2D-Packing Images on a Large Scale

Dominique Thiebaut
Dept. Computer Science
Smith College
Northampton, Ma 01063
Email: dthiebaut@smith.edu

***Abstract*—We present a new heuristic for 2D-packing of rectangles inside a rectangular area where the aesthetics of the resulting packing is amenable to generating large collages of photographs or images. The heuristic works by maintaining a sorted collection of vertical segments covering the area to be packed. The segments define the leftmost boundaries of rectangular and possibly overlapping areas that are yet to be covered. The use of this data structure allows for easily defining ahead of time arbitrary rectangular areas that the packing must avoid. The 2D-packing heuristic presented does not allow the rectangles to be rotated during the packing, but could easily be modified to implement this feature. The execution time of the present heuristic on various benchmark problems is on par with recently published research in this area, including some that do allow rotation of items while packing. Several examples of image packing are presented.**

***Keywords*—bin packing; rectangle packing; multi-threaded and parallel algorithms; heuristics; greedy algorithms; image collages.**

I. Introduction

We present a new heuristic for placing two-dimensional rectangles in a rectangular surface. The heuristic keeps track of the empty area with a new data structure that allows for the natural packing around predefined rectangular areas where packing is forbidden, and the packing flows in a natural way around these “holes” without subdividing the original surface into smaller packing areas. The main application for this heuristic is to generate collages of large collections of images where some images are disproportionally larger than the others and positioned in key locations of the original surface. This feature could also be applied in domains where the original surface has defects over which packing is not to take place.

We are especially interested in avoiding packings that place the larger items concentrated on one side of the surface, and keep covering the remainder of the surface using decreasingly smaller items. These are not aesthetically pleasing packings.

This form of *2D-packing* is a special case of the *2D Orthogonal Packing Problem* (OPP-2) which consists in deciding whether a set of rectangular items can be placed, rotated or not, inside a rectangular surface without overlapping, and such that the uncovered surface area is minimized. In this paper we assume that all dimensions are expressed as integers, and that items cannot be rotated during the packing, which is important if the items are images. 2D-packing problems appear in many areas of manufacturing and technology, including lumber processing, glass cutting, sheet metal cutting, VLSI design, typesetting of newspaper pages, Web-page design or data visualization. Efficient solutions to this problem have direct implications for these industries [11].

Our algorithm packs thousands of items with a competitive efficiency, covering in the high 98 to 99% of the original surface for large collections of items. We provide solutions for several benchmark problems from the literature [5], [12], [14], and show that our heuristic in some cases generates tighter packings with less wasted space than previously published results, although running slower than the currently fastest solution [15].

To improve the aesthetics of the resulting packing, we use Huang and Chen’s [13] surprising quasi-human approach borrowed from masons who pack patios by starting with the corners first, then borders, then inside these limits (also similar to the way one solves a jigsaw puzzle). Our algorithm departs from Huang and Chen’s in that it implements a greedy localized best-fit first approach and uses a collection of vertical *lines* containing *segments*. Each vertical segment represents the leftmost side of rectangular area of empty space extending to the rightmost edge of the area to cover. The collection keep the lines ordered by their x-coordinate. All the segments in a line have the same x-coordinate and are ordered by their y-coordinate. Representing empty space in this fashion permits the easy and natural definition of rectangular areas that can be excluded from packing, which in turn offers two distinct advantages: the first is that some rectangular areas can be defined ahead of time as containing images positioned at key locations, and therefore should not be packed over. The second is that subsections of the area to pack can easily be delineated and given to other threads/processes to pack in parallel. Simple scheduling and load-balancing agents are required to allow such processes to exchange items as the packing progresses.

The impetus for this algorithm is to pack a large number of images, typically thousand to millions, in a rectangular surface of a given geometry to form large-scale *collages*. In such applications items are not rotated 90 degrees since they represent images. This type of packing is referred to as *nesting* [7].

II. Review of the Literature

Possibly because of its importance in many fabrication processes [11], different forms of 2D-packing have evolved and been studied quite extensively since Garey and Johnson categorized this class of problems as NP-hard [9]. It is hence a challenge to create a comprehensive review of the literature, as any 2-dimensional arranging of rectangular items in a rectangular surface can be characterized as packing. Burke,

