Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Computers & Operations Research 33 (2006) 1521–1539

computers & operations research

www.elsevier.com/locate/cor

An improved method for calculating the no-fit polygon

Hamish T. Dean*, Yiliu Tu, John F. Raffensperger

200 Armagh Street, P.O. Box 13-761, Christchurch, New Zealand

Abstract

The no-fit polygon (NFP) is the set of feasible locations that one polygon may take with respect to another polygon, such that the polygons do not overlap. Feasible locations are required for most of the solutions to two-dimensional packing problems, and also for other problems such as robot motion planning.

Efficient methods to calculate the NFP of two convex polygons, or one convex and one non-convex polygon have been developed by other researchers. However, when both polygons are non-convex, the current methods of calculation are inefficient or difficult to implement. This paper presents an extension of Ghosh's (CVGIP: Image Understanding 54(1991)119) NFP algorithm, and uses manipulation of sorted lists of polygon edges to find the NFP efficiently.

Keywords: No-fit polygon; Two-dimensional packing; Minkowski sum

1. Introduction

An issue in two-dimensional packing is determining the set of feasible locations that one polygon may take with respect to another polygon, such that the polygons do not overlap. This set of locations is known as a *no-fit polygon* (NFP). The terms *Minkowski sum*, *Φ-function*, *hodograph*, *dilation*, *envelope* and *configuration space obstacle* have also been used by other researchers.

Let each polygon be represented by an ordered list of edges. The location of each polygon i in the two-dimensional plane is represented by a reference point, r_i. The reference point is located at point (0, 0) of a polygon's local coordinate system (see Fig. 1).

* Corresponding author. Tel.: +64 3 377 3140.
E-mail address: hamish@shapeshifter.net.nz (H.T. Dean).

0305-0548/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2004.11.005

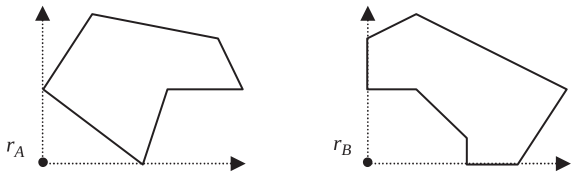

Fig. 1. Polygon reference points.

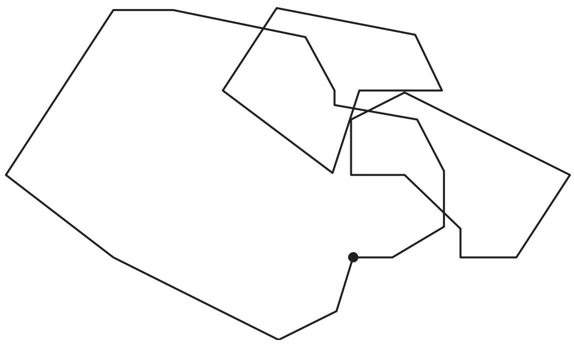

Fig. 2. Reference point of polygon j on NFP[A, B].

The perimeter of the NFP of polygon A and polygon B (denoted as NFP[A, B]) gives the points that r_B can take such that polygon B is touching polygon A. If r_B is located inside NFP[A, B] then the two polygons overlap. Conversely, if r_B is outside NFP[A, B] then the two polygons do not overlap, and do not touch (see Fig. 2).

Several publications (such as O'Rourke [1]) have shown the relationship between a form of vector addition known as the Minkowski sum and the NFP. If we let the vertices of polygons A and B be represented as vectors, then the Minkowski sum of A and B is defined in Eq. (1):

$$A \oplus B = \{x + y | x \in A, y \in B\}, \tag{1}$$

where $x + y$ is the vector sum of points x and y.

Geometrically, the *outer envelope* of the Minkowski sum of A and $-B$ is the equivalent of NFP[A, $-B$][1] (see Fig. 3). $-B$ can be obtained by rotating B by 180° or multiplying B by -1.

In most two-dimensional packing algorithms, many NFPs must be calculated. These calculations are computationally expensive (most of the computational time is spent in calculating the outer envelope of the Minkowski sum). The number of edges in a Minkowski sum is $2mn$, where m and n are the number of edges on polygons A and B, respectively. In industrial cases polygons may have in excess of 100 edges. Polygons of this complexity result in Minkowski sums of very large size, and therefore require a time-consuming process to create the corresponding NFP. Because of this, many researchers have tried to calculate NFPs using different methods (other than the Minkowski sum).

[1] From now on NFP[A, $-B$] will be referred to as NFP[A, B].

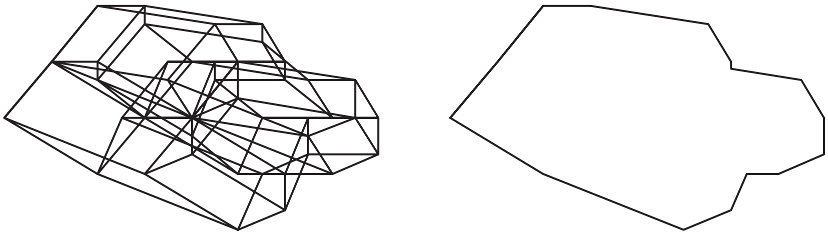

Fig. 3. Minkowski sum of A, $-B$, and NFP[A, $-B$].

