrying out some operations the result will be a floating point number. For example, when detecting the intersection of two lines (with each line being represented by two points) the result is unlikely to be a pair of integer numbers, which represent the exact point of intersection.
Similarly if a point is rotated, the new position for the point is not guaranteed to be x and y co-ordinates which are integers.
It was for these reasons that our point data type was implemented using a floating point representation. The potential danger of using this representation is possible loss of accuracy when performing calculations. In practise this problem has never manifested itself; although it is still recognised as a potential problem area.

The main operation we needed to be able to perform on a point data type is to rotate it. The method to rotate a point, given just its x and y co-ordinates is well known and is reported in many texts, including (Hearn, 1994) and (Adamowicz, 1976). The following equation shows the method.

$$x' = x \cos ? - y \sin ?$$
$$y' = x \sin ? + y \cos ?$$

This rotates a point at position (x,y) through an angle ?, about the origin. If it is necessary to rotate a point around a pivot point other than the origin then the following formula can be used.

$$x' = x_r + (x + x_r) \cos ? - (y - y_r) \sin ?$$
$$y' = y_r + (x - x_r) \sin ? + (y - y_r) \cos ?$$

where x_r and y_r represent the pivot point.

2.2 Abstract Data Type - Polygon

Polygons can be represented by a list of points, with each point representing a vertex of the polygon. The last point in the list is considered to be connected to the first point. It is usual to order the points in a counter clockwise order. The operations we have implemented for polygons are described below.

2.2.1 Primitive Operations

One important primitive is to calculate the area of a triangle given its Cartesian co-ordinates. (O'Rourke, 1994) describes one such algorithm and describes how this can be used as a basis to calculate the area of a polygon.
Another useful primitive is to decide the relationship between two points (p_1, p_2) and another point (p_3). It is useful, for example, to decide if p_3 is co-linear with p1 and p_2 or if p_3 is to the left (or right) of p_1, p_2.
In (O'Rourke, 1994) this property is returned via a *left* predicate (i.e. is p_3 on the left of the line represented by p_1, p_2?). The left predicate uses the triangle area function mentioned above and returns if the area is positive (indicating that p_3 is to the left of p_1 and p_2), zero (indicating the points are collinear) or negative (indicating that p_3 is to the right of p_1 and p_2).
(Sedgewick, 1992) uses the same basic principle but instead of a left predicate, the operation is implemented as a *CCW* (Counter Clockwise) function. This returns an integer depending on whether the points travel clockwise or counter clockwise. It is Sedgewick's algorithm that is most often quoted in the literature on cutting and packing. This is due to the fact that many of the algorithms only need to know how points are oriented with regards to one another and a triangle area calculation is not required, which is the basis of O'Rourke's algorithm.
A fundamental property in computational geometry is to decide if two lines (where a line is represented by two vertices) intersect. (Sedgewick, 1992) uses the CCW primitive to implement this algorithm. (O'Rourke, 1994) uses the left predicate as the basis for his intersection algorithm.
Both of these algorithms simply return a boolean stating if the lines intersect. Sometimes it is necessary to know the point of intersection. (O'Rourke, 1994) describes an algorithm to do this and states that in this situation it is necessary to leave the *comfortable* world of integer co-ordinates and resort to floating point values that represent the x and y co-ordinates of the point of intersection. Line intersection is obviously fundamental to many other algorithms (such as deciding if two polygons overlap).

A primitive to calculate the angle a given line (represented by two points) makes with the horizontal is, at times, useful. This primitive is useful in many situations. For example, you can take an arbitrary number of points, sort them according to the angle they form with the horizontal and then create a polygon by assembling the points into a polygon data structure using a counter clockwise ordering based on their angle.
(Sedgewick, 1992) uses a theta function to carry out this operation which, when given two points, returns a real number that is *not* the angle formed with the horizontal but it has the same properties. The reason that the

actual angle is not calculated is because this would require a call to a tangent function, which is computationally expensive.