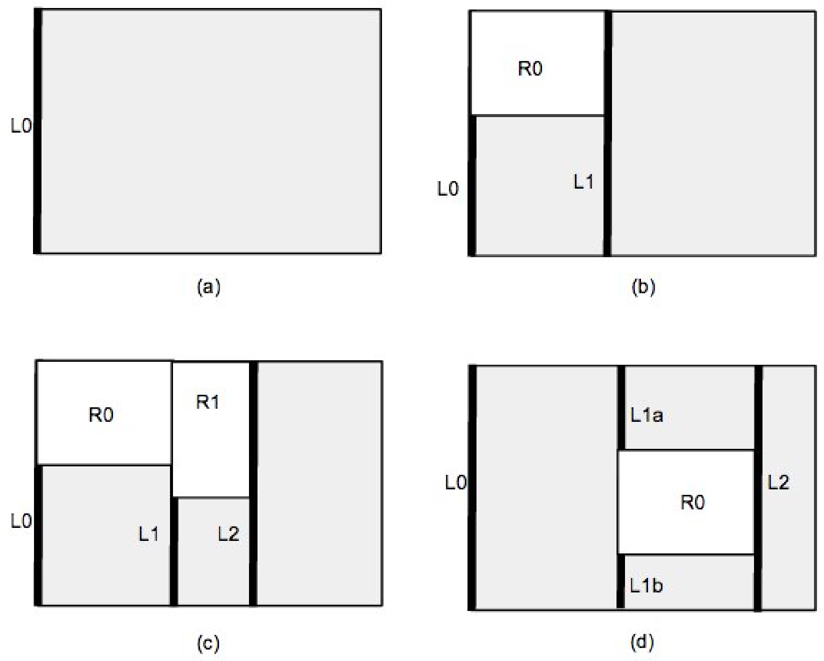

Fig. 1. The basic concept of the packing heuristic.

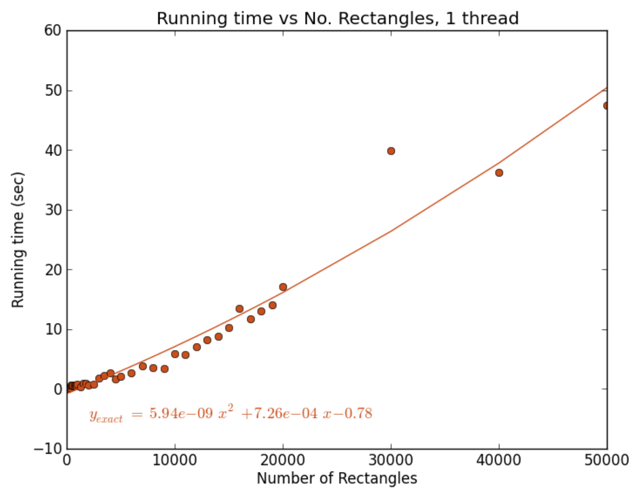

Fig. 2. Running times and regression fits for packings of 100 to 50,000 random rectangles on one core of a 3.5 GHz 64-bit AMD 8-core processor.

Kendall and Whitwell [5] and Verstichel, De Causmaecker, and Vanden Berghe [20] provide among the best encompassing surveys of the literature on 2D-packing and strip-packing research.

While exact solutions are non-polynomial in nature and slow, researchers have achieved optimal solutions for small problem sizes. Baldacci and Boschetti, for example, reports four know approaches to the particular problem of 2D orthogonal non-guillotine cutting problem [3], Beasley's optimal algorithm [4] probably being the one most often cited. Unfortunately such approaches work well on rather small problem sets. Baldacci and Boschetti, for example, report execution times in the order of tens of milliseconds to tens of seconds for problem sets of size less than 100 on a 2GHz Pentium processor.

Scientists from the theory and operations-research communities have also delved on 2D-packing and have generated close to optimal solutions [6], [8]. The *Bottom-Left* heuristic using rectangles sorted by decreasing width has been used in various situations yielding different asymptotic relative performance guarantees [1], [2], [19] [16]. Other approaches concentrate on local search methods and lead to good solutions in practice, although computationally expensive. *Genetic algorithms*, *tabu search*, *hill-climbing*, and *simulated annealing* [18] [17] are interesting techniques that have been detailed by Hopper and Turton [11] [12]. These meta-heuristics have heavy computational complexities and have been outperformed

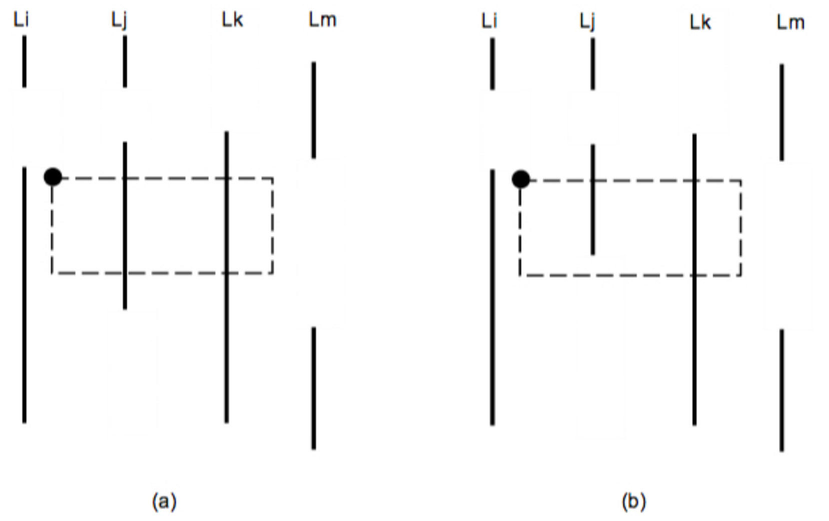

Fig. 3. Two examples of potential rectangle placements. In (a) the proposed location for the rectangle (shown in dashed line) is valid and will not intersect with other placed rectangles (not shown) because 1) its horizontal projection on the line L_i directly left of it is fulling included in a segment of L_i, and 2) its intersection with Lines L_j, L_k, and L_m is fully covered by segments of these lines. In (b) the proposed location for the rectangle is not valid, and will result in its overlapping with already placed rectangles since its intersection with Line L_j is not fully included in one of L_j's segments.

