Ann Oper Res
DOI 10.1007/s10479-013-1341-4

An effective heuristic for the two-dimensional irregular bin packing problem

Eunice López-Camacho · Gabriela Ochoa · Hugo Terashima-Marín · Edmund K. Burke

Abstract This paper proposes an adaptation, to the two-dimensional irregular bin packing problem of the Djang and Finch heuristic (DJD), originally designed for the one-dimensional bin packing problem. In the two-dimensional case, not only is it the case that the piece's size is important but its shape also has a significant influence. Therefore, DJD as a selection heuristic has to be paired with a placement heuristic to completely construct a solution to the underlying packing problem. A successful adaptation of the DJD requires a routine to reduce computational costs, which is also proposed and successfully tested in this paper. Results, on a wide variety of instance types with convex polygons, are found to be significantly better than those produced by more conventional selection heuristics.

Keywords 2D bin packing problem · Irregular packing · Heuristics · Djang and Finch heuristic

1 Introduction

The problem of finding an arrangement of pieces to cut from or pack inside larger objects is known as the cutting and packing problem, which is NP-hard. The two-dimensional (2D) bin packing problem (BPP) is a particular case of the basic problem. It consists of finding

E. López-Camacho (✉) · H. Terashima-Marín
Tecnológico de Monterrey, Av. E. Garza Sada 2501, Monterrey, NL, 64849, Mexico
e-mail: eunice.lopez@itesm.mx

H. Terashima-Marín
e-mail: terashima@itesm.mx

G. Ochoa · E.K. Burke
Computing Science and Mathematics, University of Stirling, Stirling, Scotland, UK

G. Ochoa
e-mail: gabriela.ochoa@cs.stir.ac.uk

E.K. Burke
e-mail: e.k.burke@stir.ac.uk

Published online: 28 February 2013

an arrangement of pieces inside identical objects such that the number of objects required to contain all pieces is minimum. The case of rectangular pieces is the most widely studied. However, the irregular case is seen in a number of industries where parts with irregular shapes are cut from rectangular materials. For example, in the shipbuilding industry, plate parts with free-form shapes for use in the inner frameworks of ships are cut from rectangular steel plates, and in the apparel industry, parts of clothes and shoes are cut from fabric or leather (Okano 2002). Other direct applications include the optimization of layouts within the wood, metal, plastics, carbon fiber and glass industries. In these industries, small improvements of the arrangement can result in a large saving of material (Hu-yao and Yuan-jun 2006).

This paper proposes a heuristic for selecting the next pieces to be placed when solving the 2D irregular BPP. The proposed heuristic is not only fast in execution, but it also produces excellent results when compared against more conventional selection heuristics. A simple but successful placement procedure is also presented, which produces outstanding results when coupled with the proposed selection heuristic. The paper proceeds as follows. The next section presents the problem description and a literature review. Section 3 describes the DJD heuristic, which is the base for our approach. Section 4 describes the implementation details of the heuristic. Finally, Sects. 5 and 6 give the experimental results and conclusions, respectively.

2 The 2D irregular bin packing problem

The cutting and packing problem is among the earliest problems in the literature of operational research. Wäscher et al. (2007) suggested a complete problem typology which is an extension of Dychoff (1990). In this paper, we consider the problem classified as the **2D irregular single bin size bin packing problem** in Wäscher, Haussner and Schumann's typology. Given a set $L = (a_1, a_2, \ldots, a_n)$ of pieces to cut or pack and an infinite set of identical rectangular larger elements (called *objects*), the problem consists of finding an arrangement of *pieces* inside the *objects* such that the number of objects required to contain all pieces is minimum. A feasible solution is an arrangement of pieces without overlaps and with no piece outside the object limits. A *problem instance* or *instance* $I = (L, x_0, y_0)$ consists of a list of elements L and object dimensions x_0 and y_0. The term 2D regular BPP is mainly used when all pieces are rectangular (although circles and other regular shapes could fall under this name too Wäscher et al. 2007); otherwise, the problem is called 2D irregular BPP. The problem is offline, and therefore the list of pieces to be packed is static and given in advance.

The **strip packing problem** is a popular variant of the 2D cutting and packing problem which has only one large rectangular object with fixed width, its length is variable and has to be minimized after placing all the small pieces (some approaches and reviews by Dowsland and Dowsland 1995; Hopper and Turton 2001; Burke et al. 2006). A variation of this typical strip packing problem consists on nesting irregular pieces in one large irregular object (examples by Lamousin and Dobson 1996; Lamousin and Waggenspack 1997; Whelan and Batchelor 1992; Ramesh 2001). The amount of research devoted to the strip packing problem has been larger when compared to the research about the 2D irregular BPP. We found that Okano (2002) studied a problem similar to our **2D irregular single bin size BPP** but using variable bin sizes, where the problem solution involves finding appropriate sizes of material objects (bins) among given standard sizes in order to reduce waste. As far as we know, there are no previous studies for, specifically, the *2D irregular single bin*

size bin packing problem. Therefore, all 2D irregular problem instances in the literature are intended for the strip packing problem; and there are not 2D irregular BPP instances available. Also, since the strip packing problem is similar to the 2D irregular BPP, many heuristic implementations may be similar; although, results for both kind of problems are not comparable. Nevertheless the lack of research in the 2D irregular BPP, there exist many practical applications where irregular pieces are cut from identical rectangular objects (Okano 2002).

For the 1D case, it is common to use the terms *items* and *bins*, regarding the small and large elements respectively; whereas, for the 2D case, a variety of terms has been used. The small elements have been named *pieces*, *shapes* or *items*; and the large elements have been called *objects*, *stock* or *sheets*. Here, we use the terms *items* and *bins* regarding the 1D case, and *pieces* and *objects* when referring to the 2D case.

For many real-world problems, an exhaustive search for solutions is not a practical proposition due to the large size of the solution search space. Hence, many heuristic approaches have been adopted. It is common that heuristic approaches for the bin packing problem present at least two phases: first, the selection of the next piece to be placed and the corresponding object to place it; and second, the actual placement of the selected piece in a position inside the object according to some given criteria. Some approaches consider a third phase as a local search mechanism.

The placement procedure for irregular pieces has attracted more attention than the exploration of the selection criteria. There are several techniques to generate potential placement positions for the next piece to be placed, and many of them are based on building the no-fit polygon. The no-fit polygon gives the set of non-overlapping placements for a given pair of polygons. Good examples of this type of procedure that work well for the strip packing problem are described by Hu-yao and Yuan-jun (2006), Gomes and Oliveira (2002), Dowsland et al. (2002), Burke and Kendall (1999b), Burke et al. (2007). While the no-fit polygon is a powerful geometric technique, there are several issues that limit its scalability for industrial applications. No-fit polygon techniques are notorious for the large quantity of degenerate cases that must be handled to make it completely robust (Burke et al. 2006). Whilst the generation of the no-fit polygon is academically challenging, it is a tool and not a solution (Burke et al. 2007). After generating the possible placement positions, it is necessary to have some criteria for choosing the best position. In 2D cutting and packing problems the most commonly used method for packing regular and irregular pieces involves the bottom-left class of heuristics. These methods involve simply placing the input list of pieces into the bottom-leftmost location on the packing sheet (Allen et al. 2011).

Regarding the selection criteria, most researchers have focused upon exploring different ways of finding good permutation of pieces. Okano (2002) obtains an ordering of pieces with respect to their areas and the similarities among them. Dowsland et al. (2002) use eight static orderings, which have the common strategy of trying to place the difficult-to-place pieces first. Dynamic selection permits all pieces to be available to be placed next Bennell and Oliveira (2009), for example, Bennell and Song (2010) use beam search. This approach searches the breadth first tree, and prunes the tree at each level according to two evaluation functions.

Several metaheuristic approaches have been applied to the strip packing problems, for example, tabu search (Bennell and Dowsland 2001), evolutionary computation (Bounsaythip and Maouche 1997) and ant algorithms (Burke and Kendall 1999a). Metaheuristic techniques are often very effective; however, there can be some reluctance to use them for money-critical problems. Practitioners in industry often favor the use of very simple and readily understandable methods even if they deliver relatively inferior results (Ross 2005). Among the main criticisms of stochastic-based problem solving techniques are: (1) the fact

that they involve some randomness and unpredictability, so that identical runs may deliver very different results; (2) there is little understanding about their average- and worst-case behavior (Ross and Marín-Blázquez 2005); (3) solutions quality greatly depend on a good parameter choice; and (4) the parameter tunning task requires time, knowledge and experience about the problem domain and properties which makes metaheuristics problem-specific solution methods that can be developed and deployed only by experts (Ross 2005; Bilgin et al. 2006).