2.2.3 Single Polygon Operations

To implement a rotate operation for a polygon is easy as we already have the ability to rotate a point. The most common requirement is to rotate a polygon around one of its vertices, *pv*. Each point, with the exception of *pv*, has to be rotated around *pv*.
Implementing an operation to return the area of a polygon is also easy to implement as we can use the calculation of the area of a triangle as the basis for this algorithm ***(ref needee O'Rouke – I think).***
Finding the convex hull of a polygon is another useful operation. This can be visualised as stretching an elastic band around a shape. The contour made by the band is the convex hull of the polygon. There are several algorithms that can be used to calculate the convex hull of an arbitrary polygon.
Package (or Gift) Wrapping is described in both (O'Rourke, 1994), (Sedgewick, 1992) and (Preparata, 1985). The original idea for this method was proposed by (Chand, 1970). The algorithm works by choosing a point that is obviously on the convex hull (for example, the vertex with the smallest y co-ordinate) and then sweeping a horizontal ray in the positive direction and moving it round until another point is hit. This point is guaranteed to be on the hull. Of course, in practise, the algorithm needs to consider each point to see which one will be hit next by the sweep line. In (Sedgewick, 1992), the theta function is used to determine which point should be next on the hull. The main disadvantage with this algorithm is that it runs in $O(n^2)$ in the worst case, that is when every point is on the hull.
Other convex hull algorithms are also available. For example, Quick Hull (Preparata, 1985) is an alternative to Package Wrapping but it is the Graham Scan (Graham, 1972) that is most often used with regards to the cutting and packing group of problems. The main advantage of this algorithm is that it runs in O(n log n) in the worst case. Most of the computational geometry textbooks show an implementation of the Graham Scan.

Another basic operation we need for polygons is to decide when one polygon intersects another. There are two ways in which we can represent the intersection. The first is just to return a boolean value indicating if the polygons intersect. A more complex, but potentially more useful, operation is to return a polygon that represents the intersecting area. In addition we need to consider both convex and non-convex polygons. The operation is a more complex for non-convex polygon (see section 1.1).
In this paper we will only consider convex polygons. Only if our current line of research leads to good quality solutions for convex polygons will we tackle the stock cutting problem for non-convex polygons.

The obvious method to decide if two polygons intersect is to check each line of one polygon against every line of the other polygon. If any lines intersect then the polygons intersect. One degenerate case has to be catered for. That is, if one polygon totally includes another. Assuming no lines intersect inclusion can be checked for by taking every point on one polygon and checking to see if it lies to the left (using the left predicate or the CCW primitive) of every line of the other polygon. If this is the case the polygon lies totally within the other. Of course we only need to check if the polygon with the smallest area is included within the larger polygon. The reverse case, of the larger polygon fitting inside the smaller polygon obviously cannot occur.
(Sedgewick, 1992), although not presenting an algorithm to detect if two polygons intersect, does provide all the necessary algorithms to implement the method described above. It also states that such a naive approach, to check if any lines intersect, runs in time proportional to n^2, where n is the number of edges.
Therefore, an algorithm is presented that can detect if any two lines intersect that runs in time proportional to n log n (but still n^2 in the worst case). This method, based on maintaining a scan line as it passes across a plane containing the points, was originally proposed by (Shamos, 1976).
In 1978 (Shamos, 1978) developed the first polygon intersection algorithm that ran in linear time; O(n + m), where n and m are the number of edges.
In (O'Rourke, 1994) an algorithm is presented that also achieves O(n + m) time. This is based on an idea developed by O'Rourke and his students (O'Rourke, 1982). The algorithm involves two lines "chasing" one another around the edges of the polygons and plotting points as the head of the lines are advanced. At

termination the algorithm returns a polygon representing the overlap area. This method is also described in (Preparata, 1985).