recently by simpler best-fit based approaches, including those of Hwang and Chen [13] [14], or Burke, Kendall and Whitwell [5]. Huang and Chen show that placement heuristics such as their *quasi-human* approach inspired by Chinese masons outperforms the meta-heuristics in minimizing uncovered surfaces in many cases, although requiring relatively long execution times. Burke et al. propose best-fit heuristic that is a close competitor in the minimization of the uncovered surface but with faster execution times.

Probably the fastest algorithm to date is that of Imahori and Yagiura [15] which is based on Burke et al.'s best-fit approach. Their algorithm is very efficient and requires linear space and $O(n \log n)$ time, and solves strip-packing problems where the height of the surface to pack can expand infinitely until all items are packed. They report execution times in the order of 10 seconds for problems of size 2^{20} items. Our application is slower, as our timing results show below, but provide a better qualitative aesthetic packing in a fixed size surface with similarly small wasted area. Because the time consuming operation of a collage of image is in the resizing and merging of images on the canvas which vastly surpasses our packing time by several orders of magnitude, the added value of the quality of the aesthetics of the packing makes our algorithm none-the-less attractive compared to the above cited faster contenders.

In the next section we present the algorithm, its basic data structure, and an important proposition that controls the packing and ensures the positioning of items without overlap. We follow with an analysis of the time and space complexities of our algorithm, and show that the algorithm uses linear space and requires at most O(N^3log$(N)^2$) time, although experimental results show closer to linear evolution of the execution times. This is due to the fact that the algorithm generally finds a rectangle to pack in the first few steps of the process, and the execution time is proportional mostly to the number of rectangles. Only the last few remaining rectangles take the longest amount of time to pack in the left over space. We compare our algorithm to several test cases taken from the literature in the benchmark section, and close with several examples illustrating how the algorithm operates. The conclusion section presents future research areas.

III. The Algorithm

A. Basic Data-Structures

The algorithm is a *greedy*, *localized best-fit* algorithm that finds the best fitting rectangles to pack closest to either one of the left side or top side of the surface. Figure 1 captures the essence of the algorithm and how it progresses.

The algorithm maintains ordered collections of vertical *segments* representing rectangular areas of empty space. Segments are vertical but could also be made horizontal without impeding the operation of the algorithm. These vertical segments can be thought of as the left-most height of a rectangle extending to the right-most edge of the surface to pack. Vertical segments with the same x-coordinate relative to the top-left corner of the surface to cover are kept in vertical *lines*. The algorithm's main data structure is thus a *collection* of lines ordered by their x-coordinates, each line itself a collection of segments, also ordered by their y-coordinates. The collections are selected to allow efficient *exact searching*, *approximate searching* returning the closest item to a given coordinate, *inserting* a new item (line or segment) while maintain the sorted order. *Red-black trees* [10] are good implementations for these collections.

The main property on which the algorithm relies to position a new rectangle on the surface without creating an overlap with already positioned rectangles is expressed by the following proposition:

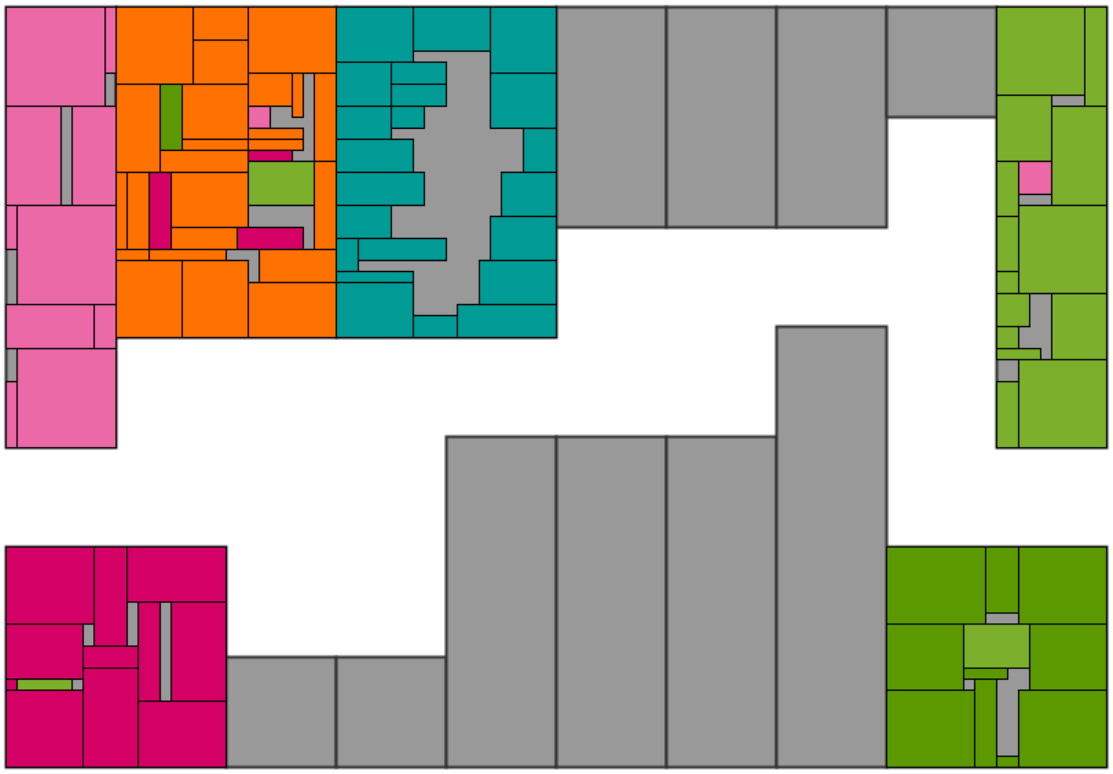

Fig. 4. The packing of 100 items in 16 objects as proposed by Hopper as the "M1a" case.

Proposition 1: A new rectangle can be positioned in the surface such that its top-left corner falls on the point of coordinates (x_{tl}, y_{tl}) and such that it will not intersect with already positioned rectangles if it satisfies two properties relative to the set of vertical lines:

1) Let L_{left} be the vertical line whose x-coordinate x_{left} is the floor of x_{tl}, i.e. the largest x such that $x <= x_{tl}$. In other words, L_{left} is the vertical line the closest to or touching the left side of the rectangle. For the rectangle to have a chance to fit at its present location, the horizontal projection of the rectangle on L_{left} must intersect with one of its segments that completely contains this projection.
2) The horizontal projection of the rectangle on *any* vertical line that intersects it must also be completely included in a segment of this line.

Figure 3 illustrates this proposition.

B. Basic Operation

The algorithm starts with two vertical lines, L_0 and L_∞. The first line originates at the top-left corner of the surface to cover, and contains a single segment whose length defines the full *height* of the surface to pack. L_∞ is a vertical line located at an x-coordinate equal to the width of the surface to pack. L_∞ contains no segments. It identifies the end of the area to pack. Any rectangle that extends past the end of the area to cover will cross L_∞, and because this one does not contain segment, the second part of the proposition above will reject the rectangle.

To simplify the description of the algorithm, we use the generic term *line* to refer to lines and segments. The algorithm packs from top to bottom and from left to right. Starting with the vertical line L_0 it finds the item R_0 with the largest height less than L_0. If several items have identical largest height, the algorithm picks the one with the largest perimeter and tests whether it can be positioned without overlapping any other already placed items. The algorithm tries three different locations: at the top of L_0, at the bottom of L_0, or at the centre of L_0. The item is positioned at the first location that offers no overlap, otherwise the next best-fitting item is tested, and so on.

The positioning of R_0 shortens L_0, as shown in Figure 1(b). A new line L_1 is added to the right of R_0 to indicate a new band of space to its right that is free for packing.

The goal is to place all larger items first and automatically the smaller ones find places in between the larger ones.

In Figure 1(c) the algorithm finds R_1 as the rectangle whose width is the largest less than L_1 and positions it against the left most part of L_1. Adding R_1 shortens L_1, indicating that all the space right of the now shorter L_1 is free for packing.