3 The DJD heuristic

The proposed approach is based on the DJD heuristic, which is a selection heuristic designed for the 1D case. In its original version, as explained by Ross et al. (2002), the DJD heuristic puts items into a bin, taking items largest-first until that bin is at least one third full. It then tries to find one, or two, or three items that completely fill the bin. If there is no such combination it tries again, but looking instead for a combination that fills the bin to within 1 unit of its capacity. If that fails, it tries to find a combination that fills the bin to within 2 units of its capacity; and so on. This routine is to be performed as long as there are pieces to place. DJD is a single-pass constructive heuristic. A popular variation of DJD is called DJT (Djang and Finch, more tuples) which considers combinations of up to five items rather than three items.

Problems known to be hard have certain characteristics. In bin packing, especially in the 1D case, instances with many small items are not hard, since the small items can be employed as *sand* to fill up the remaining space when large items were packed. Difficulty arises when most of the pieces have an area that is a significant fraction of the total object area, for example at least 20 % of the object area, so that the challenge is to find the subset among a large number of pieces to be placed in a given object (Ross 2005). We hypothesize that the DJD heuristic is intended for these kind of hard instances; because, exactly as stated, it works well in many problems known to be hard, but it fails in other types of problem. For example, consider a very easy problem in which the bins have capacity 1000 and there are 10,000 items each of weight 1. Packing these items will need only 10 bins. However, DJD will first fill a bin until it contains 334 items (just over one-third) and then add just three more items into the bin, so the bin will contain 337 items. Thus, 30 bins will be needed ($337 \times 29 = 9773$) (Ross 2005), a solution far from optimality. The obvious remedy to this situation is to keep trying to place item combinations until no single item can be placed. Although, in the case when items are so small compared with the bin free space, there is no advantage in trying every combination of items, since no combination would result in zero waste in the first attempts.

Ross et al. (2002, 2003), Marín-Blázquez and Schulenburg (2006) and Pillay (2012) have implemented the DJD and its variation DJT for the 1D BPP as a part of procedures (hyper-heuristics) that learn to combine heuristics for solving the underlying problem. In these approaches, the idea is to automatically apply different heuristics to different states of the construction process. In this scenario, DJD and DJT were reported as the best heuristics considered. Also, the DJD heuristic was adapted to solve the problem of scheduling transportation events for minimizing the number of vehicles used, while satisfying the customer demand (Terashima-Marín et al. 2005b). Kos and Duhovnik (2000) describe the same heuristic but named it as *Exact Fit* in an approach for rod cutting optimization with remnants utilization. Sim et al. (2012) present an evolutionary algorithm for evolving classifiers that are used for predicting the best heuristic to solve each of a set of unseen 1D BPP instances.

They present the *Adaptive DJD* which packs items into a bin in descending order until the free space in the bin is less than or equal to three times the average size of the items remaining to be packed. This recent version of the DJD heuristic obtains poorer results than the DJD and DJT when the metric is the percentage of instances solved using the optimum number of bins. When the metric is the fitness given in Eq. (2) or the percentage of extra bins required over the optimal, *Adaptive DJD* is better than DJD and worse than DJT when averaging 1320 instances.

Where pieces to be placed are rectangles, DJD and DJT have been adapted and implemented as a selection heuristic (Terashima-Marín et al. 2005a, 2006). For the 2D irregular BPP, where pieces to be placed are convex polygons, DJD has been implemented as a member of a heuristic repository in a hyper-heuristic approach (Terashima-Marín et al. 2010). To our knowledge, DJT has not been implemented for the 2D irregular BPP. In the previous studies, the performance of DJD was not analyzed, nor reported separately and the authors did not report a routine for improving the running times. This is essential in the 2D case, because simply comparing the area of a 1, 2 or 3-piece combination against the free area of the object does not imply that the pieces can actually be placed. Indeed, several groups of pieces may need to be tried before a given combination of pieces can be placed. Moreover, the same pieces may be tested several times in different combinations before the algorithm is successful in placing a 1, 2 or 3-piece group. Besides, to determine whether or not a piece can be placed in a given object is the most time-consuming task when solving a 2D bin packing problem. The placement task requires even more running time when pieces are irregular. In this paper, DJD is adapted to and thoroughly analyzed when solving a variety of instances of the 2D irregular BBP. Moreover, a routine for reducing redundant computation is proposed and successfully tested.

4 The proposed DJD heuristic for the 2D irregular BPP

The DJD algorithm for the 2D case works as a selection heuristic, but it alone does not solve the problem completely. DJD has to be paired with a placement heuristic which will determine the exact position of each piece inside an object.

For the 2D case (regular and irregular), the general process of the DJD heuristic is outlined in the pseudo code of Algorithm 1. In the 2D adaptation of the heuristic, DJD puts pieces into an object, taking them by decreasing area, until at least one-third of its area is covered. It then tries to find one, or two, or three pieces that completely fill the object. If there is no such combination it tries again, but looks instead for a combination that fills the bin to within w of its capacity. If that fails, it tries to find such a combination that fills the object to within $2w$ of its capacity; and so on. In the 1D case, the waste incremental suggested is 1 unit. Depending on the order of magnitude of the object and pieces sizes, in 2D it would not be feasible to manage a 1-unit incremental. Therefore, the incremental should be selected according to the total object area. For the 2D adaptation of the heuristic, the processes of reviewing groups of one, two or three pieces are modified to optimize running time. These processes mentioned in Algorithm 1, are described in Algorithms 2, 3 and 4 respectively.

Pieces are placed sequentially when trying groups of 2 or 3 pieces. Only when the first piece is placed successfully, a next one is tried, and so on. If all possible second pieces fail to fit, the first piece is unplaced and then we try a different group.

Every time a combination of 1, 2 and 3 pieces is placed, the checking process starts all over again in the same object (resetting the allowed waste to 0). Only when no more pieces

Algorithm 1 The DJD heuristic

Input: A list of pieces sorted by decreasing area; width and height of the rectangular objects.

Output: All pieces placed in objects.

1: $waste = 0$; w [increment of allowed waste, $w = 1$ in the original version of the heuristic]

2: **while** there are pieces to place **do**
3: Fill the object until at least one-third of its area is covered
4: Register in memory every piece that does not fit

5: Try pieces one by one [*see Algorithm 2*]
6: **if** a piece could be placed leaving a free area up to *waste* **then**
7: reset $waste = 0$ and start again trying pieces one by one

8: Try groups of 2 pieces [*see Algorithm 3*]
9: **if** a pair of pieces could be placed leaving a free area up to *waste* **then**
10: reset $waste = 0$ and start again trying pieces one by one

11: Try groups of 3 pieces [*see Algorithm 4*]
12: **if** a group of 3 pieces could be placed leaving a free area up to *waste* **then**
13: reset $waste = 0$ and start again trying pieces one by one

14: **if** no piece could be placed trying all possible 1, 2 or 3-piece groups AND $waste <$ object free area **then**
15: $waste = waste + w$
16: **else**
17: open a new object

Algorithm 2 Trying pieces one by one

1: **for** all pieces in decreasing size order **do**
2: **if** object free area − area of piece > *waste* **then**
3: break
4: **if** area of piece > object free area OR piece has failed to fit **then**
5: continue [with the next piece]
6: Try to place the piece in the object
7: **if** the piece could be placed **then**
8: return
9: **else**
10: register in memory that the piece does not fit

can be placed in an object, a new object is opened. The DJD heuristic works in one open object at a time, there is no need to review previous open objects. Order is important in 2D packing; groups with the same pieces are revised considering all possible orderings. When executing a placement heuristic, a piece combination that cannot be placed in a particular order could be placed in another piece order.

In order to reduce the computational effort, a record of what pieces have been tried so far as a first member of a 1, 2 or 3-pieces group is kept for each object, so the algorithm does

Algorithm 3 Trying groups of 2 pieces

```
1:  for all pieces in decreasing size order do
2:    if object free area − piece's area − largest piece's area > waste then
3:      break
4:    if the piece has failed to fit OR piece's area + smallest piece's area > free space then
5:      continue [with the next piece in the list]
6:    Try to place the piece in the object
7:    if piece could not be placed then
8:      register it in memory
9:    else {select a second piece when a first piece succeeded to be placed}
10:     for all remaining pieces do
11:       if object free area − area of the 2 pieces > waste then
12:         break
13:       if the piece or the pair of pieces has failed to fit OR 2 pieces' area > free space
          then
14:         continue [with the next piece]
15:       Try to place the second piece in the object
16:       if the piece could be placed then
17:         return
18:       else
19:         unplace first piece AND register that the pair of pieces does not fit
```

not try again the same piece in a different group. Additionally, a record is kept of all ordered pairs of pieces that failed to be placed in a particular object either as a 2-piece group or as the first 2 pieces of a 3-piece group. These pairs of pieces are, therefore, not tried again in the same order in the same object. Finally, all ordered 3-piece groups that fail to fit in an object are recorded as well. These records help to reduce an important amount of redundant computation.