2.2.4 No Fit Polygon (NFP)

The No Fit Polygon (NFP) determines all arrangements that two arbitrary polygons may assume such that the shapes do not overlap but they cannot be moved closer together without intersecting. The concept of the NFP was originally proposed by (Art, 1966) and was used by (Adamowicz, 1972 and 1976). It is the Adamowicz papers that are most often cited with regards to stock cutting.
To show how the NFP is calculated we give an example. Consider the following two polygons; P_1 and P_2.

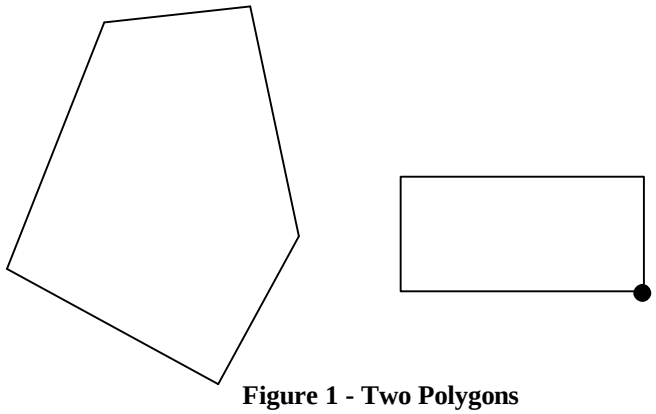

Figure 1 - Two Polygons

The aim is to find an arrangement such that the two polygons touch but do not overlap. If we can achieve this then we know that we cannot move the polygons closer together in order to obtain a tighter packing. In order to find the various placements we proceed as follows

- One of the polygons (P_1) remains stationery.
- The other polygon, P_2, moves around P_1 and stays in contact with it but never intersects it.
- Both polygons retain their original orientation. That is, they never rotate.
- As P_2 moves around P_1 one of its vertices (the reference point – shown as a filled circle) traces a line (this becomes the NFP).

The following diagram shows the starting (and finishing) positions of P_1 and P_2. The NFP is shown as a dashed line. It is slightly enlarged so that it is visible. In fact, some of the edges would be identical to P_1 and P_2.

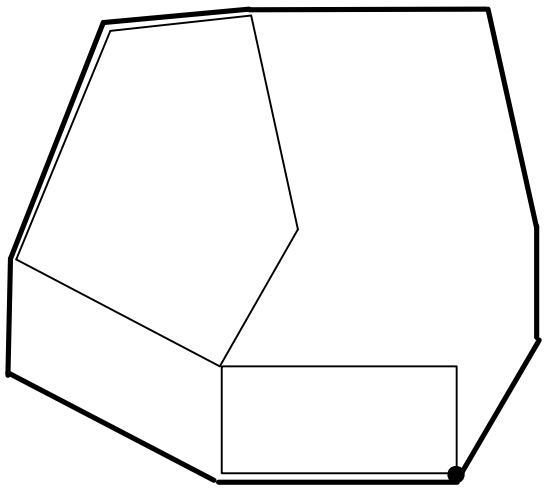

Figure 2 - No Fit Polygon

Once we have calculated the NFP for a given pair of polygons we can place the reference point of P_2 anywhere on an edge of the NFP in the knowledge that it will touch, but not intersect, P_1.

In order to implement a NFP algorithm it is not necessary to emulate one polygon orbiting another (although we initially implemented an algorithm that did this). (Cunninghame-Green, 1992) presented an algorithm that we now use. It works on the assumption that (for convex polygons only) the NFP has its number of edges equal to the number of edges of P_1 plus the number of edges of P_2. In addition, the edges of the NFP are copies of the edges of P_1 and P_2, suitably ordered. To build the NFP it is a matter of taking the edges of P_1 and P_2, sorting them and building the NFP using the ordered edges. This algorithm is also presented in (Cunninghame-Green, 1989), although here it is presented as a Configuration Space Obstacle (CSO).

3 Evaluation

By utilising the NFP, we have developed an evaluation strategy that we use in our research. We will describe the evaluation using convex polygons but we hope to use the same methods for non-convex polygons once we have shown the effectiveness of the method.

