JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 23 No. 1 (2015), pp. 47-62

Graph Search Approach to Rectangle Packing Problem*

Marcin Korzeń[1], Przemysław Klęsk[1]
[1]*West Pomeranian University of Technology*
Faculty of Computer Science and Information Technology
Zołnierska 49, 71-210 Szczecin, Poland
{mkorzen,pklesk}@wi.zut.edu.pl

Abstract. *A rectangle packing problem is considered, where the goal is to suitably arrange a subset of given rectangles within a container so that the area of wastes (uncovered spaces) is the smallest. We propose a reduction of this problem to a graph search problem and show possible solving approaches by means of well known BFS, Dijkstra's and A* algorithms. We explain the way we construct search graphs for the problem, taking under consideration two main variants: (1) with arbitrary straight-line cuts, (2) with straight-line cuts which must go along the whole length or width of the remaining container — 'full cuts'. We also give some insights on: optimization criterion, search stopping condition and heuristics we use. Finally, we present results of experiments carried out.*

Keywords: *rectangle packing problem, graph search algorithms, search heuristics.*

---

*This work has been conducted under the ongoing TEWI project financed by the European Fund for Regional Development, `http://www.wi.pb.edu.pl/index.php/projekty-ue/tewi`, `http://tewi.p.lodz.pl`. Information about the graph/tree searching subproject named *SaC* can be found at: `http://tewi.p.lodz.pl/Windchill/downloadedFiles/sac.pdf` (in Polish).

1. Rectangle packing problem

We consider a rectangle packing problem defined as follows. Given is a set of rectangles $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and a rectangular container C. The goal is to arrange the largest (with respect to area) possible subset of $\mathcal{R}$ within C in such a way that arranged rectangles are disjoint. We assume that only placements that are orthogonal to the sides of C are permitted. A detailed definition and introduction can be found in [1], some special variants can be also found in [2].

1.1. Reduction to graph search problem

The rectangle packing problem can be tackled by many different techniques, but it is known that most of its variants are NP hard [3]. This means that a practical solution is a compromise between computational complexity and the quality of approximated solution. We consider solving the problem by its reduction to the graph search problem. Obviously other approach are also possible [4, 2, 3, 1].

The search graph is generated in the following way. The root state is an empty container with a list of all remaining rectangles. In the first step the whole area of C constitutes a single *empty space* into which rectangles can be placed (at its left bottom corner) to produce descendant states. When some rectangle is placed into the first empty space there appear two new empty spaces that can be used in the next step. In some i-th step a graph state s consists of:

1. the list the remaining rectangles $\mathcal{R}_r(s)$, awaiting to be used;
2. the list of used rectangles $\mathcal{R}_u(s) = \mathcal{R} \setminus \mathcal{R}_r(s)$, already placed within C;
3. the list of empty spaces $\mathcal{E}(s)$, which are implied by the current arrangement of $\mathcal{R}_u(s)$;

Anytime a new rectangle is added to C, we have to update all the lists. We consider two different update procedures. The first (called ***any cuts***) corresponds to the case when the cutter can cut any straight-line shapes. The second (called ***full cuts***) concerns the case when the cutter must cut through the whole length or width of the container (or current subcontainer). It should be noted that these two cases appear in practical industrial applications. In particular, the second one is typically met when cutting glass (the glass cutter has to make a full cut to the end of the material). On the other hand the first case can be met when more advanced techniques like laser cutting are applied.