As it can be seen in Algorithms 2, 3 and 4, when DJD checks one, two or three-piece groups, first it compares the pieces' areas against the maximum waste allowed and against the available object area. Only then, does DJD try to place them. For the 2D BPP, checking if a piece could or could not be placed, is computationally expensive. Pieces should be in descending order when the DJD heuristic starts, allowing the *For* cycles to *break* at some point when reviewing pieces; thus, reducing comparisons (see Algorithms 2, 3 and 4).

According to the placement procedures considered, when a piece cannot be placed in an object at a given time, there is a slight possibility that it can actually be placed later when one or more pieces had been placed. Considering this possibility in the implementation would increase the algorithm running time. If time is not a constraint, the option would be to keep a record of pieces that fail to fit just until one piece or group is placed, and then clean up the records.

5 Algorithms for geometric computation

The algorithms presented in this section are the building blocks for implementing the placement heuristics (Sect. 6.2). These algorithms are suited for dealing with convex and non-convex shapes (even though our testbed instances have only convex pieces). Each piece is

Algorithm 4 Trying groups of 3 pieces

```
1:  for all pieces in decreasing size order do
2:    if object free area − piece's area − area of the 2 largest pieces > waste then
3:      break
4:    if the piece has failed to fit OR piece's area + 2 smallest pieces' area > free space
      then
5:      continue [with the next piece]
6:    Try to place the piece in the object
7:    if piece could not be placed then
8:      register it in memory
9:    else {select a second piece when a first piece succeeded to be placed}
10:     for all remaining pieces do
11:       if object free area − area of the 2 pieces − area of largest piece > waste then
12:         break
13:       if the piece or the pair of pieces has failed to fit OR
          area of the 2 pieces + area of smallest piece > object free area then
14:         continue [with the next piece]
15:       Try to place the second piece in the object
16:       if the piece could not be placed then
17:         unplace first piece AND register that the pair of pieces does not fit
18:       else {select a third piece when two pieces have been placed}
19:         for all remaining piece do
20:           if object free area − area of the 3 pieces > waste then
21:             break
22:           if any piece, or pair or 3-piece group of pieces have failed to fit OR
              area of the 3 pieces > object free area then
23:             continue [with the next piece]
24:           Try to place the third piece in the object
25:           if the piece could be placed then
26:             return
27:           else
28:             unplace first 2 pieces AND register that the 3-piece group does not fit
```

represented by its vertices coordinates ordered counterclockwise. Algorithms are mainly devoted to the task of detecting overlapping (Algorithms 5, 6 and 7), to compute the distance that one piece can slide without crossing to another (Algorithms 9 and 10) and to compute adjacency (Algorithm 11).

Most of the algorithms are based on basic geometrical concepts, but particular cases and exceptions deserve special care. The easiest-to-solve cases should be reviewed first in order to avoid unnecessary computations. For example, when checking whether a point is inside a shape, a quick computation to determine is the point is above (or below) the top or (the bottom) of the piece will discard many cases. Trivial cases like this one are a frequent scenario when applying the placement heuristics with our set of instances.

Two considerations to take into account are:

1. Our function that reviews if two segments have an intersection returns **false** if they belong to the same line, even if one segment touch the other by one of their ends or if they overlap. In other words, our definition of intersection of segments refers to segments

Algorithm 5 Decide if two pieces intersects each other

Input: A list of coordinates of two pieces P_1 and P_2.
Output: A boolean value indicating whether the two pieces intersects each other.

if lowest end of P_1 is above upper end of P_2 OR lowest end of P_2 is above upper end of P_1 **then**
  return false
if leftmost end of P_1 is right of the rightmost end of P_2 OR leftmost end of P_2 is right of the rightmost end of P_1 **then**
  return false

for all edges e_1 of P_1 **do**
  for all edges e_2 of P_2 **do**
    if Intersects(e_1, e_2) **then**
      return true
return false

Algorithm 6 Decide if a point is inside a shape

Input: A list of coordinates of a piece and a point (x, y).
Output: A boolean value indicating whether the point is or not inside the piece.

if $x \leq$ the piece lowest part OR $x \geq$ the piece upper part **then**
  return false
if $y \leq$ the piece leftmost part OR $y \geq$ the piece rightmost part **then**
  return false
for all vertices of the piece **do**
  if the point (x, y) is equal to the vertex **then**
    return false
for all sides of the piece **do**
  if the point (x, y) is along the side **then**
    return false

Create the point (M, y), where M is a very large number.
for all sides of the piece **do**
  if the side of the piece intersects the segment (x, y) to (M, y) **then**
    counter ++
for all vertices i of the piece **do**
  if the vertex belong to the segment (x, y) to (M, y) **then**
    $D_1 \leftarrow$ *Dfunction*(segment, vertex $i - 1$) [*see Eq. (1)*]
    $D_2 \leftarrow$ *Dfunction*(segment, vertex $i + 1$)
    if D_1 and D_2 have different signs **then**
      counter ++

if counter is odd **then**
  return true
else
  return false

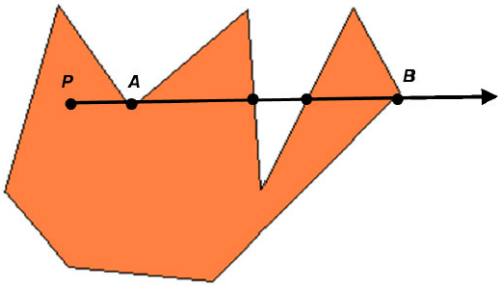

Fig. 1 Ray from point P to the right actually touches 4 times the shape boundaries. The ray crosses the shape at vertex B. In contrast, the ray touches vertex A only tangentially and does not cross the shape at this point. Therefore, the count for crosses is 3. Since 3 is an odd number, we conclude that P is inside the shape

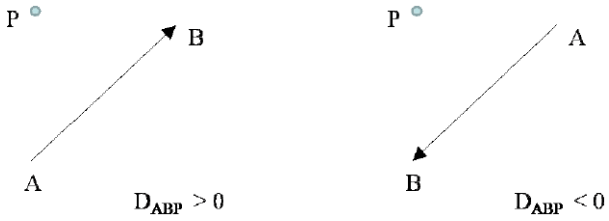

Fig. 2 Interpretation of the D-function

that *crosses* but are not coincident. Note that this definition implies that reviewing for intersection of two segments that are exactly the same, the function will return **false**.

2. Our function that reviews if a point is inside a segment returns **true** if the point is one of the ends of the segment. When the sum of the distances from the point to the two ends of the segment is equal to the segment length, then we consider that the point belongs to the segment.

To know whether two pieces intersects each other, a routine that checks intersection for each pair of sides from both pieces was implemented (Algorithm 5). Initially, a revision is done to confirm that the orthogonal rectangles that circumscribe both pieces intersect. This is used to discard the easiest non-intersection cases. This test does not work if one piece is completely inside the other, in which case no edges intersect but the pieces do intersect. In consequence, this algorithm is always followed by Algorithm 7 that reviews if one piece is completely inside another.

Algorithm 6 determines whether a point is inside a shape. If the point is along an edge of the piece or one of its vertices, then the algorithm will return false. The basic idea is to trace a ray from the point to any fixed direction. If the ray cuts the shape an odd number of times, then the point is inside the shape; otherwise it is outside. If the ray touches a vertex of the shape; it is important to determine if the ray touches the shape tangentially or if it actually crosses the shape (see Fig. 1). This is done employing the D-function (Eq. (1)). For line intersection, the D-function gives the relative position of a point P with respect to an oriented edge AB (see Fig. 2). The D-function is defined as follows:

$$D_{ABP} = (X_A - X_B)(Y_A - Y_P) - (Y_A - Y_B)(X_A - X_P) \tag{1}$$

Depending if D_{ABP} is negative or positive, the point P is on the left or the right side of the edge AB. The definition of left and right is as follows: if an observer would stand at point A looking in the direction of B, point P would be at the observer's left or right. If $D_{ABP} = 0$, the point P is on the supporting line of edge AB (Bennell and Oliveira 2008).

Algorithm 7 is used to determine if a piece is completely inside another piece. Initially, a revision is done to confirm that the orthogonal rectangles that circumscribe both pieces

Algorithm 7 Decide if a shape is completely inside another shape

Input: The two pieces P_1 and P_2.
Output: A boolean value indicating whether one of the pieces is inside the other.