Although we are experimenting with various evaluation functions they all use the same basic method. Given a set of polygons, $P_1..P_n$, we initially find the NFP for P_i (i = 1) and $P_{i+1.}$ Next we find the optimal placement of these two polygons. This is done by placing P_{i+1}'s reference point on the NFP and calculating the convex hull of P_i and P_{i+1}. After calculating convex hulls for a number of placements, the convex hull with the smallest area is returned. This represents the best placement of P_{i+1} in relation to P_i. The polygon returned from the convex hull operation is then paired with the next polygon in the set (P_{i+2}).
This process continues until P_n has been placed.
As stated above, this is the basic method but we are experimenting with variations on this theme. For example, placing the polygons within the confines of a bin so that we can evaluate two-dimensional bin packing.

However, there are two problems that we need to address in order to make the evaluation strategy feasible.
The number of placements of the reference point on the NFP is infinite as the reference point can be placed anywhere on the edge of the NFP. In order to reduce the problem to manageable proportions, we only place the reference point on the vertices of the NFP when looking for the optimal placement. Thus we calculate the convex hull as many times as there are vertices on the NFP.

The second problem is that there could be more than one optimal placement for two given polygons.
Consider two rectangles of the same dimensions. There are four optimal placements, as shown below

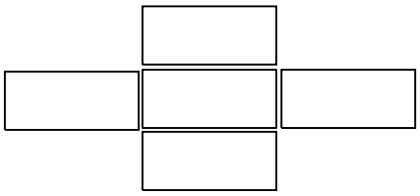

Figure 3 - Four optimal placements of two polygons

No matter which placement we choose the four evaluations return the same value as the convex hulls all have the same area. Therefore, it is immaterial which one we choose. However, it may make a difference when later polygons are added. Consider for example if we select the P_{23} placement in relation to P_1. Depending on the characteristics of the next polygon, P_3, it could effect the quality of the solution that we are building.
The following figure, 4, illustrates this. They show that by choosing P_{23} initially we get a better solution than if we had chosen P_{22}.

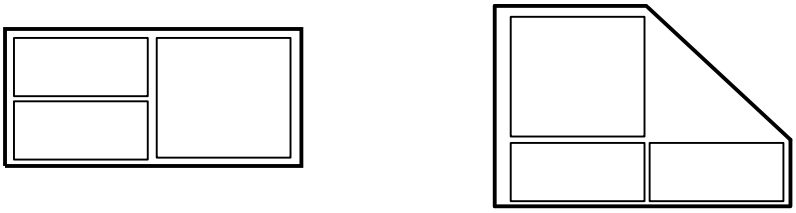

Figure 4 - Later evaluations can effect solution quality

One option we could incorporate is to implement some type of "look-ahead" so that we can select better placements earlier in the solution. However, we have not implemented this as we consider it would be too computationally expensive, although it could be a future line of research. Instead, when faced with two or more equal evaluations we choose one at random. However, we do further address this issue in section 6.

4 Caching of Evaluations

The manipulation of polygons is computationally expensive. In addition, due to the nature of meta-heuristic algorithms the evaluation function has to be called many times. The following illustrates this.

1. Given polygons, $P_1..P_n$, we initially pair P_1 and P_{1+1}. The eventual output of this pairing is the convex hull of the two polygons optimally placed using the NFP. This polygon is paired with P_3 and the process repeated until P_n has been placed.
2. Therefore, for polygons $P_1..P_n$ there are n-1 NFP calculations required.
3. In addition, we need to calculate as many convex hulls as there are vertices on each NFP in order to find the optimal placements.
4. Steps 2 and 3 are repeated for as many iterations as the algorithm requires. For example, using simulated annealing with a starting temperature of 30, a decrement of 0.5 and 20 iterations at each temperature would require steps 2 and 3 to be carried out 1200 times.
5. Using a population based meta-heuristic (such as a genetic algorithm) the problem is potentially larger as the entire population has to be evaluated on each iteration.