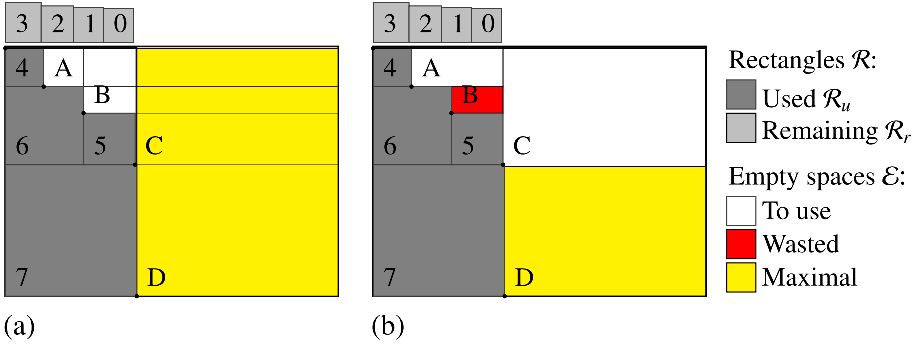


Figure 1. A state reached after 4 steps: $\mathcal{R}_r = \{0, 1, 2, 3\}$, $\mathcal{R}_u = \{4, 5, 6, 7\}$, $\mathcal{E} = A, B, C, D$. Two different kinds of cuts are presented: (a) *any cuts* possible, b) only *full cuts* allowed

Exemplary states corresponding to both cases are shown in the Fig. 1 — (a) *any cuts* possible, (b) only *full cuts* allowed. Empty spaces presented in the example have their bottom left corners marked by bolder dots. In the *full cuts* case the top right corners of empty spaces coincide with the top right corner of the whole container. Red color represents the wasted area, and yellow points out the current maximal empty space (rectangle).

Let us consider how many descendant states can be produced from a given state. Potentially, each remaining rectangle can fall into each of available empty spaces (unless it does not fit). This means that the maximum number of descendants is[1] $2 \cdot \#\mathcal{R}_r(s) \cdot \#\mathcal{E}(s)$ (allowing for a 90 degrees rotation of each rectangle). Hence, the graph grows exponentially. In practice, we may restrict the branching factor of the algorithm by limiting the number of possible placements and possible remaining rectangles, but the exponential growth still remains.

Therefore, to reduce the number of search paths a heuristic function must be defined, determining an economic order in which states are visited, and allowing for application of algorithms like: A^* and *Best-First-Search* (BFS) [5, 6].

1.2. Terminal states, optimal states, search stopping

We define terminal and optimal states in the graph in the following manner.

[1]# denotes the cardinality of a set.

Definition 1 *A state s is **terminal** if for all $R_i \in \mathcal{R}_r(s)$ there exist no $E \in \mathcal{E}(s)$ such that R_i fits into E.*

Let us denote the set of all terminal states as T.

Definition 2 *We say that a state s^* is **optimal** when*

$$\sum_{R_i \in \mathcal{R}_r(s^*)} area(R_i) = \min_{s \in T} \sum_{R_i \in \mathcal{R}_r(s)} area(R_i), \tag{1}$$

or equivalently when

$$\sum_{R_i \in \mathcal{R}_u(s^*)} area(R_i) = \max_{s \in T} \sum_{R_i \in \mathcal{R}_u(s)} area(R_i). \tag{2}$$

In order to have a graph search algorithm which is *exact* (i.e. finds an optimal solution, not an approximation of it), one must guarantee that when some terminal states occur in the main queue, an optimal state must be polled from the queue first (stopping condition) before any other terminal, but non-optimal ones. Further in the paper, we will show how this is possible but also why it is difficult and more complex computationally than approximate approaches.

For further considerations it is also useful to think of the **wasted space** a terminal state holds within its container. Let us denote it as

$$\mathcal{W}(s) = \bigcup_{E \in \mathcal{E}(s)} E, \quad \text{for } s \in T. \tag{3}$$

Obviously, equivalently to (1) and (2), the optimality can also be defined by $s^* = \arg\min_{s \in T} \text{area}(\mathcal{W}(s))$. In this context it is worth to see two general, but significantly different, cases:

1. there exist at most one terminal state with the list of remaining rectangles being empty,
2. there exist two or more terminal states with the list of remaining rectangles being empty.