Part 1
if lowest end of P_1 is above upper end of P_2 OR lowest end of P_2 is above upper end of P_1 **then**
  return false
if leftmost end of P_1 is right of the rightmost end of P_2 OR leftmost end of P_2 is right of the rightmost end of P_1 **then**
  return false
if the 2 pieces intersect each other [*see Algorithm 5*] **then**
  return false

Part 2 [*At this point we only have pieces that do not intersect each other*]
$y_{max} \leftarrow$ max(maximum P_1 y-coordinate, maximum P_2 y-coordinate)
$y_{min} \leftarrow$ min(minimum P_1 y-coordinate, minimum P_2 y-coordinate)
$x_{max} \leftarrow$ max(maximum P_1 x-coordinate, maximum P_2 x-coordinate)
$x_{min} \leftarrow$ min(minimum P_1 x-coordinate, minimum P_2 x-coordinate)
if $(y_{max} - y_{min})(x_{max} - x_{min}) <$(area of P_1+ area of P_2) **then**
  return true

Part 3
$\bar{y}_1 \leftarrow$ average(maximum P_1 y-coordinate, minimum P_1 y-coordinate)
$\bar{x}_1 \leftarrow$ average(maximum P_1 x-coordinate, minimum P_1 x-coordinate)
$\bar{y}_2 \leftarrow$ average(maximum P_2 y-coordinate, minimum P_2 y-coordinate)
$\bar{x}_2 \leftarrow$ average(maximum P_2 x-coordinate, minimum P_2 x-coordinate)
if point $(\bar{x}_1, \bar{y}_1)$ is inside P_1 and P_2 **or** point $(\bar{x}_2, \bar{y}_2)$ is inside P_1 and P_2 **then**
  return true

Part 4
for all vertices **and** edge midpoints **and** points near each vertex of P_1 **do**
  if inside P_2 **then**
    return true
for all vertices **and** edge midpoints **and** points near each vertex of P_2 **do**
  if inside P_1 **then**
    return true

Part 5
if P_1 is equal to P_2 **and** in the same position **then**
  return true
else
  return false

intersect and that the actual pieces do not intersect (part 1). If both pieces do not intersect, we find the orthogonal rectangle that circumscribe both pieces at the same time. If the area of this rectangle is less than the sum of areas of both pieces, it means unequivocally that one piece is inside the another (part 2). If the point in the middle of piece 1 is inside piece 1, then

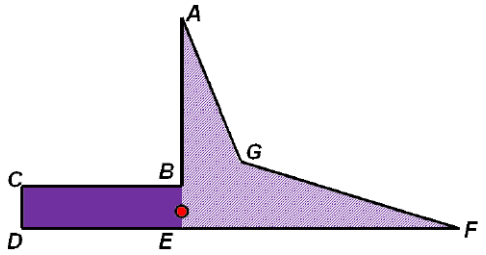

Fig. 3 *Piece AEFG* is inside *piece ABCDEFG*. In this case, checking if all vertices and edges midpoints of *AEFG* are inside *ABCDEFG* will return **false**. Only when a point very close to vertex *E* is found inside *ABCDEFG*, the algorithm returns **true** to the question about if one of the pieces is inside the other. In this case, reviewing intersection of these two pieces with Algorithm 5 will return **false** because none of the sides crosses another (although they coincide)

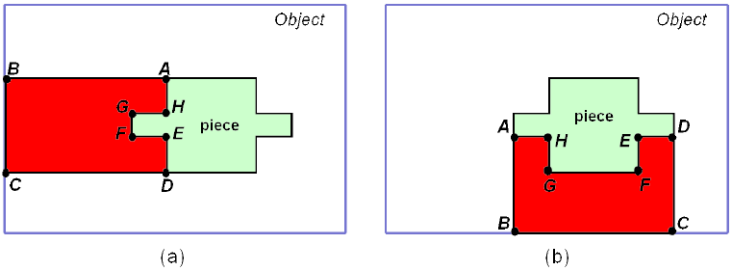

Fig. 4 Piece *ABCDEFGH* contains all area to the (**a**) left and (**b**) below a given piece

we check if this point is inside piece 2. The same is checked for the middle point of piece 2 (part 3). If this is not the case, then, all vertices and edge midpoints from both pieces are checked to know if they are inside the other piece. Checking vertices and edges midpoints is not an infallible test with non-convex shapes. It is possible to find a case where all vertices edges midpoints of the inside shape are all along the contour of the larger piece. See for example Fig. 3. Therefore, two points close to each vertex (one for each of the edges) are also tested (part 4). Finally, it is convenient to check whether the two pieces are not equal and in the same position (part 5).

Algorithm 8 builds a piece that holds all the area in the object that is left of a given piece (see Fig. 4a). A similar procedure is done to build a piece containing all the area below a given piece (see Fig. 4b). Algorithm 9 computes the distance by which a point can reach horizontally a segment. An analogous procedure finds a vertical distance from a point to a given segment.

Algorithms 8 and 9 are needed when executing Algorithm 10 which computes the distance that a given piece can be moved to the left avoiding collision against other pieces in the object and without exceeding the object limits. A similar procedure was implemented in this investigation to find how much a given piece can be moved down. The implementation of this algorithm is basic for bottom-left moves that take place in all placement heuristics (Sect. 6.2).

Algorithm 11 returns the distance in which two segments coincide. This algorithm constitutes the basis for implementing the heuristic called *Constructive Approach with Maximum Adjacency* (Sect. 6.2).

Algorithm 8 Builds a piece containing all the area at the left of a given piece

Input: A piece P.
Output: A piece whose area is the same that the left area of P.

Find (x_1, y_1), the vertex at the top of P which is leftmost [*point A in Fig. 4a*].
Find (x_2, y_2), the vertex at the bottom of P which is leftmost [*point D in Fig. 4a*].
return the piece comprised by the following vertices:
    (x_1, y_1)
    $(0, y_1)$
    $(0, y_2)$
    (x_2, y_2) and
    all vertices in P between (x_2, y_2) and (x_1, y_1)

Algorithm 9 Computes the horizontal distance from a point to a given segment. Distance is zero if the point is along the segment. Distance is positive if the point is in the right of the segment. Otherwise it is negative

Input: A point (x, y) and a segment defined by points (x_1, y_1) and (x_2, y_2).
Output: The horizontal distance from (x, y) to the segment defined by (x_1, y_1) and (x_2, y_2).

if $(y < y_1$ **and** $y < y_2)$ **or** $(y > y_1$ **and** $y > y_2)$ **then**
    return 'The point does not reach horizontally the segment'
if $(y = y_1$ **and** $y = y_2)$ **and** $(x > x_1$ **and** $x > x_2)$ **then**
    return $\min(x - x_1, x - x_2)$
if $(y = y_1$ **and** $y = y_2)$ **and** $(x < x_1$ **and** $x < x_2)$ **then**
    return $-\min(x_1 - x, x_2 - x)$
if $(y = y_1$ **and** $y = y_2)$ **then**
    return 0
else
    return $x - x_1 + (x_1 - x_2)(y_1 - y)/(y_1 - y_2)$

Algorithm 10 Computes the distance that a given piece can be moved to the left without overlapping other pieces and without exceeding the object limits

Input: A piece P and the other pieces that are inside the same object.
Output: The distance that P can be moved to the left.

Build piece P' whose area is the same that area at the left of P [*Algorithm 8 and Fig. 4a*].
Find the set S containing all pieces in the object that intersect or are inside P' but do not intersect nor are inside P.
$m \leftarrow$ minimum x-coordinate of P
if S is empty **then**
    return m
for all vertices i of P **do**
    for all edges j of **all** pieces in the object **do**
        if vertex i reaches edge j projecting to the left **then**
            $d \leftarrow$ distance from vertex i to edge j [*Algorithm 9*]
            if $d < m$ **then**
                $m \leftarrow d$
return m

Algorithm 11 Measures the distance in which two segments coincide

Input: The two finite segments S_1 and S_2.
Output: The distance in which S_1 and S_2 coincide.

if lowest end of S_1 is above upper end of S_2 OR lowest end of S_2 is above upper end of S_1 **then**
  return 0
if leftmost end of S_1 is right of the rightmost end of S_2 OR leftmost end of S_2 is right of the rightmost end of S_1 **then**
  return 0
if slope of $S_1 \neq$ slope of S_2 **then**
  return 0
if y-intercept of $S_1 \neq$ y-intercept of S_2 **then**
  return 0 [segments are parallel]
if S_1 and S_2 are both horizontal **then**
  $p_1 \leftarrow$ rightmost point out of the leftmost ends of S_1 and S_2.
  $p_2 \leftarrow$ leftmost point out of the rightmost ends of S_1 and S_2.
  return distance from p_1 to p_2
else
  $p_1 \leftarrow$ upper point out of the lowest ends of S_1 and S_2.
  $p_2 \leftarrow$ lowest point out of the upper ends of S_1 and S_2.
  return distance from p_1 to p_2

6 Experiments and results

This section describes how the DJD is tested against seven other selection heuristics combined with four different placement heuristics. Characteristics of the problem instances are given as well as a way to measure the quality of the solution.