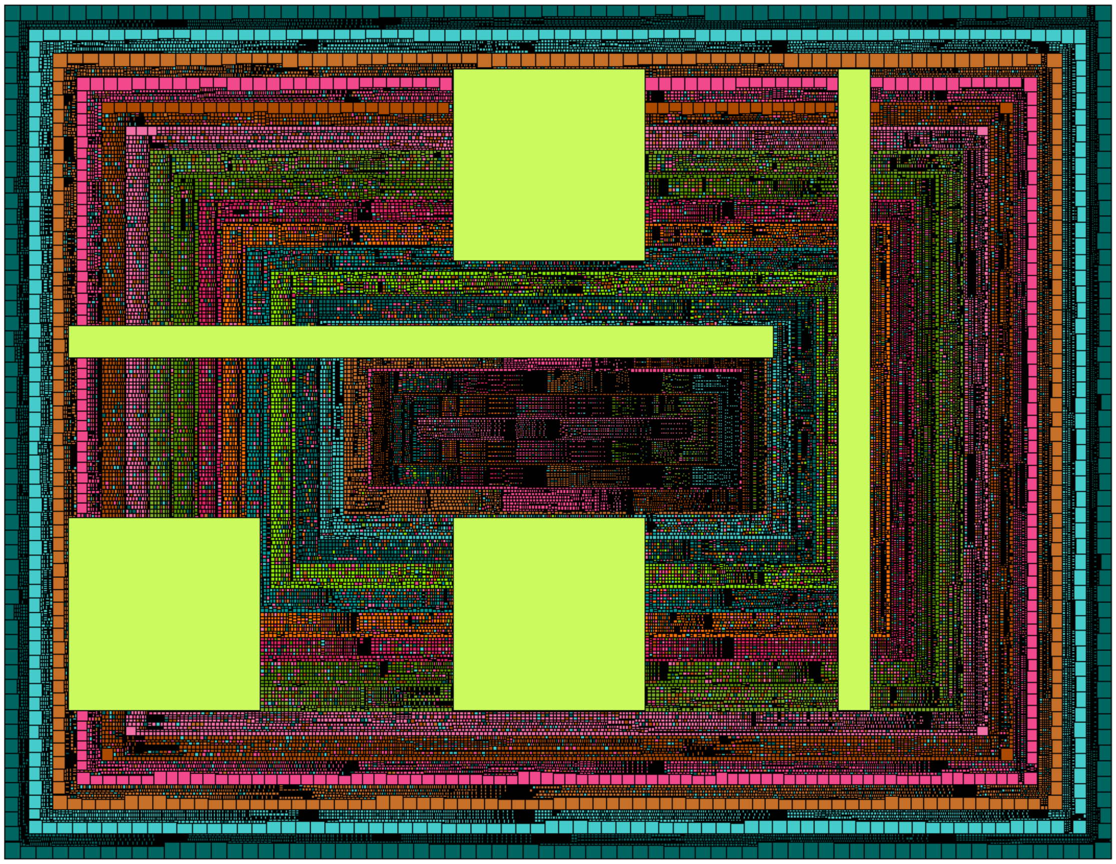

Fig. 5. The packing of 97,272 randomly generated items in a a rectangular surface. The application is multithreaded, each thread associated with a rectangular border. 5 large lime-green rectangles with different geometries are placed in various locations before the computation starts.

Again, a new line L_2 is added to delineate a band of empty space to the right of R_1.

We implement the data-structures for the lines as trees sorted on the line position relative to the top-left corner of the initial surface, so that a line or a group of lines perpendicular to particular length along the width or height of the original surface can be quickly found.

Note that in our context these line-based data-structures allow for the easy random positioning of rectangles in the surface before the packing starts, as illustrated in Figure 1(d) where a rectangle R_0 is placed first in the middle of the surface before the packing starts.

C. The Code and its Time and Space Complexities

We now proceed to evaluate the time complexity of our heuristic, whose algorithmic description is given in Algorithm 1. In it, N is the number of items to pack, $rects$ is the list of items to pack, VL the collection of vertical lines, and vl one such individual line.

Since N is the original number of items to pack, then clearly the size of VL is $O(N)$. Given a line vl of VL, we argue that the average number of segments it contains (exemplified by $L1a$ and $L1b$ in Figure 1) is O(N). The goal of the Loop starting at Line 3 is to pack all rectangles, and it will repeat N times, hence O(N). The combined time complexity of the loops at Lines 5 and 7 is O(N) because they touch at most all segments in all the lines, which is bounded by O(N). The time complexity of Line 16 is clearly O$(N \log N)$, although on the average the number of pairs to sort will be O$(\sqrt{N})$ rather than O$(N \log N)$. The loop starting at Line 17 processes at most O(N) pairs, and for each rectangle in it, must compare it to at most O(N) line vl. So it contributes O(N^2), which overpowers the sorting of the list. Therefore the combined complexity of the whole loop starting on Line 3 is O(N^3). Empirically, however, the algorithm evolves in nearly linear fashion as illustrated in Figure 2 where various selections of rectangles with randomly picked sizes are packed in a rectangular surface that is selected ahead of time to be of a given aspect ratio, and whose total area is 1/98% larger than the sum of all the items to pack. We found this approach the best for packing quickly the great majority, if not of all the items.

The space complexity is clearly O(N), since the packing of a new item in the surface introduces at most 2 new segments in the data structures.