It can be appreciated that the evaluation function is a potential bottleneck within the system.
However, it is also the case that many of the evaluations could be repeated many times. Therefore, it would appear to be a good idea to store the results of previous evaluations so that we do not need to evaluate the current solution if it has been evaluated before. In practise this means implementing a cache.
In order to implement the cache we label each polygon with a unique identifier. In this way we can recognise a solution by considering a concatenation of the identifiers.

For example, given polygons, $P_1..P_6$, we could label these as A, B, C, D, E and F. One potential solution is ABCDEF, another is FEDCBA and yet another is DEABCF.

We use this concatenation of identifiers as a key to a hashing function which accesses the cache. When we access the cache we are either returned a null value, meaning that the solution is not in the cache, or we are returned the previous evaluation which means we do not have to call the evaluation function.
In addition we are also returned a polygon data structure so that we can pair this with the next polygon. In fact, the cache can hold more than one entry for each key. If you look at figure 3 you can see that this has four optimal placements. These are all held in the cache but only one is returned, this being randomly selected.
In addition to storing complete solutions in the cache, we can also hold partial solutions.
Assume we have $P_1..P_6$, with identifiers ABCDEF and these polygons are presented for evaluation.
As we evaluate each polygon we store the result in the cache. Therefore, after evaluating this permutation the cache will hold

AB
ABC
ABCD
ABCDE
ABCDEF

and with each key will be stored the result from the evaluation together with a polygon data structure.
The next permutation we evaluate might be ABCDFE.
When we evaluate AB we find it is already in the cache so do not need to call the evaluation function. The same will be true for ABC and ABCD. It is only when we evaluate ADCDF (and ABCDFE) that we need to call the evaluation function.

We have one last decision to make. How big should the cache be? Obviously, the larger the better but we cannot let it grow unbounded due to memory limitations.
The size of the cache will actually depend on the memory you have available and the data structure being held in the cache (in this case the number of vertices). At present it is of more interest to ensure that the cache does actually speed up the processing time. Intuition says it should and a simple experiment should be able to prove this.

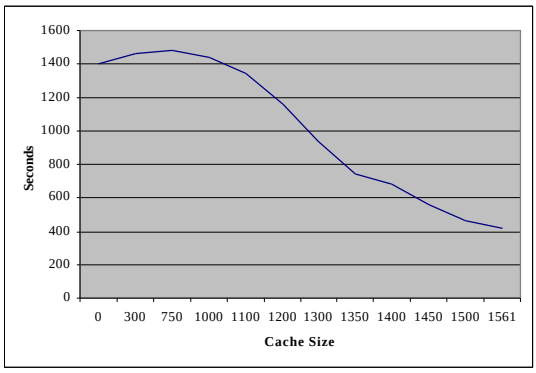

Figure 5 - Cache Size vs Running Time (problem one)

Our first experiment consists of 14 polygons. We used simulated annealing to try and pack the polygons into a small an area as possible using the evaluation method we have described above.
In this experiment we are aiming to show that increasing the cache size does lead to a decrease in the running time.
The following graph shows that this is the case.

The figures are averaged over three runs and were carried out on a Pentium90 processor using 16Mb of memory.

It can be seen that the size of the cache has a significant effect on the running time of the algorithm.
The largest number of elements in the cache (1561) was obtained by running the algorithms with a cache size of 20,000. At the end of the run the cache held 1561 entries. This provided us with a baseline figure and we carried out the remaining runs by altering the cache size between zero and 1500.

As a further test we tested the effect of the cache size on a problem taken from the real world (see figure 6)
The objective is to cut polycarbonate shapes (used in conservatories) from larger stock sheets.
In reality, the company receiving the orders has to cut many shapes from many stock sheets whilst minimising the waste. This figure is a solution for one stock sheet on a given day. However, it is a worthwhile problem to see the effect of the cache size.

