

# EVALUATION OF TWO DIMENSIONAL BIN PACKING PROBLEM USING THE NO FIT POLYGON

E. K. Burke, The University of Nottingham, UK  
G. Kendall, The University of Nottingham, UK

## ABSTRACT

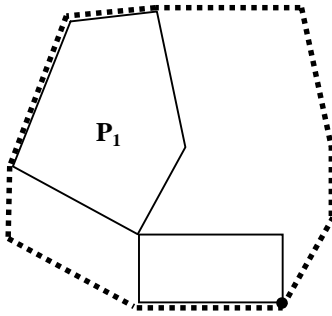
When employing evolutionary algorithms it is often the case that the evaluation function is the most computationally expensive part of the algorithm. Our evaluation function calculates the no fit polygon (NFP) for two polygons and then calculates the smallest convex hull for these two polygons. This process is repeated for each polygon. As the manipulation of polygons is computational expensive, the algorithm shows a bottleneck at this stage. However, many of the evaluations are simply reevaluating solutions that have already been evaluated. In order to use this information we use a cache which stores previous evaluations. By increasing the size of the cache size the speed of the algorithm is significantly increased. In addition the concept of a polygon type allows much better use to be made of the cache. In some circumstances, it may not be beneficial to use a cached evaluation. A reevaluation parameter is introduced which forces a complete reevaluation of a solution. We show that this parameter can be set to a small value so that we do not lose the advantages of the cache. These approaches are intuitive but are not often implemented, but they will become increasingly important as evolutionary algorithms are more widely used.

## 1. INTRODUCTION

In the nesting problem larger shapes must be divided into smaller pieces. This is achieved by arranging the smaller pieces onto the larger shapes so that they do not overlap, they lie within the confines of the larger shapes and the waste is minimised. Only two dimensions are relevant; height and width. In this paper only a single bin (larger shape) of fixed width and infinite height is used. The nesting problem is also characterized by the fact that it works with irregular shapes (polygons). In this paper only convex polygons are considered. [2] is one of the first references to discuss the nesting problem and, in particular, the No Fit Polygon (NFP) (see section 2). [1] also uses the NFP to calculate the minimum enclosing rectangle for two irregular shapes. Recently [6] also used the no fit polygon. Pieces are placed into the bin one at a time. The location of the next piece is calculated using the NFP. Once the best placement has been found the piece is added to the partial solution and the next piece is placed. The evaluation function is often the most expensive part of an algorithm. Other researchers have used various methods in order to reduce the computational load of their evaluation function on their algorithm. In [7] a warehouse scheduling problem is solved using a genetic algorithm. The evaluation function is a list based simulation of orders progressing through the warehouse. An internal (detailed) simulator is used to verify solutions. This takes about three minutes. An external (coarse) simulator runs in about one tenth of a second and is used to identify potential solutions. [8] uses *delta evaluation* on the timetabling problem. Instead of evaluating every timetable they show that, as only small changes

are being made between one timetable and the next, it is possible to evaluate just the changes and update the previous cost function using the result of that calculation. Our research uses the no fit polygon and evolutionary algorithms to produce solutions to the nesting problem. In particular, we present a method of evaluation that allows the algorithm to be speeded up by reducing the number of times the evaluation function is called.

## 2. NO FIT POLYGON



The No Fit Polygon (NFP) determines all arrangements that two arbitrary polygons may assume such that the shapes do not overlap but so that they cannot be moved closer together without intersecting. To show how the NFP is constructed consider two polygons;  $P_1$  and  $P_2$ . The aim is to find an arrangement such that the two polygons touch but do not overlap. If this can be achieved 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 the procedure is as follows (see figure 1). One of the polygons ( $P_1$ ) remains stationary.  $P_2$  moves around  $P_1$  and stays in contact with it but never intersects it.  $P_1$  and  $P_2$  retain their

original orientation. That is, they never rotate. As  $P_2$  moves around  $P_1$  one of its vertices (the filled circle) traces a line.