It seems that the first case is more natural when thinking of optimality, because clearly we want to pack the container as densly as possible and leave the smallest remaining area on the side. In the second case, we have several equally good states (with all given rectangles fitting in the container), and the only thing one might try to do is to think if some particular geometric arrangement of all rectangles is

in some sense preferred than other arrangements[2]. Further in the paper, we shall argue for the case 2 that it is sensible to prefer states containing the **maximal empty space** (maximal with respect to area). Intuitively, this can be motivated by the fact that the maximal empty space potentially allows to pack more rectangles in the future, if for example one thinks hypothetically of some infinite supply stream of rectangles. We shall also show that it is actually good to encourage the algorithm to keep a large empty space throughout the whole search procedure and we will use this idea to build heuristics.

2. Cost functions and search heuristics

We remind the well known from A^* algorithm cost function (which decides about the order of generated states remaining in the queue to be visited):

$$f(s) = g(s) + h(s), \tag{4}$$

where g is an exact measure of cost taken to reach state s from the initial state, and h is a heuristic estimate of the remaining cost to the goal state. Additionally, if h is known to be a lower bound on the remaining cost, satisfying the monotonicity condition $\forall s_1, s_2 \; h(s_1) \leq g(s_2) - g(s_1) + h(s_2)$, then it is called an *admissible heuristic* and the algorithm is guaranteed to find an optimal solution [5, 7, 8].

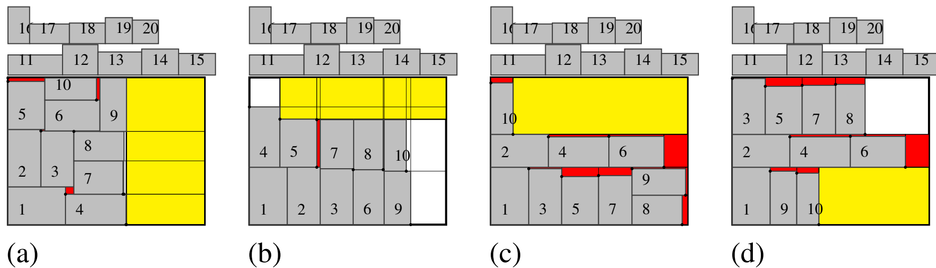


Figure 2. States consisting of 10 rectangles packed in different ways: (a) *any cuts*, heuristics: maximal empty space, (b) *any cuts*, heuristics: minimal wasted space (c) *full cuts*, heuristics: maximal empty space, (d) *full cuts*, heuristics: minimal wasted space

Many different heuristics for the rectangle packing problem can be considered [4, 2, 1]. Generally, there are three main concepts about how the search procedure

[2]To be precise the same problem may occur in the case 1.

should be carried out: 1) the wasted area should be minimized, 2) the area of remaining rectangles should be minimized, and 3) large empty spaces in the final arrangement should be maximal. Illustrations in the Fig. 2 give an intuitive outlook on these elements (wasted spaces are marked in red, maximal empty spaces in yellow). Basing on this intuition we discuss several variants of cost functions formulated for the rectangle packing problem.

2.1. Best-First-Search — "remaining rectangles"

Consider the following BFS variant of the cost function:

$$g(s) = 0, \qquad \text{for all } s; \tag{5}$$

$$h(s) = \sum_{R_i \in \mathcal{R}_r(s)} \text{area}(R_i). \tag{6}$$

It is easy to see that such a BFS is practically useless and in fact degenerates to a *Depth-First-Search* with successive rectangles being placed from the largest to the smallest. This easily can lead to solutions much worse than optimal.

2.2. Best-First-Search — "remaining rectangles minus maximal empty space"

Consider another BFS variant:

$$g(s) = 0, \qquad \text{for all } s; \tag{7}$$

$$h(s) = \sum_{R_i \in \mathcal{R}_r(s)} \text{area}(R_i) + \text{area}(C) - \max_{E \in \mathcal{E}(s)} \text{area}(E). \tag{8}$$

This cost function tries to focus simultaneously on minimizing the area of remaining rectangles and maximizing the largest empty space kept in the container. Intuitively, this is achieved by placing successive rectangles in such a way that they suitably fill holes (or partial holes) occurring in the arrangement, and possibly do not reduce the maximal empty space. Therefore, the search procedure does not degenerate to a depth-first as before.