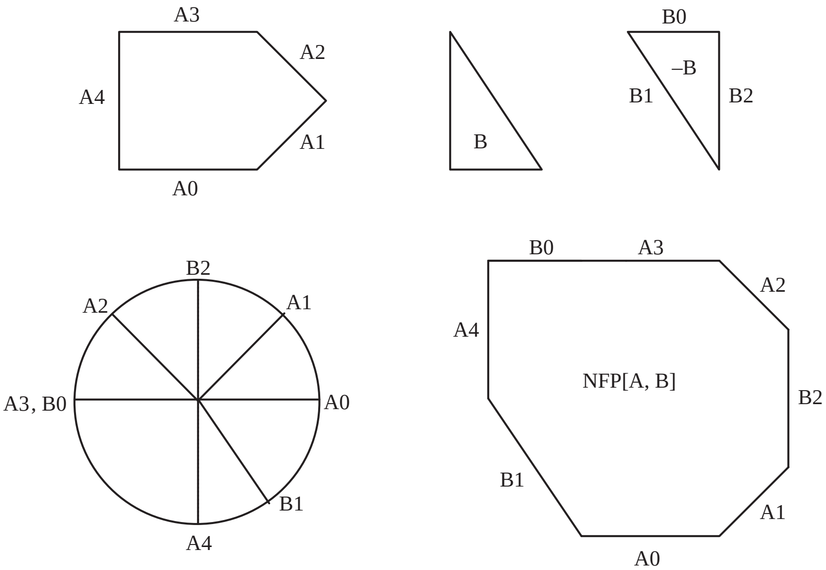

Fig. 4. Polygon edge slope order is equivalent to NFP edge order.

Cunninghame-Green [2] showed that for the case when polygons A and B are convex, NFP[A, B] can be created by ordering the edges of A and $-B$ in increasing slope order. NFP[A, B]'s edges correspond exactly to this slope order (see Fig. 4).

When one or more of the polygons are non-convex, an obvious way of calculating the relevant NFP is to decompose each polygon i into a set of N_i convex sub-polygons ($\mathrm{CSP}[i]_1 \rightarrow \mathrm{CSP}[i]_{Ni}$). Overlap will occur between the two polygons if any sub-polygon of A overlaps any sub-polygon of B. NFP[A, B] is the union of NFP[$\mathrm{CSP}[A]_i,\ \mathrm{CSP}[B]_j$], where $i = 1 \ldots N_A$ and $j = 1 \ldots N_B$.

There are two drawbacks to the polygon subdivision approach. Firstly, efficient algorithms are required for polygon decomposition and polygon composition. Secondly, it is possible that a non-convex polygon that has N edges in cavities (see Fig. 5) can be decomposed into no less than N CSPs. The NFP of two of these polygons would require the composition of N^2 sub-NFPs. Polygons used in industries such as garment manufacturing often have large numbers of edges in their curve-like cavities, and the sub-division

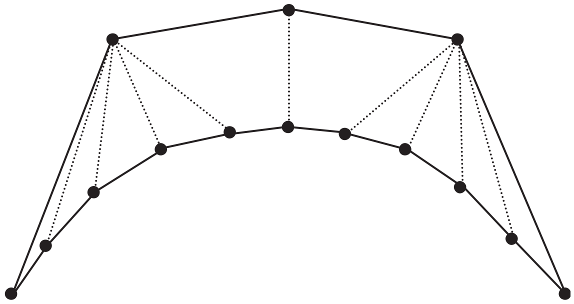

Fig. 5. A polygon with 10 edges in decomposed into 10 convex sub-polygons.

technique becomes inefficient compared with the slope-based techniques described later in this paper. A method of the convex subdivision technique is given by Lo Valvo [3].

Mahadevan [4] describes a different method for calculating the NFP for two non-convex polygons. The basis of this algorithm is that one polygon orbits another, and at each position the reference point of the orbiting polygon is stored. These stored points become the vertices of the NFP. In implementing this algorithm, Kendall [5] found degenerate cases. Kendall describes these degenerate cases, and how they have been overcome. The main drawback of the orbiting polygon approach is that as the orbiting polygon slides along an edge of the stationary polygon, a test must be performed in order to calculate the sliding distance. This is because for non-convex polygons, the sliding distance is not always equal to the stationary edge length. The test involves extending every vertex on the orbiting polygon in the direction of motion by the length of the sliding edge. Extended vertices are then checked for intersections with the stationary polygon. For polygons with a large number of vertices, this method can be computationally expensive. Also, an orbiting method may not detect that polygon B may be placed in a cavity of polygon A, when polygon B cannot slide in from the outside.

2. Ghosh's approach

The method presented in this paper is an extension of the algorithm given by Ghosh [6]. The method is based on the fact that the NFP of any two polygons is a function of their boundary edges. An outline of this algorithm is now given.

Firstly, we give some definitions and starting conditions used in Ghosh's method and the remainder of the paper:

Condition 2.1. The edges of polygon A are ordered anti-clockwise, starting at the lowest, leftmost edge.

Condition 2.2. The edges of polygon B are ordered anti-clockwise, starting at the lowest, leftmost edge. B is then inverted to give $-B$. $-B = (-1)^* B$.

Definition 2.1. If edge i extends from point D to point E, and edge $i+1$ extends from point E to point F, then $\alpha(i) = DE \times DF$.

Definition 2.2. An edge i of a polygon is a *turning point* if the sign of $\alpha(i)$ is opposite to the sign of $\alpha(i+1)$.

Bennell et al. [7] states that a polygon is *convex* if and only if it does not contain any turning points. Otherwise it is *non-convex*.

The initial stage of Ghosh's approach is to sort all the edges of polygon A and polygon B by slope into one list which we will call *MergeList*. If both polygon A and B are convex, then MergeList gives the edge order for NFP[A, B], and the method is equivalent to that of Cunninghame-Green [2].

Assuming polygon A is non-convex, and polygon B is convex, the method proceeds as follows:

Starting in MergeList at the first edge of polygon A, visit the edges of A in order, and add them to the list of edges (*NFPList*) which make up NFP[A, B]. If edge A is a turning point, then the direction of travel along MergeList is reversed. Any edges of B which are passed are added to NFPList. B edges are positive if the direction of travel forward, and negative if the direction is backward. This continues until the first edge of polygon A has been returned to. The resulting NFPList we will call *GhoshList*. The above algorithm is given in Pseudocode 2.1.