6.1 The other selection heuristics

The selection heuristics used for comparison against DJD are:

1. **First Fit (FF)**. Considers the open objects in the order they were initially opened, and places the next piece in the first object where it fits. If the piece does not fit in any open object, a new object is opened to place it. Pieces are processed in the order they are presented in the instance being solved. This heuristic considers all partially filled objects as candidates for the next piece to be packed.
2. **First Fit Decreasing (FFD)**. Sorts pieces by decreasing area, and places the pieces according to FF.
3. **First Fit Increasing (FFI)**. Sorts pieces by increasing area, and places the pieces according to FF.
4. **Filler**. Sorts the pieces in order of decreasing area and packs as many pieces as possible within the open object. When no single piece can be placed in the open object, a new object is opened to continue packing the pieces from largest to smallest. This heuristic works with one open object at a time.
5. **Best Fit (BF)**. Considers the open objects in the increasing order of free area, and places the next piece in the first object where it fits, that is, in the object that leaves minimum waste. If the piece does not fit in any open object, a new object is opened to place it.

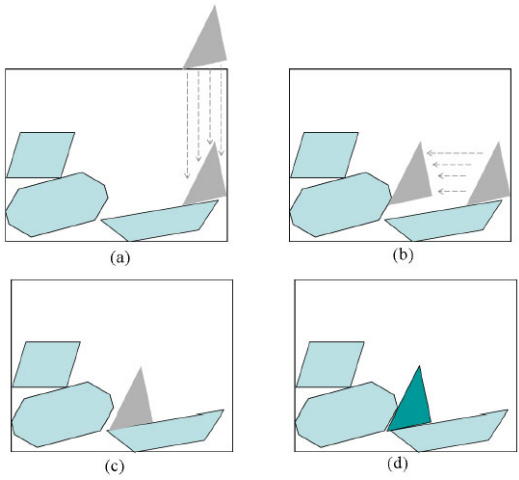

Fig. 5 Bottom-left heuristic for irregular pieces (Terashima-Marín et al. 2010)

Pieces are processed in the order they are presented in the instance being solved. This heuristic considers all partially filled objects as candidates for the next piece to pack.

6. **Best Fit Decreasing (BFD)**. Same as the previous heuristic, but sorts the pieces by decreasing area.
7. **Worst Fit (WF)**. Same as heuristic BF but places the piece in the open object where it worst fits (that is, in the object with the largest available room).

Notice that the first part of DJD, when an object is filled until one-third, corresponds to the FFD heuristic. These seven heuristics are all the single-pass selection heuristics that we could get in the literature for the offline BPP. These selection heuristics are mainly associated with rectangles in the literature (Hopper and Turton 2002; Ross et al. 2002; Terashima-Marín et al. 2006). To implement them with irregular shapes we need to employ adequate functions for shape movement and feasibility check (Sect. 5).

6.2 The placement heuristics

Once a piece and an object are selected, the placement heuristic states the way in which the piece is located inside the object. Two different placement heuristics could arrive to different conclusions as to whether a piece can or cannot be placed inside the object, and about the piece's final coordinates. We consider four placement heuristics that work in combination with the selection heuristics:

1. **Bottom-Left (BL)**. This is the best known heuristic of its type. The piece starts at the top right hand corner of the object and it slides down and left with a sequence of movements until no other movement is possible (see Fig. 5). If the final position does not overlap the object boundaries, the piece is placed in that position. The heuristic does not allow a piece to skip around another placed piece. The performance of this heuristic greatly depends on the initial ordering of the pieces (Dowsland et al. 1998, 2002). Its advantage lies in its speed and simplicity.
2. **Constructive Approach (CA)**. This heuristic is based on the work presented by (Hifi and M'Hallah 2003). The heuristic starts by placing the first piece at the bottom-left of the object. Then, for each placed piece five alternative positions are computed and stored in a list: $(\bar{x}, 0)$, $(0, \bar{y})$, $(\underline{x}, \bar{y})$, $(\bar{x}, \bar{y})$ and $(\bar{x}, \underline{y})$, where $\bar{x}$, $\underline{x}$, $\bar{y}$, and $\underline{y}$ are the maximum and

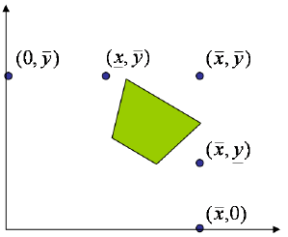

Fig. 6 Positions to be considered in the Constructive Approach (Terashima-Marín et al. 2010)

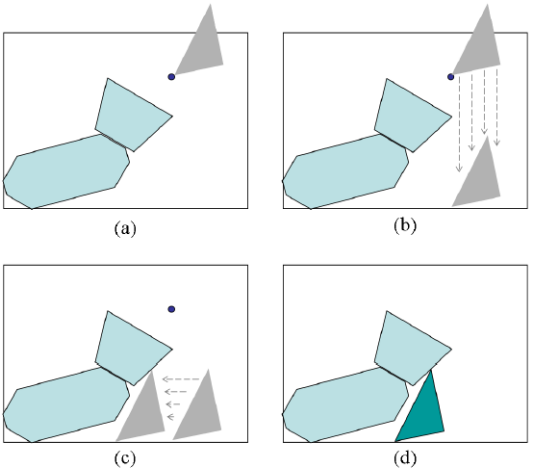

Fig. 7 Constructive Approach Heuristic (Terashima-Marín et al. 2010)

minimum coordinates in x and y (see Fig. 6). Given that some positions might coincide, each position appears only once in the list. In our implementation, the four corners of the object (the large sheet of stock material) were also added as candidate positions. From each position in the list, the next piece slides vertically and horizontally following down and left movements as shown in Fig. 7. Positions with overlapping pieces, or exceeding the object dimensions, are discarded. Among the remaining positions, the most bottom-left position is chosen. Using the corners as a departure point to slide the piece bottom and left, allows the method to reach certain gaps between pieces, which would not be reachable if only the five initial positions were considered.

3. **Constructive Approach (Minimum Area) (CAA)**. This is a modification of the previous heuristic. The variation consists of selecting the best position from the list based on which one yields the bounding rectangle with minimum area, containing all pieces, and that fits in the bottom left hand corner of the object. The bounding rectangle area is computed with the product of the maximum horizontal coordinate and the maximum vertical coordinate of all pieces already placed plus the new piece to be located in the proposed position. Figure 8 shows the rectangle with minimum area for two different positions for a single piece. This criterion was chosen based on the idea of selecting a point with which all pieces, not only the last piece, are deepest (bottom and left).
4. **Constructive Approach (Maximum Adjacency) (CAD)**. This heuristic is partially based on the approach suggested by Uday et al. (2001). However, when the first piece is to be placed, our implementation considers only the four corners of the object. For the subsequent pieces, the *possible points* are the same as those in the constructive approach (CA, listed as our second placement heuristic), described above. Each position in the

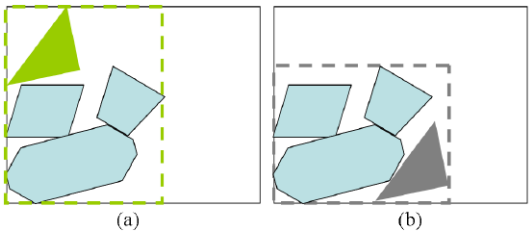

Fig. 8 Candidate rectangles when locating a piece (Terashima-Marín et al. 2010)

list is evaluated twice: the piece starts in the initial position and its adjacency (i.e., the common boundary between its perimeter and the placed pieces and the object edges) is computed. Then, the piece is slid down and left and the adjacency is computed again. The position with the largest adjacency is selected as the position of the new piece.

Placement is the most time-consuming procedure in building a solution because the geometric algorithms from Sect. 5 have to be performed many times in every attempt. For example, consider that we have an object with some placed pieces and we want to place a new piece employing the BL heuristic. We start placing the new piece at the top right hand corner of the object and computing the distance that the piece can slide downwards. This distance is the minimum of all distances among all new piece vertices and all the edges of the placed pieces. The sliding operations downwards and leftwards may be performed several times until the new piece reaches its definite position and then a validation is performed to check if the final position does not exceed the object limits. Otherwise we say that the heuristic fails to place the new piece in the given object. The CA, CAA and CAD heuristics involve finding several final positions and then selecting the best valid position according to the given criteria for each heuristic. Several attempts to place a piece may be executed before one piece succeeds. A new object is opened only when all the remaining pieces fail to fit into the current object.