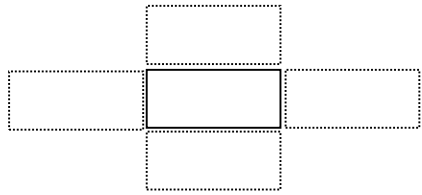
Figure 1 shows the starting (and finishing) positions of  $P_1$  and  $P_2$ . The NFP is shown as a dashed line. It is slightly enlarged so that it is visible. In fact, some of the edges would be identical to  $P_1$  and  $P_2$ . Once the NFP has been calculated for a given pair of polygons the reference point (the filled circle) of  $P_2$  can be placed 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. [4] presents the algorithm that we use.

Further details of the computational geometry techniques we have used can be found in [3].

## 3. EVALUATION

### 3.1 The Basic Method

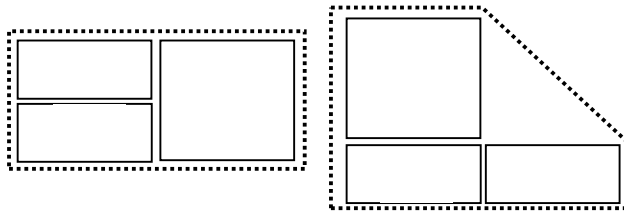
In order to fill the bin we proceed as follows. The first polygon is chosen and this becomes the stationary polygon ( $P_1$  in figure 1). The next polygon ( $P_2$  in figure 1) becomes the orbiting polygon. Using these two polygons the NFP is constructed. The reference point of  $P_2$  is now placed on various points on the NFP. For each position the convex hull for the two polygons is calculated. Once all placements have been considered the convex hull that has the minimum area is returned as the best packing of the two polygons. This larger polygon now becomes the stationary polygon and the next polygon is used as the orbiting polygon. This process is repeated until all polygons have been processed. As each large polygon is created, its width is checked. If this exceeds the width of the bin, then a new row within the bin is started. In this case the polygon which forced the width of the bin to be exceeded becomes the stationary polygon. That is, the large polygon built thus far forms one row and the next row is constructed using a single polygon as a starting point.



There are two problems that need to be addressed in order to make the evaluation strategy feasible. The number of placements the reference point can take on the NFP is infinite. In order to reduce the problem to manageable proportions, we only place the reference point on the vertices of 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 in figure 3.1.

No matter which placement is chosen 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 the placement is chosen where one polygon is placed on top of P. Depending on the characteristics of the next polygon it could effect the quality of the solution that is being built. Figure 3.2, illustrates this. It shows that by

choosing one placement over another different solutions can be obtained later in the packing.

The cost function that measures a complete (or partial) solution is taken from [5]. Rather than simply measuring the bin height (which is the intuitive evaluation function), the cost function measures how efficiently the bin has been packed. This gives a much better search space for the algorithm to explore.

### 3.2 Caching of Evaluations

Manipulating polygons is computationally expensive and, due to the nature of evolutionary algorithms, the evaluation function has to be called many times. It is likely that the same evaluation will be performed many times. This makes this part of the algorithm a serious bottleneck. In order not to evaluate previously seen solutions again a cache has been implemented. Each polygon is assigned a unique identifier. In this way a solution (complete or partial) can be recognised by considering a concatenation of the identifiers (a key), which is used to access the cache. The cache either returns a null value, meaning that the solution is not present, or it returns the previous evaluation which means the evaluation function does not have to be called. In addition a polygon data structure is held in the cache so that this can be returned and paired with the next polygon. In fact, the cache can hold more than one polygon for each hash key. Figure 2 shows four optimal placements. These placements would all be held in the cache but only one (randomly selected) is returned.

### 3.3 Types

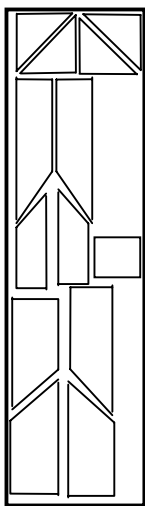
Another potential bottleneck is evaluating polygon permutations that have already been evaluated. Consider polygons with identifiers ABCDE and assume the polygons DE are identical. After evaluating the permutation ABCDE, {AB, ABC, ABCD, ABCDE} will be stored in the

cache. Later in the algorithm ABCE might need to be evaluated. This key will not be in the cache and the evaluation function is called. 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. In order to cater for this, the notion of a *polygon type* was introduced. This gives each polygon a type identifier which acts as a pointer to the polygon description. This can significantly reduce the size of the search space and allows for much more effective use of the cache.