D. Algorithmic Features

Our heuristic sports one feature that is key for our image-collage application: Rectangular areas in which packing is forbidden can easily be identified inside the main surface to

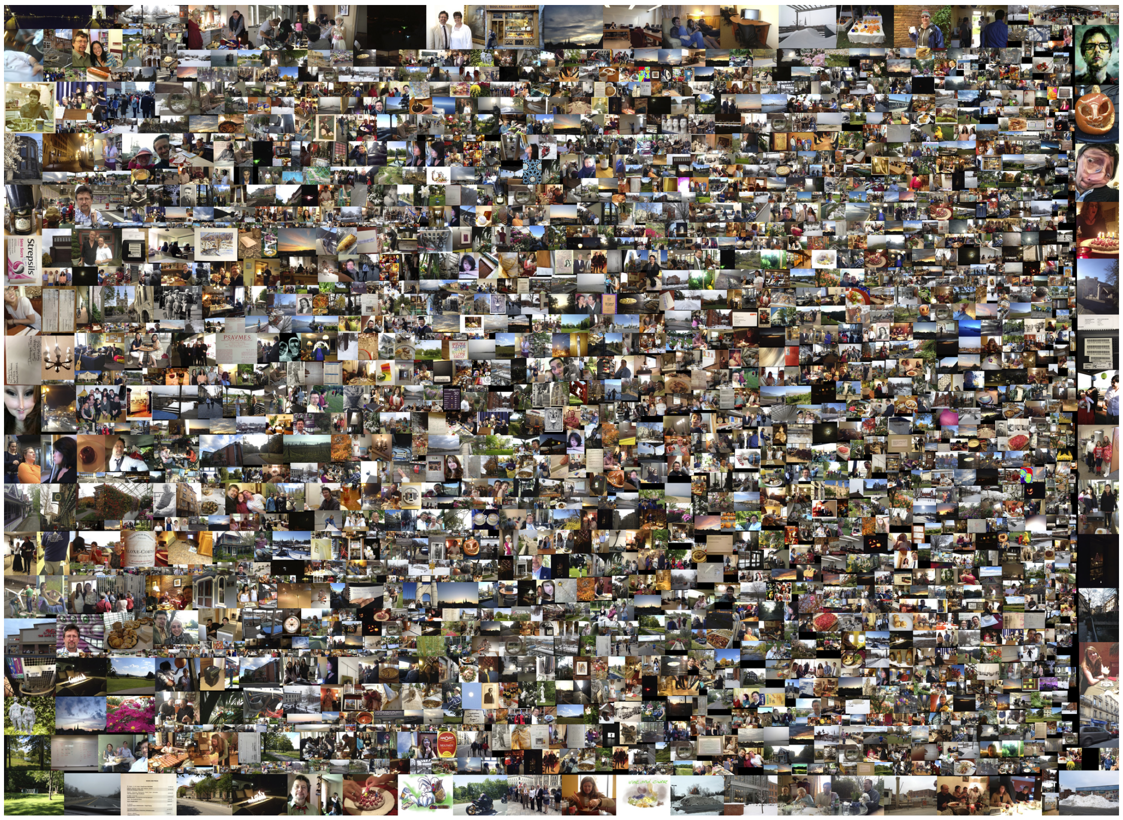

Fig. 6. The packing of 2,200 photos of various sizes and aspect-ratios, as many are cropped for artistic quality. The size of the photos is randomly picked by the algorithm. All the photos belong to this author.

be packed, either statically before starting the packing or even dynamically during run time. We refer to these areas as *empty zones*. This feature offers the user the option of positioning interesting images at key positions on the surface to be packed ahead of time. In other domains of application these could be areas with defects. Additionally, it allows parallel packing approaches where rectangular empty zones can be given out to new processes to pack in parallel, possibly shortening the execution time.

IV. Benchmarks

A set of benchmark cases used frequently in the literature are those of Hopper and Turton [12], and of Burke, Kendall and Whitwell [5]. For the sake of brevity we select a sample of representative cases and run our heuristic on each one. The computer used to run the test is one core of a 64-bit Ubuntu machine driven by a 3.5GHz AMD 8-core processor, with 16GB of ram. The heuristic is coded in Java. Note that all published results are on different types of computers, ranging from ageing memory-limited laptops to supped up desktops, all with different processor speed and memory capacities. To provide a more objective comparison, we make the following assumptions: *a*) all results reported in the literature corresponded to compiled applications that are all memory residents, *b*) they are the only workload running on the system, *c*) MIPS are linearly related to CPU frequency, and thus we scale the execution times of already published data reported by the ratio of their operating CPU frequencies to that of our processor (3.5GHz).

We follow the same procedure used by the researchers whose algorithms we compare ours to, and we run our application multiple times (in our case 30 times) on the same problem set and kept the best result.

Table I shows the scaled execution times of the various heuristics for problem sets taken from the literature. The times are those reported in the literature multiplied by a scaling factor equal to the 3.5GHz/*speed of processor*, where the processor is the one used by the researchers. For the Burke column, the speed of the processor is 850MHz. For the GRASP column, 2GHz, and for the 3-way column, 3GHz.

We observe that, as previously discovered [15] our packing efficiency improves as the number of items gets larger (in the thousand of items), which is the size of our domain of interest. The execution times of our heuristic are faster than those of Burke's best-fit, or of GRASP, and at most five times slower than the fast running 3-way best-fit of Imahori and Yagiur [15]. This difference might be attributed to either the choice of language used to code the algorithm, Java in our case, versus C for theirs, in our multithreaded approach–see the examples below–which adds a level of overhead, or possibly some inefficiency in the selection of slower data structures.