$p =$ Position in MergeList which corresponds to $A0$
$i = 0$
$Dir = 1$
Loop{
If MergeList$[p]$.$PolygonType = A$ **Then**
    If MergeList$[p]$.$PolygonIndex = i$ **Then**
        GhoshList $=$ GhoshList $+$ MergeList$[p]$
        If MergeList$[p]$.$IsTurningPoint = True$ **Then** $Dir = Dir^* - 1$
        $i = i + 1$ (**If** $i > A.Size$ **Then** $i = 0$)
    End If
Else
    GhoshList $=$ GhoshList $+$ MergeList$[p]^*Dir$
End If
$p = p + Dir$
}**While**($i \neq 0$)

Pseudocode 2.1: Algorithm to find GhoshList

The process of finding GhoshList is seen easily with what Ghosh calls a slope diagram (see Fig. 6). The points on the diagram are at the slope of the edges of polygons A and B.

Following around the slope diagram, starting and finishing at $A0$, mimics the process of traversing over MergeList. The outer envelope of NFPList gives NFP[A, B] (see Fig. 7).

Ghosh's method works for all simple polygons (no holes) when polygon A is non-convex and polygon B is convex. The method also works when both polygons are non-convex, as long as no two cavities from either polygon interfere which each other. This occurs when an interval of MergeList has wrongly ordered edges from both polygons. When this does occur, this interval must be traversed in two or more parallel paths. Although the theory of traversal by parallel paths holds true for complex non-convex cases, there are considerable implementation problems in sorting out the paths. These difficulties led Bennell et al. [7] to seek a different approach.

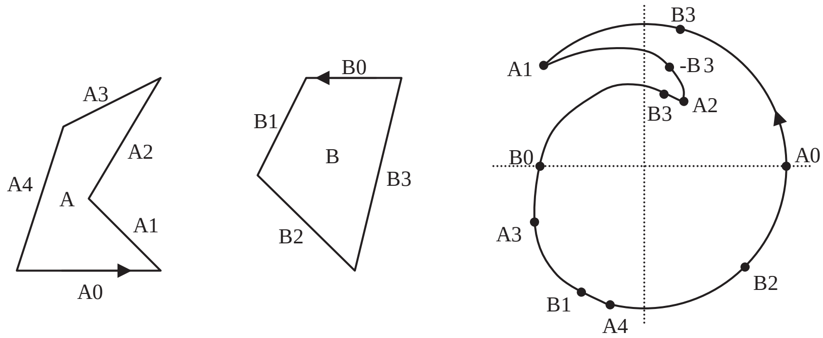

Fig. 6. Slope diagram of polygons *A* and *B*.

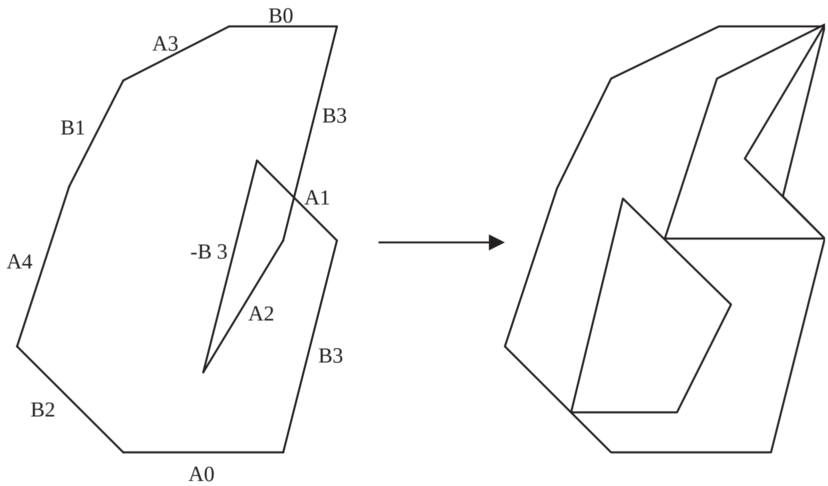

Fig. 7. NFPList and resulting NFP.

Their approach exploits the fact that the NFP of a non-convex polygon and a convex polygon can be easily and efficiently found by Ghosh's method. When both polygons are non-convex, the convex hull of polygon *B* (conv[*B*]) is used. Conv[*B*] can be regarded as a copy of *B* with its cavities replaced by dummy edges. Ghosh's method is then used to create an edge listing (GhoshList) for NFP(*A*, conv[*B*]). GhoshList may include both positive and negative occurrences of the dummy edges.

For each type of dummy edge, GhoshList is split into segments containing a positive or negative dummy edge. Each occurrence of a dummy edge is then replaced by a combination (ReplaceList) of the *B* edges from which the dummy edge was derived (edges $B_{\mathrm{CavStart}} \rightarrow B_{\mathrm{CavFin}}$), and *A* edges within the segment. Starting at the dummy edge, all occurrences of B_{CavStart} within the segment are "found" and added to ReplaceList before moving on to finding $B_{\mathrm{CavStart}+1}$. Any *A* edges "passed" on the way are also added to ReplaceList. This is continued until all B_{CavFin} edges have been found, and the dummy edge has been returned to. The dummy edge in GhoshList is then replaced by ReplaceList.

Bennell's method works well when the edges in a *B* cavity occur in slope order. However, if the *B* edges within a cavity are out of slope order an incorrect NFP is occasionally calculated.

The calculation difficulties of Bennell's method has motivated development of a more robust and efficient method of calculating NFPs. Like Bennell's method, it exploits the fact that the NFP of a non-convex polygon and a convex polygon can be easily and efficiently found by Ghosh's method. However, the new method does not use dummy edges to replace cavities of *B*.

3. A new method

Intuitively, it would seem a good idea to modify Bennell's method to start "looking" for the next *B* edge of a cavity once an occurrence of the current *B* edge has been found, instead of continuing to look for the furthest occurrence of that *B* edge. However, if there is more than one occurrence of a *B* edge in any given segment then this approach will run into difficulties.

A solution to this is to make sure that each traversal segment contains only positive or negative occurrences of each *B* edge of a particular cavity. Replacing a *B* cavity with a dummy edge *D* will not guarantee this (see Fig. 8).

Fig. 8 shows a dummy edge *D*, whose cavity is composed of edges $B1$ and $B2$. In this example, Bennell's method would require only one segment which would contain a single occurrence of *D*. However, this segment contains both positive and negative occurrences of $B1$.