For clarity, we explain that although the area(C) summand is constant, thus immaterial and can be skipped, we added it explicitly so that $h(s) \geq 0$, which is a common convention.

2.3. Dijkstra's algorithm — "wastes known for certain"

Recall the wasted space $\mathcal{W}(s)$ we defined in (3) for terminal states. We will formulate Dijkstra's algorithm in terms of this notion.

Imagine some partial arrangement of rectangles represented by a non-terminal state s. Assume we are able to programistically detect existing wastes in this arrangement, that is such empty spaces, or fragments of empty spaces, which *for certain* cannot accommodate any of remaining rectangles (look back to the Fig. 2, where such certain wastes are marked in red). Let us denote the union of such wastes as $\mathcal{W}'(s)$. Now, suppose that state s is driven to a terminal state t by some sequence of moves. The following simple facts can be observed:

$$\mathcal{W}'(s) \subseteq \mathcal{W}(t), \tag{9}$$

$$\mathrm{area}\left(\mathcal{W}'(s)\right) \leq \mathrm{area}\left(\mathcal{W}(t)\right), \tag{10}$$

$$\mathcal{W}'(t) = \mathcal{W}(t), \tag{11}$$

$$\mathrm{area}\left(\mathcal{W}'(t)\right) = \mathrm{area}\left(\mathcal{W}(t)\right) = \mathrm{area}(C) - \sum_{R_i \in \mathcal{R}_u(t)} \mathrm{area}(R_i). \tag{12}$$

Therefore, the following setup of the cost function:

$$g(s) = \mathrm{area}\left(\mathcal{W}'(s)\right); \tag{13}$$

$$h(s) = 0, \qquad \text{for all } s. \tag{14}$$

results in a search procedure working analogically to Dijkstra's algorithm for shortest path problems [9].

It is worth to give some remarks on such an algorithm. Surely, in its early stage the algorithm generates a wide search graph (initially exhaustive), since for most of early states no certain wastes can be seen and $g(s) = 0$. In the middle stage, where already some wastes can be seen ($g(s) > 0$) the search becomes more selective and follows more promising paths. In the stage close to terminal, some transitions from a non-terminal state s to a terminal state t may suffer from a drastic growth $g(t) - g(s)$. This is due to the fact that the waste known for certain $\mathcal{W}'$ grows slowly along some search paths and poorly approximates the final true waste $\mathcal{W}$, which sometimes reveals itself no sooner than a terminal is reached. This can also be understood by looking at observations (11) and (12). Nevertheless, terminal states with large growth of g are then placed suitably far in the main queue, and it is easy to see that the first state polled from the queue which is terminal must also be an optimal one.

2.4. A^* with inadmissible heuristic - "future wastes estimated by maximal empty space"

Consider the following cost function

$$g(s) = \text{area}\left(\mathcal{W}'(s)\right); \quad (15)$$

$$h(s) = \text{area}\left(C\right) - \sum_{R_i \in \mathcal{R}_u(s)} \text{area}(R_i) - \max_{E \in \mathcal{E}(s)} \text{area}(E). \quad (16)$$

The idea behind this function is to join features of described algorithms: Dijstra's algorithm and the BFS based on maximal empty space. This means to both: calculate wastes known for certain, and to estimate future unknown wastes by the idea related to the maximal empty space.

Estimation of the future wastes according to formula (16) can be directly explained as follows: we are making a guess that the current maximal empty space will in future be packed ideally and reveal no waste at all (optimistic part) and simultaneously we are saying that the complement of currently packed rectangles and the maximal empty space to the whole container (e.g. white-colored spaces in the Fig. 1) will in future be totally a waste (pessimistic part). Obviously this is only a guess, and the true outcome may be different if state s would be driven to the goal by optimal moves (by an oracle). However, please note that in cases when in such a true outcome the measure of waste within maximal empty space is greater than within the complement then formula (16) underestimates the remaining cost, which is a wanted property.