In the next section we show several packings generated by our heuristic.

TABLE I. PERFORMANCE COMPARISON TABLE

			Burke		GRASP		3-way		DT	
Case	No. items	optimal	diff.	time (s)	diff.	time (s)	diff.	time (s)	diff.	time (s)
N1	10	40	0	∼14.571	0	∼34.286	5	<0.009	0	0.05
N2	20	50	0	∼14.571	0	∼34.286	3	<0.009	6	<0.01
N3	30	50	1	∼14.571	1	∼34.286	4	<0.009	10	<0.01
N4	40	80	2	∼14.571	1	∼34.286	6	<0.009	49	<0.01
N5	50	100	3	∼14.571	2	∼34.286	4	<0.009	5	0.03
N6	60	100	2	∼14.571	1	∼34.286	2	<0.009	22	0.01
N7	70	100	4	∼14.571	1	∼34.286	7	<0.009	14	<0.01
N8	80	80	2	∼14.571	1	∼34.286	3	<0.009	23	<0.01
N9	100	150	2	∼14.571	1	∼34.286	13	<0.009	5	0.04
N10	200	150	2	∼14.571	1	∼34.286	2	0.01	10	0.03
N11	300	150	3	∼14.571	1	∼34.286	2	0.01	2	0.49
N12	500	300	6	∼14.571	3	∼34.286	5	0.02	7	0.07
N13	3152	960	4	∼14.571	3	∼34.286	4	0.20	5	0.927
C7-P1	196	240	4	∼14.571	4	∼34.286	6	<0.009	17	0.02
C7-P2	197	240	4	∼14.571	3	∼34.286	4	<0.009	41	0.02
C7-P3	196	240	5	∼14.571	3	∼34.286	5	<0.009	24	0.01

Algorithm 1 Simplified Packing Heuristic

```
1:  $N$ = dimension( $rects$ )
2:  $VL = \{L_0, L_\infty\}$
3:  while not $VL$.isempty() do
4:      $success$ = false
5:      for all line $\ell$ in $VL$ do
6:          $list$ = { } // empty collection
7:          for all segment $sl$ in $\ell$ do
8:              $rect$ = rectangle in $Rects$ with height closest to
                but less than $sl$
9:              if $rect$ not null then
10:                 $list$.add( Pair( $rect$, $sl$ ) )
11:             end if
12:         end for
13:         if $list$.isempty() then
14:             continue
15:         end if
16:         sort $list$ in decreasing order of ratio of $rect$.length to
            $sl$.length
17:         for all $pair$ in $list$ do
18:             $rect$, $sl$ = $pair$.split()
19:             if $rect$ fits in $VL$ then
20:                 pack $rect$ at the top of $sl$
21:                 update $VL$
22:                 $success$ = true
23:                 break
24:             end if
25:         end for
26:         if $success$ == true then
27:             break
28:         end if
29:     end for
30:     if $Rects$.isEmpty() then
31:         break
32:     end if
33: end while
```

V. PACKING EXAMPLES

In this section we provide several examples of packing under various conditions and constraints, some of them taken from the literature.

In Figure 4 we apply our heuristic to Hopper's *M1a case* [11] where 100 items must be packed into 16 different objects. Our algorithm also packs the objects, although this is not a requirement of the test. In this experiment our heuristic is multithreaded and several threads pack the different objects. A scheduler simply distributes the objects to separate threads, picking the largest object first and assigning it to a new thread implementing our packing heuristic. Then the scheduler picks the next largest object (in terms of its area) and assigns it to a new thread, and so on. The earliest starting threads are given a random sample of the items to pack. Threads that start last have to wait until earlier threads finish packing and return items that couldn't be packed. This automatically packs objects in such a way that as few objects as possible are packed, and some left empty, which may be desirable.

In Figure 5 the original surface is divided at run time into smaller surfaces, or *borders* one inside the other as the packing progresses, and individual threads are running the packing on individual borders. Here again the threads are given random samples of the original population of items and a load balancing scheme allows for the exchange of items between threads. This is represented by items with different colors. For example, the items associated with the first thread are all dark green, and some can be found in the light green, orange or pink borders as they are rejected by the first thread once it has packed the dark green band. Note that the utilization of the surface is 99.30%.

In Figure 5 we have placed five large items (yellow-green rectangles) on the surface before launching the packing algorithm. Notice how the heuristic naturally packs around these areas. Note also that as in Figures 4 and 5, we follow Huang and Chen's quasi-human approach [13] and pack corners and borders first before proceeding with the inside areas. Note that this modification of the algorithm fits completely with the natural properties of the heuristic, and enhances the visual aspect of the final packing.