This test data is different from the first set of data in that every polygon is different (some may look the same but they all have different dimensions). This means we cannot take advantage of *polygon types* (described below). This results in less cache usage than is possible when polygons of the same type are presented.
In addition the evaluation is slightly different. The same basic method is used but is now amended to accommodate the bin packing restriction.

This graph (figure 7) shows the results from this run

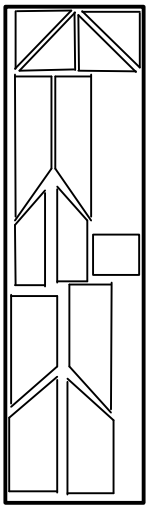

Figure 6 - Sample Layout

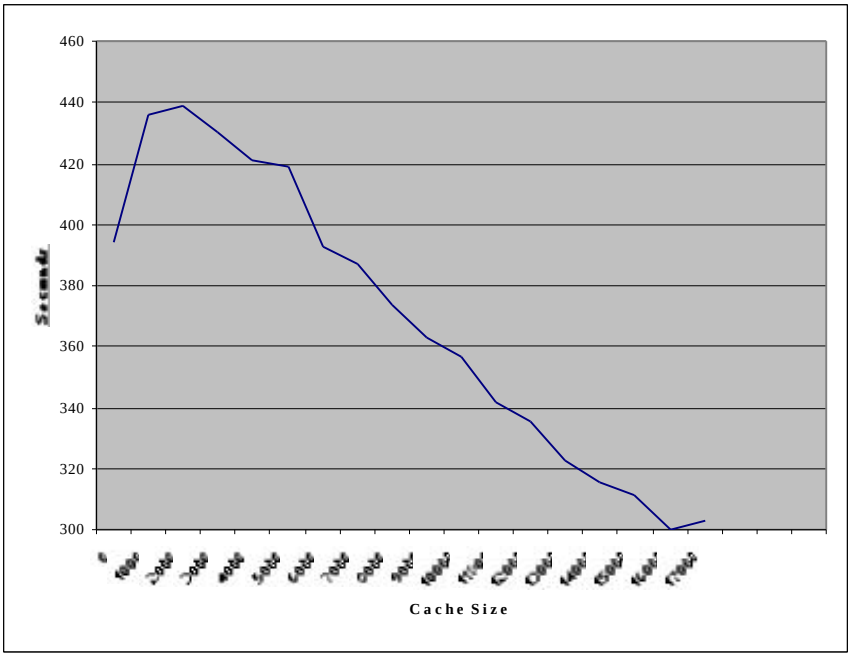

Figure 7 - Cache Size vs running time (problem two)

Again, the results are averaged over three runs. This time, the algorithm was run on a Cyrix 166 processor, with 64 MB of memory.
The upper bound (17000) was found in a similar way to the first test.
It is interesting to note that with a cache size of zero, the algorithm actually performs faster than with small cache size. This is also the case with the earlier test. We attribute this to the fact that when the cache is small the overheads in maintaining the cache outweighs the advantages in using the data stored in the cache.

5 Types

In addition to the cache we realised that another potential bottleneck in the algorithm was evaluating similar polygon permutations that had already been evaluated before.
Consider polygons with identifiers ABCDE and assume the polygons DE are identical (for example rectangles with the same dimensions).

In evaluating ABCDE we will eventually have AB, ABC, ABCD and ABCDE stored in the cache. Later in the algorithm we might need to evaluate ABCED. When we evaluate ABCE we find that this evaluation is not in the cache and we go through the evaluation function. In fact, there is no need to do this as polygons D and E have the same dimensions and evaluating ABCD will yield the same result as ABCE. Of course, we do not recognise this as the keys to the cache (the concatenation of the identifiers) are different.

In order to cater for this we introduced the notion of a polygon *type*. This gives each polygon a type identifier which acts as a pointer to the polygon description.
When we tested the cache size using our first test we could have described the polygons using fourteen different identifiers (that is, ABCDEFGHIJKLMN). In fact, there were only three different types of polygon so the identifiers became AAAABBBBBCCCCCC.
It can be appreciated that this significantly reduces the size of the search space for the problem. In addition, it allows us to make much more effective use of the cache. To show this we did two tests, one where the polygons were described as different types and another test where the polygons were described using just three

different types. Again, these tests were averaged over three runs and we set the cache size to 2000. We also set the re-evaluation probability (see below) to zero.