Checking the monotonicity condition for (16), i.e. $h(s_1) \leq g(s_2) - g(s_1) + h(s_2)$, where s_2 is some descendant (not necessarily a direct one) of s_1, leads to:

$$\begin{aligned} &-\sum_{R_i \in \mathcal{R}_u(s_1)} \text{area}(R_i) - \max_{E \in \mathcal{E}(s_1)} \text{area}(E) \\ &\qquad \leq \text{area}\left(\mathcal{W}'(s_2)\right) - \text{area}\left(\mathcal{W}'(s_1)\right) - \sum_{R_i \in \mathcal{R}_u(s_2)} \text{area}(R_i) - \max_{E \in \mathcal{E}(s_2)} \text{area}(E) \end{aligned} \quad (17)$$

which after regrouping can be rewritten (more descriptively) as:

$$\underset{s_1 \to s_2}{\Delta} \text{area}(\mathcal{R}_u) \leq \underset{s_1 \to s_2}{\Delta} \text{area}(\mathcal{W}') + \underset{s_2 \to s_1}{\Delta} \max_{E \in \mathcal{E}} \text{area}(E). \quad (18)$$

Note that $\max_{E \in \mathcal{E}(s_1)} \text{area}(E) - \max_{E \in \mathcal{E}(s_2)} \text{area}(E) \geq 0$.

The condition (18) suggests that admissibility of the proposed heuristic (16) is satisfied only for such transitions $s_1 \to s_2$ where the total area of new rectangles

added to the container is not greater than the sum of new wastes and the absolute decrease of maximal empty spaces (produced by this transition). It is easy to see many cases when this condition is not met, e.g. whenever a new rectangle is placed outside the maximal empty space and this causes no wastes noticeable immediately. Therefore, we explicitly remark that the proposed heuristic (16) is *not* admissible.

Below we propose a parameterized set of cost functions, which in the early stage of the search procedure work accordingly to the proposed formula (16) and in later stages switch off the heuristic part to zero to make the search like in Dijskstra's algorithm.

Consider the following set of cost functions with a parameter $\alpha \in [0, 1]$:

$$g(s) = \text{area}\left(\mathcal{W}'(s)\right); \tag{19}$$

$$h_\alpha(s) = \begin{cases} \text{area}\,(C) - \sum\limits_{R_i \in \mathcal{R}_u(s)} \text{area}(R_i) - \max\limits_{E \in \mathcal{E}(s)} \text{area}(E), & \text{for } \frac{\sum_{R_i \in \mathcal{R}_u(s)}}{\text{area}(C)} < \alpha; \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

Practical experiments presented in the next section show that this approach allows to make significant savings in the number of visited states and as $\alpha \to 0_+$ the quality of solution tends to the exact optimal solution. In this sense the approach may be regarded as an approximation of Dijkstra's algorithm.

3. Experiments

All experiments presented in this section have been carried out in Java using the *Search and Conquer (SaC)* library, which is dedicated to graph and game tree search problems. The library is currently being developed by authors of this paper under the ongoing TEWI project[3].

3.1. Exemplary single problem — solutions and graphs

By this section we want to give the reader a rough overview on solutions and search graphs produced by different algorithms for a single packing problem.

[3]Information about the TEWI project: `http://www.wi.pb.edu.pl/index.php/projekty-ue/tewi`, `http://tewi.p.lodz.pl`. Vision document for the *SaC* library (being a subproject within TEWI): `http://tewi.p.lodz.pl/Windchill/downloadedFiles/sac.pdf` (in Polish).

Given was a set of 15 rectangles, each with random integer sides in the range 50÷150 and a container of size 500×300. The container allowed for any orthogonal straight-line cuts (not constrained to full cuts). The branching settings were: only 1 of remaining rectangles analyzed at each step, but with 2 rotations possible, tried at 2 available empty spaces.