For the last listed three placement heuristics: CA, CAA and CAD, the algorithm rotates each piece by multiples of 90 degrees and chooses the rotation that is better according to each heuristic criterion. For the first heuristic, BL, no rotation is considered, since BL does not choose among several possible positions as the other three placement heuristics do. Our empirical study explored all combinations of selection and placement heuristics with each of the available instances.

6.3 Description of instances

The benchmark instances in our study were produced by the generator proposed by Terashima-Marín et al. (2010). The generated pieces are convex irregular polygons with a number of sides between 3 and 8. A total of 540 instances were generated within 18 different types. For each generated instance, the order of all the pieces is randomized before writing the list of pieces. Their characteristics are listed in Tables 1 and 2. The optimum solution in all instances, except type G, is achieved only when all objects are filled up to 100 %. Objects for all instances are squares.

The instance set contains a wide variety of feature values (see Table 2). For example, there are instances whose pieces have an average size of $1/30$ of the object, while other instances have huge pieces (averaging $1/3$ of the object size). Average piece rectangularity goes from 0.35 to 1 along the 540 instances. Rectangularity is a quantity that represents the proportion between the area of a piece and the area of a horizontal rectangle containing it. This measure depends on the original orientation of the piece which is given by the particular

Table 1 Description of problem instances

Type	Objects side	Pieces	Num. of instances	Optimal (num. of objects)
A	1000	30	30	3
B	1000	30	30	10
C	1000	36	30	6
D	1000	60	30	3
E	1000	60	30	3
F	1000	30	30	2
G	1000	36	30	≤ 15
H	1000	36	30	12
I	1000	60	30	3
J	1000	60	30	4
K	1000	54	30	6
L	1000	30	30	3
M	1000	40	30	5
N	1000	60	30	2
O	1000	28	30	7
P	1000	56	30	8
Q	1000	60	30	15
R	1000	54	30	9
Total			540	

instance to solve. The lower the rectangularity, the more irregular the pieces are. As one can see in Table 2, instances of type I have a rectangularity value of 1 which means that all of these instances are rectangular.

6.4 Measure of performance

The quality of a solution, produced by any pair of selection and placement heuristics for a given instance, is based on the percentage of usage for each object, given by:

$$f = \frac{\sum_{i=1}^{N_o} U_i^2}{N_o} \tag{2}$$

where N_o is the total number of objects used and U_i is the fractional utilization for each object i. This measure of fitness rewards objects that are filled completely or nearly so, and avoids the problem of too many ties among different heuristics that occur when performance is measured by the number of objects used.

6.5 Results

The value of the waste incremental w is an important choice in the DJD heuristic. As observed experimentally in our instance set, if the waste incremental is set to $w = 1$, many 1-unit increments occur during the solution construction at a high computational cost without placing any single piece. We empirically found that a waste incremental of one-twentieth of the total object area is a good balance between fast and good solutions. Increments lower

Table 2 Characteristics of problem instances

	Average piece area	Piece area standard deviation	Average rectan-gularity	Percentage of right angles	Percentage of vertical/horizontal sides
Minimum	0.033	0.014	0.35	11	34
Total average	0.154	0.100	0.68	42	65
Maximum	0.354	0.280	1	100	100
Average of instances per type					
A	0.100	0.069	0.70	42	68
B	0.333	0.162	0.87	67	84
C	0.167	0.124	0.68	36	63
D	0.050	0.036	0.57	23	51
E	0.050	0.035	0.41	12	38
F	0.067	0.050	0.59	29	57
G	0.332	0.156	0.87	67	83
H	0.333	0.158	0.86	67	83
I	0.053	0.017	1	100	100
J	0.067	0.034	0.83	68	83
K	0.154	0.150	0.63	34	60
L	0.100	0.075	0.51	23	50
M	0.125	0.102	0.55	28	55
N	0.033	0.024	0.62	32	60
O	0.250	0.223	0.57	27	58
P	0.143	0.173	0.49	18	43
Q	0.250	0.053	0.89	51	76
R	0.167	0.153	0.63	36	62

than $w = 1/20$ of the object total size have a high computational cost, without a significant improvement in fitness, while increments higher than $w = 1/20$ of the object size lead to inferior results.

We explored the phenomenon of a piece that cannot fit into an object and it later fits (when there are one or more pieces in the object). This is rare for placement heuristics CA, CAA and CAD. Therefore, in this case, trying to fit pieces after they have failed to be placed, increases the running time. Results may be slightly better (or worse) in some cases, but generally speaking, the small improvement does not pay the huge excess of processing time (although this would depend on the particular application). For the BL placement heuristic this phenomenon is less rare. Hence, when using BL in combination with any selection heuristics, a record of pieces that fail to be placed is kept until a piece is successfully placed. After that, the records are cleaned.

We explored four different initial levels of fullness before trying to place combinations of pieces within an allowed waste, namely, $1/4$, $1/3$, $1/2$ and $2/3$. DJD heuristics with these levels are referred to as $\text{DJD}_{1/4}$, $\text{DJD}_{1/3}$, $\text{DJD}_{1/2}$ and $\text{DJD}_{2/3}$, respectively. Along with the 7 selection heuristics described above, we have 11 different selection heuristics overall.

All instances were solved with all heuristics (11 selection heuristics $\times$ 4 placement heuristics = 44 ways to solve a given instance). Figure 9 shows the solution of an instance type C with the selection heuristic $\text{DJD}_{1/3}$ and the placement heuristic CAD.

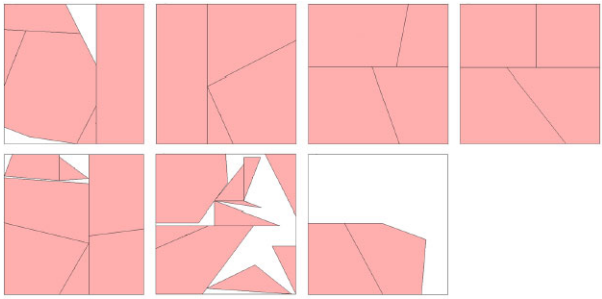

Fig. 9 $DJD_{1/3}$ in combination with CAD heuristic solves an instance of type C. This solution gets one more object than the optimal solution in which there would be zero waste. In this solution, fitness is 0.776 measured with Eq. (2)

Table 3 Average fitness for all the combinations of selection and placement heuristics over the 540 instances

	Selection heuristics											
Placement heuristics	FF	FFD	FFI	Filler	BF	BFD	WF	DJD 1/4	DJD 1/3	DJD 1/2	DJD 2/3	Average
BL	0.347	0.422	0.302	0.426	0.348	0.423	0.306	**0.485**	0.472	0.435	0.429	0.400
CA	0.439	0.563	0.352	0.569	0.437	0.564	0.385	**0.583**	**0.583**	0.566	0.563	0.510
CAA	0.436	0.560	0.350	0.567	0.438	0.562	0.373	0.574	**0.576**	0.561	0.562	0.505
CAD	0.501	0.648	0.383	0.650	0.501	0.650	0.421	0.682	**0.683**	0.653	0.649	0.584
Average	0.431	0.548	0.347	0.553	0.431	0.550	0.371	**0.581**	0.578	0.554	0.551	0.500

Table 3 shows the average fitness for every possible combination of selection and placement heuristics along the 540 instances. Two variants of the DJD heuristic, $DJD_{1/3}$ and $DJD_{1/4}$, outperformed the other selection heuristics tried. The best combination of selection and placement heuristic is $DJD_{1/3}$ + CAD, closely followed by $DJD_{1/4}$ + CAD. The placement heuristic CAD is clearly the best regardless of which selection heuristic it is paired with. For the different variations of the DJD tried, $DJD_{1/4}$ is the best when used along with the BL and CA placement heuristics and $DJD_{1/3}$ is the best when used along with the CA, CAA and CAD placement heuristics. Therefore, we found that the one-third of the object capacity for the initial fullness before trying different combinations of pieces, as stated by the original version of the DJD for 1D BPP, is also suitable for the 2D irregular BPP.