VI. Conclusions

We have presented a new heuristic for packing or nesting two-dimensional images in a rectangular surface. The heuristic packs the items by creating a collection of segments that are maintained in two data structures, one for horizontal segments, and one for vertical segments. The segments represent the leftmost and topmost side of rectangular surfaces that extend to the edges of the original surface to pack. These data structures permit to test quickly whether a new item can be positioned in the surface without overlapping a previously placed item.

Our packing heuristic does not rotate items, but none-the-less compares favourably with other heuristics published in the literature that solve 2D-strip packing with rotation of items allowed.

The data structure used to maintain the empty areas lends itself well to positioning items in key places ahead of time, or in subdividing the original surface into multiple holes that can be either left empty, reserved for large size items, or assigned to separate processes that will pack in parallel. Such holes may contain defects (for example in a sheet of metal, or glass) that need to be avoided by the packing process. A forthcoming paper will explore scheduling and load-balancing approaches for speeding up the packing.

Because our domain of application is that of image collages, we have found that the the quasi-human approach of Huang and Chen, along with subdividing the surface into nested rectangular area significantly improves the aesthetic quality of the packing compared to most heuristic that privilege one side or corner and put all largest items there and finish packing with the smaller items at the opposite end.

References

[1] B. S. Baker, D. J. Brown, and H. P. Katseff, "A 5/4 algorithm for two-dimensional packing," *Journal of Algorithms*, vol. 2, pp. 348–368, 1981.

[2] B. S. Baker, E. G. C. Jr., and R. L. Rivest, "Orthogonal packings in two dimensions," *SIAM Journal on Computing*, vol. 9, pp. 846–855, 1980.

[3] R. Baldacci and M. A. Boschetti, "A cutting-plane approach for the two-dimensional orthogonal non-guillotine cutting problem," *European Journal of Operational Research*, vol. 183, no. 3, pp. 1136–1149, 2007.

[4] J. Beasley, "An exact two-dimensional non-guillotine cutting tree search procedure," *Operations Research*, vol. 33, no. 1, pp. 49–64, January 1985.

[5] E. K. Burke, G. Kendall, and G. Whitwell, "A new placement heuristic for the orthogonal stock-cutting problem," *Oper. Res.*, vol. 52, no. 4, pp. 655–671, Aug. 2004.

[6] E. G. Coffman, M. R. Gazey, and D. S. Johnson, *Approximation algorithms for bin-packing an updated survey*. Springer-Verlag, 1984.

[7] R. D. Dietrich and S. J. Yakowitz, "A rule-based approach to the trim-loss problem," *International Journal of Production Research*, vol. 29, pp. 401–415, 1991.

[8] H. Dyckhoff, "Typology of cutting and packing problems," *European Journal of Operational Research*, vol. 44, pp. 145–159, 1990.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman and Company, 1979.

[10] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," *Foundations of Computer Science, IEEE Annual Symposium on*, vol. 0, pp. 8–21, 1978.

[11] E. Hopper, "Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods," Ph.D. dissertation, Cardiff University, United Kingdom, 2000.

[12] E. Hopper and B. C. H. Turton, "An empirical investigation of meta heuristic and heuristic algorithms for a 2d packing problem," *European Journal of Operational Research*, vol. 1, no. 128, pp. 34–57, 2000.

[13] W. Huang and D. Chen. (2008, July) Simulated annealing. [Online]. Available: http://cdn.intechopen.com/pdfs/4629/InTech-An_efficient_quasi_human_heuristic_algorithm_for_solving_the_rectangle_packing_problem.pdf

[14] W. Huang, D. Chen, and R. Xu, "A new heuristic algorithm for rectangle packing," *Computers and Operations Research*, vol. 34, no. 11, pp. 3270–3280, November 2007.

[15] S. Imahori and M. Yagiur, "The best-fit heuristic for the rectangular strip packing problem: An efficient implementation and the worst-case approximation ratio," *Comput. Oper. Res.*, vol. 37, no. 2, pp. 325–333, Feb. 2010.

[16] C. Kenyon and E. Remilia, "Approximate strip-packing," in *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, 1996, pp. 31–35.

[17] T. W. Leung, C. K. Chan, and M. Troutt, "Application of a mixed simulated annealing genetic algorithm heuristic for the two-dimensional orthogonal packing problem," *European Journal of Operational Research*, vol. 145, no. 3, pp. 530–542, March 2003.

[18] D. Liu and H. Teng, "An improved bl-algorithm for genetic algorithm of the orthogonal packing of rectangles," *European Journal of Operational Research*, vol. 112, no. 2, pp. 413–420, January 1999.

[19] D. Sleator, "A 2.5 times optimal algorithm for packing in two dimensions," *Information Processing Letter*, vol. 10, pp. 37–40, 1980.

[20] J. Verstichel, P. D. Causmaecker, and G. V. Berghe, "An improved best fit heuristic for the orthogonal strip packing problem," *International Transactions in Operational Research*, June 2013.