To guarantee that there is only positive or negative occurrences of a given cavity *B* edge, we split the traversal of GhoshList using the algorithm given in Pseudo-code 3.1:

$p =$ The position in GhoshList which corresponds to $A0$.
$TravelDir = -1$
$CurrentSign = +1$
$TravelSign = +1$
$i = 1$
Loop1{
    $p = p + TravelDir$
    If GhoshList[p].*PolygonType* $= A$ **Then**
        If GhoshList[p].PolygonIndex $= 0$ **And** TravelDir $= -1$ **Then**
            Seg[i].*End* $= p$
            Exit Algorithm
        Else If GhoshList[p].*IsTurningPoint* $=$ True **Then**
            $TravelSign = TravelSign^* - 1$
            $Seg[i].Start = p$
        End If
    Else If GhoshList[p].*IsInCavity* $=$ True **And** $CurrentSign \neq TravelSign$
    Then
        $TravelDir = TravelDir^* - 1$
        $p = Seg[i].Start$
        $TravelSign = TravelSign^* - 1$
        Exit Loop1
    End If
}

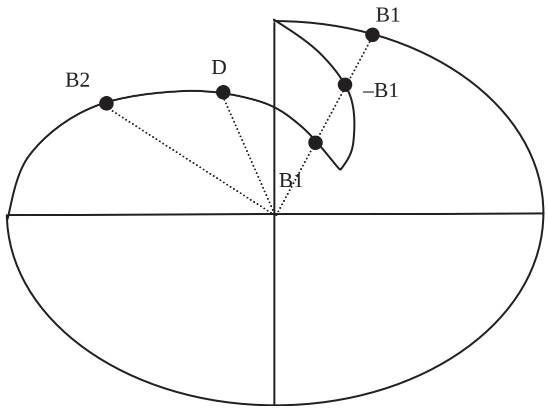

Fig. 8. $B1$ occurring more than dummy edge D.

Loop2{
    $p = p + TravelDir$
    If $p = Seg[1].Start$ **Then**
        Seg[i].$End = p$
        Exit Algorithm
    Else If GhoshList[p].$PolygonType = A$ **Then**
        If GhoshList[p].$IsTurningPoint =$ True **Then**
            $TravelSign = TravelSign^* - 1$
            $Seg[i].End = p$
        End If
    Else If GhoshList[p].$IsInCavity =$ True **And** $CurrentSign \neq TravelSign$
    Then
        $i = i + 1$
        $Seg[i].Start = Seg[i-1].End$
        $CurrentSign = CurrentSign^* - 1$
    End If
}

Pseudo-Code 3.1: Algorithm to split GhoshList into segments

The above algorithm splits GhoshList into i segments, separated at certain turning points of A. The segments are split so that each segment contains only positive or only negative occurrences of B edges which belong to cavities.

A cavity is an ordered list of edges starting at an edge which is not on the convex hull of its polygon, but whose starting point is, and contains all edges up to, and including, the next edge that is not on the convex hull, but whose ending point is. The two cavities of polygon B are shown in Fig. 9.

The *span* of a cavity k is the arc spanning the farthest clockwise and farthest anti-clockwise edges of k in the polygon's slope diagram. An example is shown in Fig. 9. Cavity 1's span is between $B1$ and $B2$, and cavity 2's span is between $B7$ and $B6$.

We say that a cavity k of B *interferes* with segment i if the span of cavity k intersects segment i.

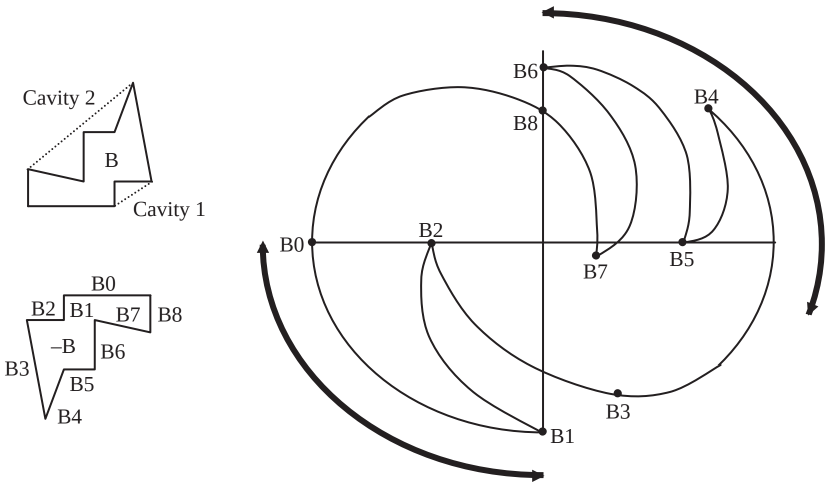

Fig. 9. Spans of cavities 1 and 2.

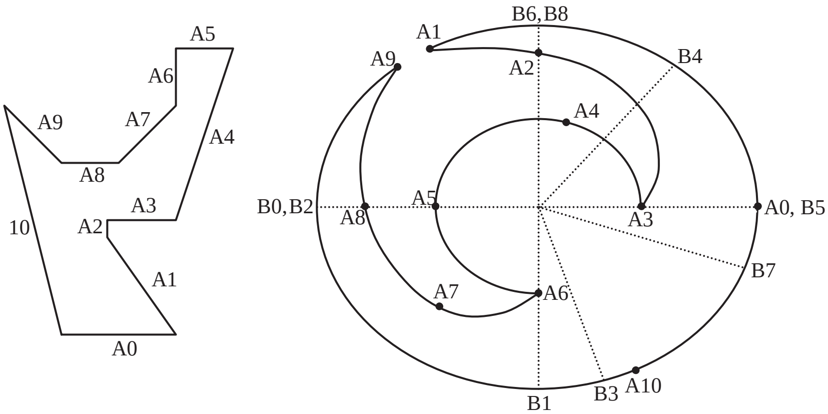

Fig. 10. Slope diagram of GhoshList.