For all the problem instances types from A to R, the CAD heuristic produced better average performance when combined with all the selection heuristics. Table 4 shows the average fitness obtained by all the selection heuristics with the CAD placement heuristic. Results are reported for the CAD placement heuristic only as it produced the best performance. Type I and J are the only instance types where $DJD_{2/3}$ outperformed all others, and type I instances are the only ones where all pieces are regular (rectangles). Apart from type I, type J instances have the highest percentage of right angles. It seems that $DJD_{2/3}$ goes well along with rectangles. All type Q instances are solved to optimally by $DJD_{1/2}$ using CAD placement heuristic, and 73 % of Q instances were solved to optimally by $DJD_{1/3}$ + CAD. Optimum solutions of all type Q instances have exactly four pieces in each of 15 objects. Several instances of types B, H and O are also solved to optimally by several variations of DJD.

Table 4 shows that only four instance types (D, E, F and N) where solved better for a heuristic different from DJD. These four instance types have an average of piece area below one-tenth of the object area. As we hypothesized above, it seems that the DJD heuristic is

Table 4 Average fitness obtained by all the selection heuristics when combined with the CAD placement heuristic for each instance type. The best selection heuristic for each instance type is in bold font

	Selection heuristics										
Instance type	FF	FFD	FFI	Filler	BF	BFD	WF	DJD 1/4	DJD 1/3	DJD 1/2	DJD 2/3
A	0.486	0.600	0.371	0.601	0.491	0.600	0.379	0.598	0.596	**0.605**	0.599
B	0.606	0.753	0.460	0.753	0.611	0.754	0.549	**0.929**	**0.929**	0.756	0.753
C	0.515	0.704	0.381	0.701	0.506	0.709	0.421	0.751	**0.763**	0.723	0.702
D	0.411	0.578	0.338	0.576	0.410	**0.579**	0.362	0.574	0.566	0.576	0.573
E	0.301	**0.412**	0.230	0.411	0.298	0.411	0.224	0.393	0.399	0.403	0.406
F	0.393	0.493	0.279	**0.496**	0.388	0.493	0.297	0.491	0.493	0.493	0.494
G	0.592	0.707	0.448	0.707	0.601	0.708	0.520	**0.814**	**0.814**	0.708	0.707
H	0.603	0.747	0.458	0.746	0.622	0.747	0.518	**0.928**	**0.928**	0.746	0.746
I	0.598	0.619	0.573	0.624	0.598	0.619	0.577	0.621	0.619	0.626	**0.627**
J	0.543	0.661	0.457	0.664	0.538	0.662	0.462	0.652	0.659	0.660	**0.665**
K	0.541	0.709	0.385	0.713	0.526	0.710	0.447	0.706	**0.718**	0.700	0.708
L	0.349	0.485	0.278	0.494	0.347	0.485	0.290	**0.512**	0.499	0.502	0.489
M	0.417	0.579	0.307	0.580	0.406	0.582	0.337	0.573	**0.589**	0.584	0.58
N	0.446	0.495	0.290	**0.503**	0.445	0.495	0.371	0.493	0.493	0.497	0.499
O	0.537	0.791	0.409	0.791	0.545	0.787	0.432	**0.823**	0.812	0.791	0.81
P	0.481	0.661	0.358	0.664	0.483	0.662	0.396	**0.678**	**0.678**	0.663	0.657
Q	0.672	0.942	0.483	0.943	0.670	0.972	0.558	0.967	0.977	**1**	0.946
R	0.533	0.724	0.390	0.726	0.538	0.726	0.442	**0.771**	0.753	0.725	0.723
Average	0.501	0.648	0.383	0.650	0.501	0.650	0.421	0.682	**0.683**	0.653	0.649

not intended for instances with many small pieces. $DJD_{2/3}$ was the best heuristic for the other two instance types in our data set (I and J) with piece area average below $1/10$ of the object area.

We ran the algorithms on a Intel Core Duo 2.33 GHz PC. Computation times, measured in seconds, are shown in Table 5. The first seven selection heuristics are listed in Table 5 from fastest to slowest (left to right). All the combinations of selection and placement heuristics produced results within reasonable time (below 10 seconds). However, the DJD variants have longer running times. It was observed experimentally, for $DJD_{1/3}$ + CAD, that there is an average computational time reduction of 80 % when compared to the case where no record is kept at all.

In order to assess the statistical significance of the results, we conducted the hypothesis testing procedure called one-way repeated measures, ANOVA. We employed the solution of each available instance with the eleven different selection heuristics (including the four variations of the DJD heuristic) in combination with the CAD placement heuristic. Considering a sample size of 540 instances, we rejected the hypothesis that observed differences were due to chance in at least a pair of heuristics (p-value < 0.0001, the Greenhouse-Geisser correction was used because the assumption of sphericity has not been met). So, at least one heuristic is significantly different from the others. In order to determine which heuristics may be considered to be of similar performance and which not, we performed multiple pairwise comparisons with the Bonferroni adjustment and a significance level of 0.05. Figure 10 ranks the eleven selection heuristics considered and connects those that perform equivalently

Table 5 Average computational time (in seconds) for all the combinations of selection and placement heuristics over the 540 instances

	Selection heuristics											
Placement heuristics	WF	BF	FF	FFI	FFD	BFD	Filler	DJD 1/4	DJD 1/3	DJD 1/2	DJD 2/3	Average
BL	0.01	0.02	0.01	0.02	0.02	0.02	0.04	1.12	0.93	0.68	0.32	0.29
CA	0.41	0.76	0.83	0.85	1.20	1.18	5.77	10.04	9.72	7.83	6.61	4.11
CAA	0.44	0.81	0.80	0.86	1.12	1.22	5.45	9.89	9.54	7.89	6.62	4.06
CAD	0.47	0.95	0.94	0.90	1.33	1.42	6.18	12.52	12.37	10.4	9.02	5.14
Average	0.33	0.63	0.64	0.66	0.92	0.96	4.36	8.40	8.14	6.70	5.64	3.40

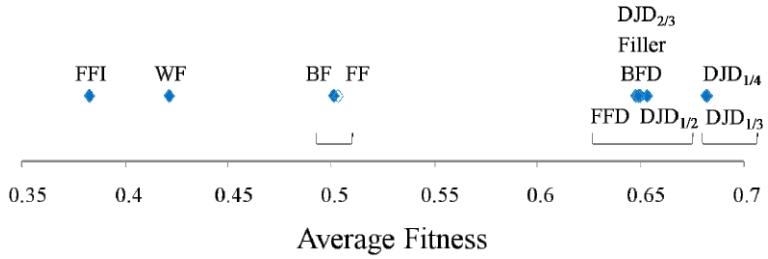

Fig. 10 Comparison of means for the 11 heuristics considered, using the Bonferroni adjustment. $DJD_{1/4}$ and $DJD_{1/3}$ are the better heuristics and there is not significant difference between them

according to the Bonferroni procedure. The FFI heuristic produced the overall lowest fitness, and it is significantly different from the other heuristics. From lowest to highest fitness, the next heuristic is WF; then, BF and FF form a group of similar heuristics since their fitness is not significantly different. Two variations of the DJD heuristic ($DJD_{2/3}$ and $DJD_{1/2}$) along with FFD, Filler and BFD perform in a similar way along the set of considered instances. The heuristics $DJD_{1/4}$ and $DJD_{1/3}$ are the best, and are significantly different from the rest. Although $DJD_{1/3}$ performs slightly better than $DJD_{1/4}$, this difference is not statistically significant.

7 Conclusions

This article proposed an adaptation, to the two-dimensional irregular bin packing problem, of the Djang and Finch heuristic originally designed for the one-dimensional bin packing problem. Four variants of the DJD heuristic (with initial fullness of 1/4, 1/3, 1/2 and 2/3, before combinations of pieces are tried to be placed within an allowed waste) were explored and compared with several alternative selection heuristics in the literature. Selection heuristics need to be paired with a placement heuristic to completely solve packing problems. Several placement heuristics were explored and the Constructive Approach with Maximum Adjacency (CAD) was found to outperform the others in our study. Also, the value of the waste incremental is an important choice in the DJD heuristic. We found, empirically, that a waste incremental of one-twentieth of the total object area represents a good balance between fast and good solutions.

An extensive empirical study was conducted over 540 irregular convex instances of different types and a wide range of characteristics. The proposed DJD heuristic was found to statistically outperform the alternative selection heuristics. Moreover, the computational time, although longer for the DJD variants is still within reasonable bounds, which was achieved by a routine keeping appropriate records to reduce the amount of redundant computation. The DJD variants with $1/4$ and $1/3$ initial fullness levels produced the best results. Therefore, the one-third of the object capacity for the initial fullness before trying different combinations of pieces, as stated by the original version of the DJD for the one-dimensional case, is also suitable in two dimensions. For further research, we plan to test this adaptation in instances with concave polygons which will increase the level of geometrical complexity.