### 3.4 Forcing Reevaluations

One problem in using a cache is that it might be holding an evaluation which is not the optimal for a particular permutation of polygons. An example of this is shown in figures 2 and 3. If the first two (smaller) polygons are evaluated, it will be found they can be positioned in four ways (see figure 2). The placement chosen is random, so assume one polygon is placed next to the other. When the third polygon is evaluated, the best configuration that can be found is that shown on the right of figure 3. It is the result from this evaluation that will be stored in the cache and whenever these three polygons are evaluated, it will be this value that is returned from the cache. We will never have access to the better solution (left hand side of figure 3) where the first two polygons are placed on top of one another. The problem is that once a configuration is stored in the cache, the only way we can arrive at another placement is for the cache to exceed its limit and for the (possibly) inferior solution to be discarded. 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 a *reevaluation* parameter is introduced. This is set to a value between 0 and 1 and determines the probability of the solution being reevaluated, regardless of whether or not it is in the cache. A value of zero means that the value in the cache, if it exists, should always be used. A value of one means that the solution is always reevaluated, effectively ignoring the cache. Higher values of the reevaluation parameter will slow the algorithm down as it is not making as much use of the cache. However, reevaluating solutions should mean that better quality solutions are found as more of the search space is explored.

## 4. Testing, Results and Comparisons

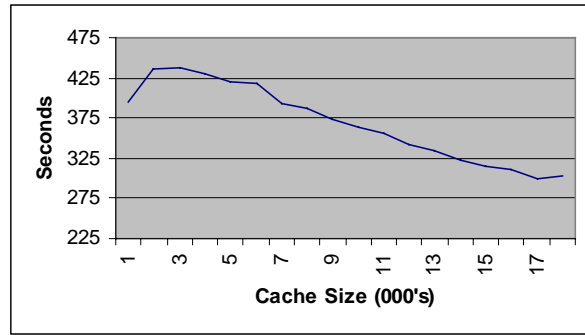
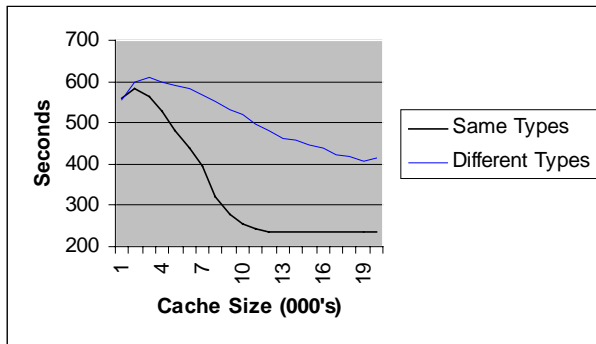


### 4.1 Test Data

Testing was carried out on three sets of data. The first test consisted of fourteen pieces which were defined using three different polygon types. The second test used the same pieces but this time the polygons were defined as (fourteen) different types. The third test used the pieces shown figure 4.1. This problem is taken from a company that cuts polycarbonate pieces for the manufacture of conservatories. This data is defined using different types. All our tests were run on a Cyrix 166 processor with 64MB of memory with the results averaged over three runs using simulated annealing.

### 4.2 Testing the Cache

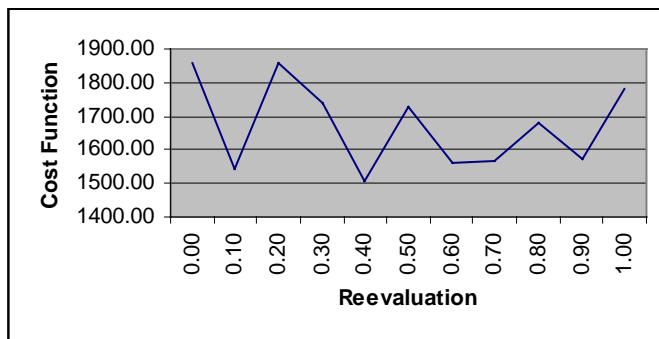
The upper bound for the cache size was found by starting with a high value and recording the maximum number of elements held in the cache at the completion of the run.