Fig. 10 shows the slope diagram for polygons A and B. It will be split into four segments, $A9 \rightarrow A1$, $A1 \rightarrow A3$, $A3 \rightarrow A6$, and $A6 \rightarrow A9$, using the method described earlier. For each of the four segments, we count the cavities of B whose span intersects it. Cavity 1 intersects segments $A9 \rightarrow A1$, $A3 \rightarrow A6$, and $A6 \rightarrow A9$. Cavity 2 intersects $A9 \rightarrow A1$, $A1 \rightarrow A3$, and $A3 \rightarrow A6$. Each segment must "process" the cavities which intersect it.

To create NFPList with correct numerical ordering of both A and B edges, GhoshList and MergeList must be re-traversed together.

3.1. Finding the starting point

We start the re-traversal in GhoshList at a B edge which is either not part of a B cavity, or is the first edge of a B cavity. Because $B0$ starts at the lowest leftmost vertex of B (therefore its starting vertex is on conv[B]), at least one of these conditions will always hold for $B0$. $+B0$ will appear in GhoshList at least once, and it is arbitrary which occurrence of $+B0$ is selected as the starting point. The A segment is selected which contains the chosen starting point as the *current segment*. Our starting direction of traversal is backwards (the same direction as travelling $A1 \rightarrow A0$) if $B0$ is a turning point, otherwise it is forwards.

3.2. Traversing over a segment

As stated earlier, all cavities which intersect the current segment must be "processed". All non-cavity B edges which lie on the segment of GhoshList which corresponds to the current segment must also be processed. Once all intersecting cavities in the current segment and non-cavity B edges have been processed, then we can move onto the next segment. Starting at $B0$, we search for the edges of intersecting cavities and non-cavity B edges in numerical order by traversing back and forth (direction is reversed if the B edge is a turning point) between these edges, taking note of any A edges which are passed, and the direction that they are passed in.

From the example in Fig. 10, if we choose $B0$ (on segment $A9 \rightarrow A1$) as our starting point, then we traverse from $B0$ to $B1$ in a forward direction. When $B1$ has been reached, we have entered cavity 1. All edges of this cavity must be found, even if this involved traversing off the boundaries of the current segment. We then turn backward to find $B2$. Cavity 1 has now been processed, as all its constituent edges have been found. We now turn forwards, heading for $B3$. Once $B3$ is found we continue forward towards $B4$, taking note of passing $A10$ and $A0$ in the forward direction. At $B4$ we turn backward to find $B5$, and then turn forward again to find $B6$. At $B6$ we turn backward to find $B7$, and in the process pass $A0$ in the backward direction. At $B7$ cavity 2 as been processed. We turn forward to find $B8$ and then reach $A1$ while in search of $B0$. Because $A1$ is the end of the current segment, and all cavities and non-cavity B edges intersecting the current segment have been found, we move on to the next segment, $A1 \rightarrow A3$. The edges added to NFPList in the traversal of the $A9 \rightarrow A1$ segment are $\langle B0, B1, B2, B3, A10, A0, B4, B5, B6, -A0, B7, A0, B8, A1\rangle$.

This process is continued through all segments, until the initial segment is re-entered, and the starting point found.

Note. The segment $A1 \rightarrow A3$ runs in a clockwise direction. Any B edges found in a segment (or part segment) whose A edges run in a clockwise direction are negative, and the order they are found in is also reversed. So for segment $A1 \rightarrow A3$ we find the B edges $-B8 \rightarrow -B4$.

3.3. Special cases

The above method needs further explanation when either of the following situations occur: Traversal Moves Onto MergeList, or Cavity Before $B0$.

3.4. Traversal moves onto MergeList

If either the starting or ending limit, L, of the current segment is reached, and all relevant cavities and non-cavity B edges have not been processed, then the traversal shall continue on MergeList rather than on GhoshList. The traversal starts on MergeList at edge L and continues searching for the current B edge in the direction that the traversal on GhoshList was taking. Any A edges encountered on MergeList are ignored. The traversal will return back to GhoshList when L is passed in the *opposite* direction of which MergeList was entered. L is added to NFPList.

A degenerate case can occur which causes the traversal of MergeList to continue without ever returning to L. The happens when, directly after the last edge of the final cavity of the current segment has been added to NFPList, the traversal does not head back to L in the opposite direction to which MergeList was entered. Because this is the final cavity of the current segment, after processing this cavity, we want to move onto the next segment. And because the final cavity of the current segment is also the first cavity of the next segment, we can make the transition from the current segment to the next segment within MergeList. After the last edge of the final cavity is added to NFPList, we add L to NFPList (opposite sign to previously added L). The next segment becomes the current segment, and we traverse back over MergeList starting at the last edge of the final cavity (now the first edge) and add B edges to NFPList in order until L is reached in the *opposite* direction of which MergeList was entered. L is added to NFPList.

3.5. Cavity before B0

As stated earlier, we start our traversals of GhoshList and MergeList at an arbitrary occurrence of $+B0$. However, often the first cavity we come across is not actually the "first" cavity of the starting segment. That is, that cavity is not the first cavity which should be processed after processing all cavities in the previous section. $+B0$ is used because it is either not part of a B cavity or is the first edge of a B cavity. Instead of moving to the next segment when all the cavities and non-cavity B edges have been processed, for the first segment we move to the next segment when we have reached the "final" cavity. That is, the cavity which should be processed directly prior to moving onto the next segment.

To determine the final cavity of the first segment, we process the cavities as per usual. If, during the traversal of MergeList, we start and finish processing a cavity which intersects the current segment without leaving MergeList, then the final cavity is the cavity which was processed directly prior to this. An example is shown in Fig. 11.

Here the B cavity involving $B2$ and $B3$ intersects the starting segment $A3 \rightarrow A2$ twice. However, $+B0$ is located between these two intersections. Starting at $+B0$, we pass $A6$, $B1$, $A7$, $A0$, $B2$, $A1$, $A2$, and $B3$. At this point we have processed one intersection (CavA) of the B cavity. We continue our traversal past $-A2$, $B4$, and $A2$. We have now reached the end of our segment, but CavB has not yet been processed, so we move onto MergeList. We pass $B0$, $B1$, $B2$, and $B3$, without leaving MergeList. Now we have processed CavB, but it was done entirely within MergeList, so our final cavity of the starting segment is CavA.