Using shapes of different types the run time was 267 seconds. Using shapes of the same type the run time reduced to 173 seconds.
In order to confirm that the cache and types were working in conjunction with one another, we ran the same tests but with the cache size set to zero. As you would expect both tests, under these conditions, took about the same time as there is no information in the cache so the polygon type is irrelevant as it needs to carry out an evaluation every time.

6 Forcing Re-Evaluations

One of the problems with holding information in the cache is that it might be holding an evaluation value which is not the best it could find for a particular permutation of polygons. An example will show why this can be the case.

Assume three polygons

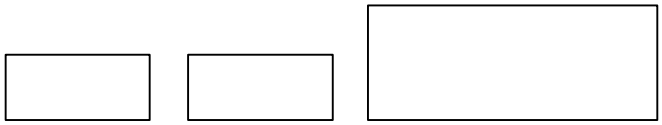

Figure 8 - Three Polygons

If we evaluate P_1 and P_2, we will find that the NFP and convex hull calculation allow us to position P_2 in relation to P_1 in four places (either on the top of P_1, on the bottom of P_1, to the left of P_1 or to the right of P_1; (see figure 3). The one we choose is random, so assume we choose P_2 to go on top of P_1.

When we evaluate P_3 with regards to P_1 and P_2 the best configuration we can find is shown here

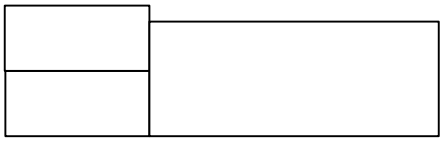

Figure 9 - Three polygons sub-optimally places if P_1 and P_2 are initially placed on top of one another

It is the result from this evaluation that will be stored in the cache and whenever P_1, P_2 and P_3 are evaluated then this value will be returned from the cache. We will never have access to the better solution where P_1 and P_2 were initially placed alongside each other (see below).

If we had initially chosen P_2 to go to the right of P_1 then we would have this configuration.

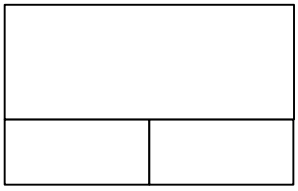

Figure 10 - Three polygons optimally placed if P_1 and P_2 are initially placed next to each other

The convex hull is now equal to the sum of the area of the three polygons, which is a better configuration than figure 9.
The only hope we have is that the cache exceeds its limit and that the inferior solution is discarded so that it has to be re-evaluated. However there is no guarantee that this will happen. In fact, the larger the cache size (in the hope of improved performance) the less chance there is of items being discarded from the cache.
In an attempt to alleviate this problem we introduced a *re-evaluation* parameter. This is set to a value (between 0 and 1) and it determines if the solution should be re-evaluated, regardless of whether or not it is in the cache. A value of zero means that the value in the cache is always used, if it exists. A value of one means that the solution is always re-evaluated, effectively ignoring the cache.

Higher values of the re-evaluation parameter will slow the algorithm down as it is not making as much use of the cache as it could do. However, re-evaluating solutions should mean that we find better quality solutions as more of the search space is explored.

Again, we tested this intuition. The first problem we used was the fourteen polygon problem. Although we do not know the optimal solution for this problem the best solution we have found is 1042. We used this as a test case to see if varying the re-evaluation parameter had an effect on the algorithm finding a solution of 1042.
We used simulated annealing for these tests and did fifty runs using two values for the evaluation parameter (0.0 and 0.2).

We found that using 0.0 the algorithm found the 1042 solution 27 times out of fifty. Using a parameter of 0.2 the 1042 solution was found 34 times out of 50.