The cache size was then varied, in increments of 1000, between zero and the upper bound. It can be seen (figure 4.2 and 4.3) that the cache size has a significant effect on the running time of the algorithm. In addition, when using polygons of the same type (figure 4.2, which shows the first two sets of test data) the program not only runs faster but also requires a smaller cache (due to the reduced size of the search space). The third test problem (figure 4.3) shows similar, confirmatory results. It is interesting to note that with a cache size of zero, the algorithm is slightly slower than when the cache is set to only hold a small number of elements. We attribute this to the fact that there is an overhead in maintaining the cache. When the cache is maintaining a small number of elements the execution time for the overhead processing exceed the time benefits that result from having data stored in the cache. However, this is not a major issue as it is always better to have the cache set to as large a value as possible and a small value should never be used, only in this case to show the effect of varying its size.

#### 4.3 Reevaluation Parameter

The effect of increasing the reevaluation parameter is to increase the run time of the algorithm as cached data is not used as often. The worst case comes when the reevaluation parameter is set to



1.0 as this is the same as setting the cache size to zero and means that every solution has to be evaluated. The three sets of test data were tested using values between 0.0 and 1.0 for the reevaluation parameter, incrementing by 0.1 on each test. We would expect to see, with higher values for the reevaluation parameter, better quality solutions appearing. In fact, this was not the case as can be seen from figure 4.4, which shows the results from the third set of data but is representative of all three tests. It is difficult to draw

conclusions from this graph although we can conclude that when the reevaluation parameter is set

to zero the result is a high cost function. We conclude from this that we should not set the parameter to zero but a low value can be used (we use a value of 0.1). We plan to do more work in this area so that we can more fully understand the effect of this parameter.

## 5. CONCLUSION

An evaluation method for the nesting problem has been presented. The evaluation function is a bottleneck in some systems, which is the case in ours due to the computational geometry aspects of the algorithm. Three ways have been presented which increase the speed of the algorithm. Although these methods are intuitive we do not believe they have been presented and implemented with regards to the nesting problem in conjunction with the no fit polygon. The first improvement stores previous evaluations in a cache so that the evaluation function can be bypassed if the same solution is seen again. By varying the cache size it has been shown that the speed of the algorithm can be significantly influenced. The concept of polygon types has also been introduced. Whilst most researchers will use this method of representing their data we have demonstrated that this approach does lead to an improvement in run time, especially when used in conjunction with the cache. Finally, we used a reevaluation parameter so that we can force a solution to be reevaluated even if it is stored in the cache. This is required as it is possible that the cache will hold an inferior solution for a given permutation of polygons.

Without the techniques we have outlined above we do not believe that an evolutionary approach to the nesting problem, using our evaluation method, would be feasible. This work provides a sound foundation for our future research which will include other evolutionary algorithms and non-convex polygons.

## 6. References

- [1] M. Adamowicz and A. Albano. Nesting Two-Dimensional Shapes in Rectangular Modules. *Computer Aided Design* 8, 27-33 (1976)
- [2] R.C. Art. An Approach to the Two-Dimensional Irregular Cutting Stock Problem. Technical Report 36.008, IBM Cambridge Centre. (1966)
- [3] E.K. Burke and G. Kendall. Implementation and Performance Improvement of the Evaluation of a Two Dimensional Bin Packing Problem using the No Fit Polygon. University of Nottingham Technical Report #ASAP99001. (1999)
- [4] R. Cunnigham-Green and L.S. Davis. Cut Out Waste! *O.R. Insight*. 5, iss 3, 4-7 (1992)
- [5] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley and Sons, Chichester (1998)
- [6] J.F. Oliveira, A.M. Gomes and S. Ferreira. TOPOS A new constructive algorithm for nesting problems. Accepted for *ORSpektrum* (1998)
- [7] A. Rana, A.E. Howe, L.D. Whitley and K. Mathias. Comparing Heuristic, Evolutionary and Local Search Approaches to Scheduling. Third Artificial Intelligence Plannings Systems Conference (AIPS-96) (1996)
- [8] P. Ross, D. Corne and F. Hsiao-Lan. Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation. In Y. Davidor, H-P Schwefel and R. Manner (eds) *Parallel Problem Solving in Nature*, Vol 3, Springer-Verlag, Berlin (1994)