So the initial partial traversal of segment $A3 \rightarrow A2$ adds edges $\langle B0, A6, B1, A7, A0, B2, A1, A2, B3, -A2, B4, A2\rangle$ to NFPList, processing CavA. The traversal of segment $A2 \rightarrow A3$ adds edges $\langle -B4, -A2, -B3, A2, -B2, A3\rangle$ to NFPList. The final partial traversal of segment $A3 \rightarrow A2$ processes CavB, and adds edges $\langle B2, A4, B3, B4, A5\rangle$ to NFPList.

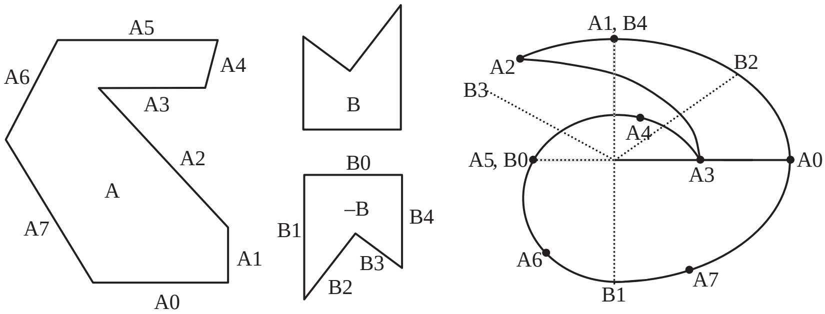

Fig. 11. Example of $+B0$ located between two concavities.

Once GhoshList has been split into segments (Pseudo-code 3.1), Pseudo-code 3.2 gives the remainder of the algorithm.

p = Position of arbitrary occurrence of $+B0$ in GhoshList.
CurrSeg = *OrigSeg* = Segment containing the arbitrary occurrence of $+B0$.
BMulti = +1
Dir = +1
NextB = 0
Loop1{
    If GhoshList[p].*PolygonType* = B **Then**
        If GhoshList[p].*PolygonIndex* = *NextB* **Then**
            NFPList = NFList + GhoshList[p]**BMulti*
            NextB = *NextB* + *BMulti*
            If GhoshList[p].*IsTurningPoint* = True **Then** *Dir* = *Dir** − 1
        End If
    Else
        NFPList = NFPList + GhoshList[p]**Dir*
        If GhoshList[p] = *Seg*[*CurrSeg*].*Start* **Then**
            GoTo TraverseMergeList()
        Else If GhoshList[p] = *Seg*[*CurrSeg*].*Fin* **Then**
            If Seg[CurrSeg].CavitiesLeft = 0 **Then**
                CurrSeg = *CurrSeg* + 1
                BMulti = *BMulti** − 1
                NextB = *NextB* + *BMulti*
            Else
                GoTo TraverseMergeList()
            End If
        Else If GhoshList[p].*IsTurningPoint* = True **Then**
            BMulti = *BMulti** − 1
            NextB = *NextB* + *BMulti*
        End If

```
        End If
        p = p + Dir
        If GhoshList[p] = B0 And NextB = 0 And CurrSeg = OrigSeg And
        Seg[CurrSeg].CavitiesLeft = 0 Then Exit Algorithm
}

TraverseMergeList{
        OrigPos = Pos = Position of p in MergeList
        OrigDir = TotalDir = MergeDir = BMulti*Dir
        Loop2{
                Pos = Pos + MergeDir
                If MergeList[Pos].PolygonType = B Then
                        If MergeList[Pos].PolygonIndex = NextB Then
                                NFPList = NFPList + MergeList[Pos]*BMulti
                                NextB = NextB + BMulti
                                If MergeList[Pos].IsTurningPoint = True Then
                                        MergeDir = MergeDir* − 1
                                End If
                                If CurrSeg = OrigSeg And Condition1() = True
                                Then
                                        CavityBeforeB0 = True
                                        Exit Loop2
                                Else If MergeDir = OrigDir And
                                Condition2() = True Then
                                        Exit Loop2
                                End If
                        End If
                Else If Pos = OrigPos Then
                        TotalDir = TotalDir + MergeDir
                        If TotalDir = 0 Then
                                Dir = Dir* − 1
                                NFPList = NFPList + MergeList[Pos]*Dir
                                Return To Loop1
                        End If
                End If
        }
        If CavityBeforeB0 = True Then
                Remove from NFPList all B edges just added in TraverseMergeList
                up to the penultimate cavity added
        End If
        NFPList = NFPList + MergeList[OrigPos]* − 1*OrigDir
        BMulti = BMulti* − 1
        Return To Loop1
}
```

```
Condition1{
        If every edge of the current cavity was found without leaving MergeList
        Then
                Return True
        Else
                Return False
        End If
}

Condition2{
        If the current cavity is the final cavity of the current segment Then
                Return True
        Else
                Return False
        End If
}
```

Pseudocode 3.2: Algorithm to find NFPList

4. Computing the outer envelope

Once we have found NFPList, the outer envelope of this must be found to find the final NFP. There are a number of algorithms available to do this, for example Hershberger [8]. However, because these algorithms have a complexity which is greater than linear, it is advantageous to use a divide and conquer strategy, splitting the problem into sub-problems.

Theorem 4.1. *If an edge, E, of polygon A is a member of* conv$[A]$, *then E is also a member of* NFP$[A, -B]$.

Proof. Edge E is a member of conv$[A]$. This implies that no other A edge, or part of edge, lies in half plane, H, bounded by the line tangential to E and containing the outward normal of E. Let polygon B lie somewhere in H. Now translate the line tangential to E in the direction of the outward normal of E until a vertex V of polygon B is hit. If V is in contact with E, then every other point on polygon B is contained in H. By definition, the boundary of NFP$[A, -B]$ are the points which the reference point of polygon B can take such that polygon A is touched. The points in H which satisfy this are edge E, as V can contact the entirety of E while remaining in H. Therefore, NFP$[A, -B]$ must contain edge E. $\square$

Theorem 4.2. *If an edge, E, of polygon B is also a member of* conv$[B]$, *then E is also a member of* NFP$[A, -B]$.