On our second test problem we ran the algorithm ten times using re-evaluation parameters of 0.0 and 0.1.

The average result when using 0.0 was 1705.32. Using 0.1 the average was 1588.46.
Again, this shows that having a small revaluation parameter does have an effect on the solution quality.

Further tests have shown increasing the re-evaluation parameter further does not have a dramatic effect on the quality of the solutions. For most of our later work we have been using a re-evaluation parameter of 0.2.

2.7 Summary

This report considers ways in which we can speed up the evaluation function that we have devised for the two-dimensional stock cutting problem.
The problem arises due to the amount of time it takes to manipulate polygons, in particular to work out the No Fit Polygon (NFP) for two polygons and then find the best placement of these two polygons by calculating the convex hull for each position of the orbiting polygon on every vertex of the NFP.
The problem is made worse as, due to the nature of meta-heuristic algorithms, these evaluations need to be carried out many times during the course of the algorithm.

In this report we have discussed the problems that computational geometry gives, in terms of computational complexity, and also described the main operations we have implemented for a point and polygon abstract data type.

Next we described the basis for our evaluation function. Essentially, this means calculating the NFP for two polygons and then finding the best placement for these two polygons via a number of convex hull calculations. This is repeated n-1 times (where n is the number of polygons in the solution).

In order to improve the run time of the algorithm the evaluations (both partial and complete) are held in a cache so that the evaluation function can be bypassed when the same solution (partial or complete) is seen again. We have shown that this method can drastically improve the run time of the algorithm.

We have also introduced the concept of types so that polygons which are the same are classified together so that the evaluation cache can make use of this when deciding if the result of the evaluation is held in the cache.

Finally, we described the need, with some probability, to force re-evaluation of a solution that might already be in the cache. This is due to the fact that a solution held in the cache may not be the best solution for a given permutation of polygons.

Our future research will make use of, and develop, the ideas we have presented in this report.

2.8 References

- Adamowicz, M., Albano, A. 1972. A Two-Stage solution of the cutting-stock problem. Information Processing, Vol. 71, pp 1086-1091.
- Adamowicz, M., Albano, A. 1976. Nesting Two-Dimensional Shapes in Rectangular Modules. Computer Aided Design, Vol 8, pp 27-33
- Art, R.C. 1966. An Approach to the Two-Dimensional Irregular Cutting Stock Problem. Technical Report 36.008, IBM Cambridge Centre.
- Chand, D.R., Kapur, S.S. 1970. An algorithm for convex polytopes. JACM vol. 17, iss. 1, pp 78-86
- Cunninghame-Green, R. 1989. Geometry, Shoemaking and the Milk Tray Problem. New Scientist, 12, August 1989, 1677, pp 50-53.
- Cunninghame-Green, R., Davis, L.S. 1992. Cut Out Waste! O.R. Insight, Vol 5, iss 3, pp 4-7
- Graham, R.L. 1972. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters, 1
- Hearn, D, Baker, P., M. 1994. Computer Graphics. Prentice Hall, New Jersey
- Laszlo, M.J. 1996. Computational Geometry and Computer Graphics in C++. Prentice Hall.
- O'Rourke, J., Chien, C.b., Olson, T., Naddor, D. 1982. A New Linear Algorithm for Intersecting Convex Polygons. Comput. Graph. Image Process. Vol. 19, pp 384-391.
- O'Rourke, J. 1994. Computational Geometry in C. Cambridge University Press.
- Preparata, F.P., Shamos, M.I., 1985. Computational Geometry: An Introduction. Springer-Verlag
- Sedgewick, R. 1992. Algorithms in C++. Addison-Wesley, Reading, Massachusetts
- Shamos, M. I., Hoey, D. 1976. Geometric Intersection Problems. Seventeenth Annual IEEE Symposium on Foundations of Computer Science, pp 208-215.
- Shamos, M.I. 1978. Computational Geometry. PhD Thesis, UMI #7819047, Yale University, New Haven, CT.