Acknowledgements This research was supported in part by Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM) under the Research Chair CAT-144 and the Consejo Nacional de Ciencia y Tecnología (CONACYT) Project under grant 99695. Gabriela Ochoa gratefully acknowledges the British Engineering and Physical Sciences Research Council (EPSRC) for financial support under grant number EP/D061571/1.

References

Allen, S. D., Burke, E. K., & Kendall, G. (2011). A hybrid placement strategy for the three-dimensional strip packing problem. *European Journal of Operational Research*, *209*(3), 219–227. doi:10.1016/j.ejor.2010.09.023.

Bennell, J. A., & Dowsland, K. A. (2001). Hybridising tabu search with optimisation techniques for irregular stock cutting. *Management Science*, *47*(8), 1160–1172.

Bennell, J. A., & Oliveira, J. F. (2008). The geometry of nesting problems: a tutorial. *European Journal of Operational Research*, *184*(2), 397–415.

Bennell, J. A., & Oliveira, J. F. (2009). A tutorial in irregular shape packing problems. *Journal of the Operational Research Society*, *60*(S1), S93–S105.

Bennell, J. A., & Song, X. (2010). A beam search implementation for the irregular shape packing problem. *Journal of Heuristics*, *16*(2), 167–188.

Bilgin, B., Özcan, E., & Korkmaz, E. E. (2006). An experimental study on hyper-heuristics and exam timetabling. In *Proceedings of the 6th international conference on practice and theory of automated timetabling* (pp. 123–140).

Bounsaythip, C., & Maouche, S. (1997). Irregular shape nesting and placing with evolutionary approach. In *IEEE international conference on systems, man and cybernetics* (Vol. 4, pp. 3425–3430).

Burke, E. K., & Kendall, G. (1999a). Applying ant algorithms and the no fit polygon to the nesting problem. In *Australian joint conference on artificial intelligence* (pp. 453–464). London: Springer.

Burke, E. K., & Kendall, G. (1999b). *Implementation and performance improvement of the evaluation of a two dimensional bin packing problem using the no fit polygon* (Tech. Rep.). University of Nottingham. Report ASAP99001.

Burke, E. K., Hellier, R. S. R., Kendall, G., & Whitwell, G. (2006). A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research*, *54*(3), 587–601.

Burke, E. K., Hellier, R. S. R., Kendall, G., & Whitwell, G. (2007). Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research*, *179*(1), 27–49.

Dowsland, K. A., & Dowsland, W. B. (1995). Solution approaches to irregular nesting problems. *European Journal of Operational Research*, *84*, 506–521.

Dowsland, K. A., Dowsland, W. B., & Bennell, J. A. (1998). Jostling for position: local improvement for irregular cutting patterns. *Journal of the Operational Research Society*, *49*(6), 647–658.

Dowsland, K. A., Vaid, S., & Dowsland, W. B. (2002). An algorithm for polygon placement using a bottom-left strategy. *European Journal of Operational Research*, *141*(2), 371–381.

Dychoff, H. (1990). A typology of cutting and packing problems. *European Journal of Operational Research*, *44*, 145–159.

Gomes, A. M., & Oliveira, J. F. (2002). A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, *141*, 359–370.

Hifi, M., & M'Hallah, R. (2003). A hybrid algorithm for the two-dimensional layout problem: the cases of regular and irregular shapes. *International Transactions in Operational Research*, *10*, 195–216.

Hopper, E., & Turton, B. C. H. (2001). A review of the application of meta-heuristic algorithms to 2D strip packing problems. *Artificial Intelligence Review*, *16*(4), 257–300. doi:10.1023/A:1012590107280.

Hopper, E., & Turton, B. C. H. (2002). An empirical study of meta-heuristics applied to 2D rectangular bin packing—part II. *Studia Informatica Universalis*, *2*(1), 93–106.

Hu-yao, L., & Yuan-jun, H. (2006). NFP-based nesting algorithm for irregular shapes. In *Symposium on applied computing* (pp. 963–967). New York: ACM.

Kos, L., & Duhovnik, J. (2000). Rod cutting optimization with store utilization. In *International design conference*, Dubrovnik, Croatia (pp. 313–318).

Lamousin, H., & Waggenspack, J. (1997). Nesting of two-dimensional irregular parts using a shape reasoning heuristic. *Computer-Aided Design*, *29*(3), 221–238. doi:10.1016/S0010-4485(96)00065-6.

Lamousin Jr., H. J., & Dobson, G. T. (1996). Nesting of complex 2-D parts within irregular boundaries. *Journal of Manufacturing Science and Engineering*, *118*(4), 615–622. doi:10.1115/1.2831075.

Marín-Blázquez, J. G., & Schulenburg, S. (2006). Multi-step environment learning classifier systems applied to hyper-heuristics. In *Lecture notes in computer science. Conference on genetic and evolutionary computation* (pp. 1521–1528). New York: ACM.

Okano, H. (2002). A scanline-based algorithm for the 2D free-form bin packing problem. *Journal of the Operations Research Society of Japan*, *45*(2), 145–161.

Pillay, N. (2012). A study of evolutionary algorithm selection hyper-heuristics for the one-dimensional bin-packing problem. *South African Computer Journal*, *48*, 31–40.

Ramesh, R. (2001). A generic approach for nesting of 2-D parts in 2-D sheets using genetic and heuristic algorithms. *Computer-Aided Design*, *33*(12), 879–891. doi:10.1016/S0010-4485(00)00112-3.

Ross, P. (2005). Hyper-heuristics. In E. K. Burke & G. Kendall (Eds.), *Search methodologies: introductory tutorials in optimization and decision support techniques* (pp. 529–556). New York: Springer.

Ross, P., & Marín-Blázquez, J. G. (2005). Constructive hyper-heuristics in class timetabling. *IEEE Congress on Evolutionary Computation*, *2*, 1493–1500.

Ross, P., Schulenburg, S., Marín-Blázquez, J. G., & Hart, E. (2002). Hyper-heuristics: learning to combine simple heuristics in bin-packing problems. In *Lecture notes in computer science. Conference on genetic and evolutionary computation* (pp. 942–948). San Francisco: Morgan Kaufmann

Ross, P., Marín-Blázquez, J. G., Schulenburg, S., & Hart, E. (2003). Learning a procedure that can solve hard bin-packing problems: a new GA-based approach to hyper-heuristics. In *Lecture notes in computer science: Vol. 2724. Conference on genetic and evolutionary computation* (pp. 1295–1306). Berlin: Springer.

Sim, K., Hart, E., & Paechter, B. (2012). A hyper-heuristic classifier for one dimensional bin packing problems: improving classification accuracy by attribute evolution. In C. A. C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, & M. Pavone (Eds.), *Lecture notes in computer science: Vol. 7492. Parallel problem solving from nature—PPSN XII* (pp. 348–357). Berlin: Springer.

Terashima-Marín, H., Flores-Álvarez, E. J., & Ross, P. (2005a). Hyper-heuristics and classifier systems for solving 2D-regular cutting stock problems. In *Lecture notes in computer science. Conference on genetic and evolutionary computation* (pp. 637–643). New York: ACM.

Terashima-Marín, H., Tavernier-Deloya, J. M., & Valenzuela-Rendón, M. (2005b). Scheduling transportation events with grouping genetic algorithms and the heuristic DJD. In A. Gelbukh, L. De Albornoz, & H. Terashima-Marín (Eds.), *Lecture notes in computer science: Vol. 3789. MICAI 2005: advances in artificial intelligence* (pp. 185–194). Berlin: Springer.

Terashima-Marín, H., Farías-Zárate, C. J., Ross, P., & Valenzuela-Rendón, M. (2006). A GA-based method to produce generalized hyper-heuristics for the 2D-regular cutting stock problem. In *Lecture notes in computer science. Conference on genetic and evolutionary computation* (pp. 591–598). New York: ACM.

Terashima-Marín, H., Ross, P., Farías-Zárate, C. J., López-Camacho, E., & Valenzuela-Rendón, M. (2010). Generalized hyper-heuristics for solving 2D regular and irregular packing problems. *Annals of Operations Research*, *179*, 369–392. doi:10.1007/s10479-008-0475-2.

Uday, A., Goodman, E. D., & Debnath, A. A. (2001). Nesting of irregular shapes using feature matching and parallel genetic algorithms. In E. D. Goodman (Ed.), *Genetic and evolutionary computation conference. Late breaking papers* (pp. 429–434).

Wäscher, G., Hausner, H., & Schumann, H. (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, *183*(3), 1109–1130. Forthcoming special issue on cutting, packing and related problems.

Whelan, P. F., & Batchelor, B. G. (1992). Development of a vision system for the flexible packing of random shapes. In *Machine vision applications, architectures, and systems integration, proc. SPIE* (pp. 223–232).