Proof. Bennell et al. [7] show that NFP$[A, -B]$ is equal to NFP$[B, -A]$ rotated by 180°. Theorem 4.1 states that any edge E which is a member of conv$[A]$ is a member of NFP$[A, -B]$. Because NFPList$[B, -A]$ and NFPList$[A, -B]$ are equivalent, every edge on conv$[B]$ that is also on polygon B appears on NFP$[A, -B]$. $\square$

Theorem 4.3. *The outer envelope of the polygon described by* NFPList$[A, -B]$ *can be constructed without negative edges of* NFPList$[A, -B]$.

Proof. The outer envelope of the Minkowski sum of A and $-B$ is equivalent to the outer envelope of polygon described by NFPList$[A, -B]$. Eq. (1) states that the Minkowski sum of A and $-B$ is the result of vector additions of all combinations of points from A and points $-B$. Consequently, the outer envelope of NFPList$[A, -B]$ can then be constructed using only positive edges of A and $-B$. Therefore, the outer envelope of the polygon described by NFPList$[A, -B]$ can be constructed without negative edges of NFPList$[A, -B]$. □

Using Theorems 4.1 and 4.2 we can reduce the number of calculations required to find the outer envelope because we know that edges that lie on the convex hull of their respective polygons occur at least once on the outer envelope. However, because construction of NFPList can give rise to multiple positive and negative copies of these edges, we need to establish some rules to determine which of these occurrences are actually members of the outer envelope.

An edge E is a member of its respective polygons convex hull, conv(P), then it will occur on NFP$[A, -B]$ if it is sliding along a convex vertex of the other polygon Q. For this to occur, the following conditions must hold:

Condition 4.1. E is non-negative.

Condition 4.2. E is a member of conv$[P]$.

Condition 4.3. If the Q edge that precedes E goes from point s to point t, then t must be a point on conv$[Q]$.

Condition 4.4. If the Q edge following E goes from point u to point v, then u must be a point on conv$[Q]$.

Condition 4.5. The angle of E must be between the angles of the Q edges that precede and follow E. If those Q edges are not a member of conv$[Q]$, then we use the convex edge that would replace them.

We can now divide the problem of finding the outer envelope of NFPList into sub-problems of finding the outer envelope of each set of edges between the edges already identified to lie on the outer envelope. We can further reduce the number of candidate edges for the outer envelope by removing negative edges using Theorem 4.3.

An example of the reduction in calculation is shown below in Fig. 12.

NFP$[A, B]$ is the equivalent of OE$[B0, B1, A4, A5, B2, B3, A0, A1, A2, B4, -A2, B5, A2, A3]$, where OE$[x]$ represents the outer envelope of x. However, using Theorems 4.1 and 4.2 we can identify that edges $A4$, $A5$, $A2$, $A2$, and $A3$ are on outer envelope. Using Theorem 4.3 we can discard edge $-A2$. So the calculation of the edge list of NFP$[A, B]$ can be simplified to $\langle$OE$[B0, B1]$, $A4$, $A5$, OE$[B2, B3, A0, A1]$, $A2$, OE$[B4, B5]$, $A2$, $A3\rangle$.

5. Adjusting for parallel edges

The method given in this paper creates NFP$[A, B]$, such that when the reference point of B is touching the perimeter of NFP$[A, B]$, polygons A and B touch. In two-dimensional packing problems, often it is

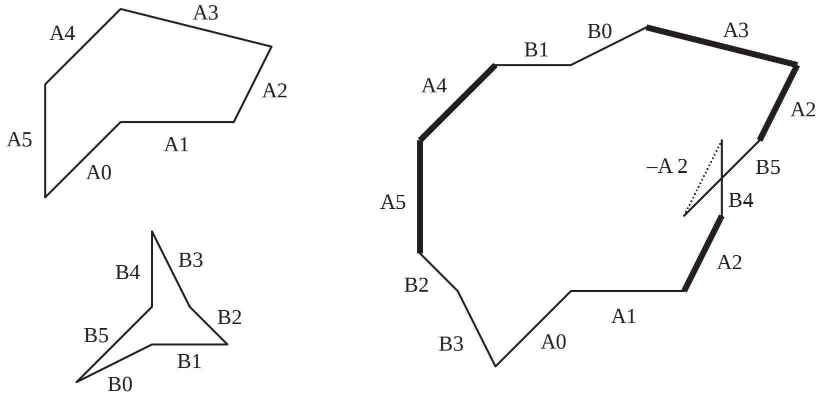

Fig. 12. NFPList for polygons A and B.

the case that polygons must be separated by a distance G (due to width of cutting blades etc.). This is simple, and is just a case of buffering NFP$[A, B]$ by G.

Some cutting machines do not require parallel edges from two polygons to be separated. In this case, $G = 0$ for the two parallel edges. Any given point on NFP$[A, B]$ implies that polygons A and B are touching. However, no information is given about whether the contact is vertex–vertex, vertex–edge, or edge–edge (two parallel edges), so there is no way of determining whether $G = 0$, or $G \neq 0$.

Distinguishing areas of the NFP which arise from edge–edge contact can be achieved by “flagging” certain edges in the construction of NFPList. If an edge from A and an edge from $-B$ have the same slope, and are adjacent in NFPList, then this signifies a possible area of NFP$[A, B]$ where the edges are parallel. These candidate edges are flagged. In construction of the outer envelope, if a candidate edge has been flagged and it appears in the outer envelope, then the points on the outer envelope containing this edge are allowed a buffer of $G = 0$, if applicable.

This method can be easily adjusted if there is a tolerance β on the parallelity of the edges. For example, if $\beta = 0.1°$, and edges of A and $-B$ are 45° and 45.05°, then these edges are considered to be parallel.

6. Computational results

In previously published papers on NFP calculation, the practice has been to test NFP algorithms on a common set of data. The data sets which have been used tend to contain simple polygons, which contain very few edges. However, in industries such as garment manufacturing, individual polygons can have over 100 edges. The algorithms in this paper were designed to improve the robustness and efficiency of calculating NFPs for polygons which have a large number of edges and many cavities. Therefore, instead of testing on the previously used data sets, we will set a performance benchmark on a new, more complex, set of data (see Fig. 13).[2] This data set will be available from the author.

The properties of each polygon is given in Table 1. The calculation results of the NFPs for each polygon pair is given in Table 2.

[2] Due to the size of the polygon drawings, some small cavities may appear invisible.

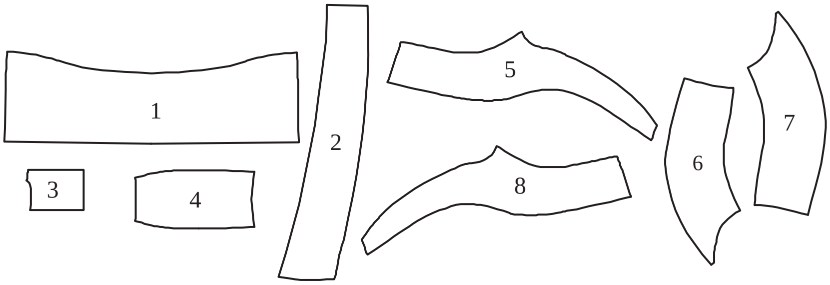

Fig. 13. Data set.

Table 1
Polygon properties

Polygon	Edges	Cavities
1	65	9
2	43	2
3	21	3
4	58	17
5	128	9
6	64	3
7	64	3
8	141	11

Table 2
NFP calculation results

A	B	Edges	Non-negative edges	Time 1 (s)	Time 2 (s)	% Impr
1	1	138	134	0.016	0.007	57.50
1	2	132	120	0.017	0.007	60.47
1	3	162	124	0.022	0.010	55.56
1	4	265	189	0.032	0.012	61.25
1	5	525	359	0.068	0.033	51.18
1	6	421	275	0.036	0.018	49.45
1	7	201	165	0.025	0.014	41.94
1	8	506	356	0.070	0.032	53.71
2	2	260	173	0.020	0.008	61.22
2	3	104	84	0.006	0.002	57.14
2	4	143	122	0.014	0.005	66.67
2	5	249	210	0.037	0.014	61.29
2	6	253	180	0.020	0.010	50.00
2	7	319	213	0.024	0.012	51.67
2	8	212	198	0.043	0.014	66.67

Table 2 (*continued*)

A	B	Edges	Non-negative edges	Time 1 (s)	Time 2 (s)	% Impr
3	3	54	48	0.002	0.001	40.00
3	4	159	119	0.012	0.006	53.33
3	5	277	213	0.037	0.018	52.17
3	6	373	229	0.024	0.014	42.62
3	7	125	105	0.010	0.005	45.83
3	8	206	184	0.035	0.014	61.36
4	4	254	185	0.024	0.008	66.10
4	5	1080	633	0.125	0.055	55.77
4	6	304	213	0.029	0.013	54.17
4	7	218	170	0.024	0.010	55.93
4	8	1157	678	0.144	0.050	65.56
5	5	1532	894	0.244	0.067	72.46
5	6	508	350	0.071	0.037	47.75
5	7	340	266	0.054	0.033	38.06
5	8	1609	939	0.291	0.081	72.25
6	6	210	169	0.024	0.007	71.67
6	7	500	314	0.049	0.027	45.08
6	8	473	339	0.075	0.024	68.45
7	7	216	172	0.021	0.008	64.15
7	8	459	332	0.078	0.024	69.59
8	8	2680	1481	0.444	0.102	77.01
Sum		16 624	10 935	2.265	0.802	64.61
Average						57.36

Removing negative edges reduced the number of candidate edges for the outer envelope by over 34% on average.

The divide and conquer strategy for calculating the outer envelope reduced the total calculation time by 64.61%. Columns "Time 1" and "Time 2" of Table 2 show the calculation times for the standard outer envelope calculation, and the divide and conquer strategy, respectively. This improvement increased as the number of candidate edges for each outer envelope calculation increased.

7. Conclusion

In this paper we have modified Ghosh's algorithm to calculate the NFP of a convex and a non-convex polygon so that the NFP of two non-convex polygons can easily and efficiently be calculated.

The time-consuming process of finding the outer envelope from the list of candidate edges, NFPList, has been improved by reducing the size of NFPList, and dividing the problem into smaller sub-problems based on the convexity of the original two polygons.

Solutions to industrial two-dimensional packing problems usually have time constraints in which they must adhere to. Most of these solutions involve the computationally expensive calculation of many NFPs. By increasing the speed in which these NFPs can be calculated, we open the possibility of more effective solution procedures to industrial packing problems.

References

[1] O'Rourke J. Computational geometry in C. Cambridge: Cambridge University Press; 1998.

[2] Cunninghame-Green R. Geometry, shoemaking and the milk tray problem. New Scientist 1989;1677:50–3.

[3] Lo Valvo E. A new simple and quick algorithm for the calculation of "no fit polygon" of generic polygons. Working paper.

[4] Mahadevan A. Optimisation in computer-aided pattern packing. Ph.D. thesis, North-Carolina State University; 1984.

[5] Kendall G. Applying meta-heuristic algorithms to the nesting problem utilising the no fit polygon. Ph.D. thesis, University of Nottingham; 2000.

[6] Ghosh PK. An algebra of polygons through the notion of negative shapes. CVGIP: Image Understanding 1991;54(1): 119–44.

[7] Bennell JA, Dowsland KA, Dowsland WB. The irregular cutting-stock problem—a new procedure for deriving the no-fit polygon. Computers and Operations Research 2001;28:271–87.

[8] Hershberger J. Finding the upper envelope of n line segments in $O(n \log n)$ time. Information Processing Letters 1989;33: 169–74.