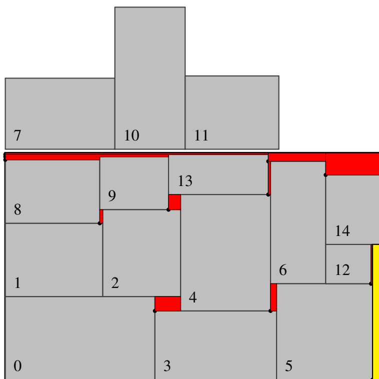
In figures 3 and 4 shown are solutions, i.e. final arrangements of rectangles, and graphs[4] produced by different algorithms. Rectangles still remaining at the moment the terminal state was reached are drawn above containers. In the graphs the path leading from the initial state (yellow) to the goal state (blue) is marked with larger circles. Illustrations are arranged in the order from the best solution towards the worst, i.e.:

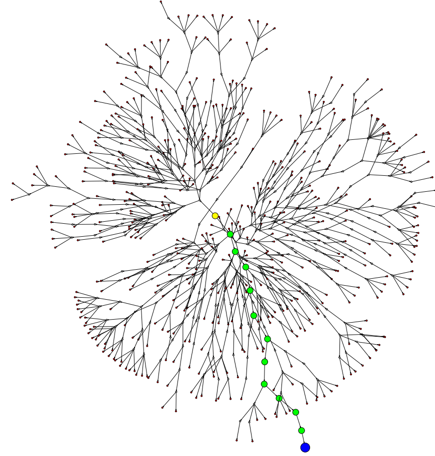
1. Dijkstra's algorithm: 6.15% of wasted area,
2. $A^*_{0.2}$ (i.e. using heuristic $h_{0.2}$): 7.82% of wasted area,
3. $A^*_{0.4}$: 10.63% of wasted area,
4. BFS guided by remaining rectangles and maximal empty space: 10.82% of wasted area,
5. BFS guided by remaining rectangles only (DFS): 22.29% of wasted area.

As the reader may note sizes of search graphs are reducing oppositely to the direction of solution quality (which is intuitive).

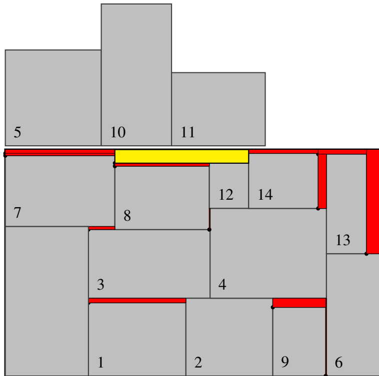
[4]Graph illustrations were drawn using *Graphviz* software, see `http://www.graphviz.org`.

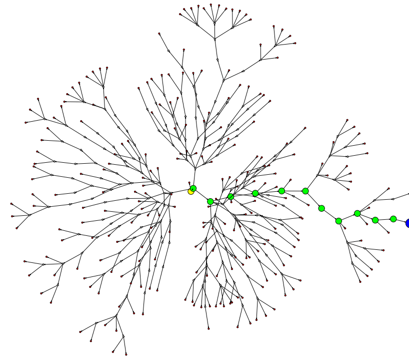
Dijkstra's algorithm "wastes known for certain":
Wastes ratio: 6.15%, visited states: 432, active states: 517.



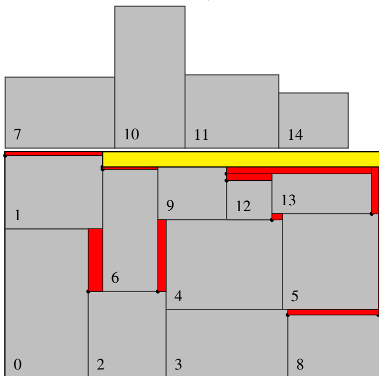


$A^*_{0.2}$"future wastes estimated by maximal empty space":
Wastes ratio: 7.82%, visited states: 216, active states: 247.





$A^*_{0.4}$"future wastes estimated by maximal empty space":
Wastes ratio: 10.63%, visited states: 101, active states: 71.



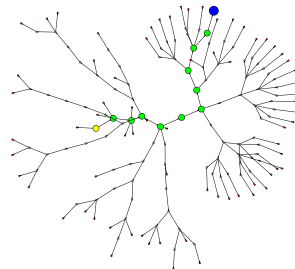
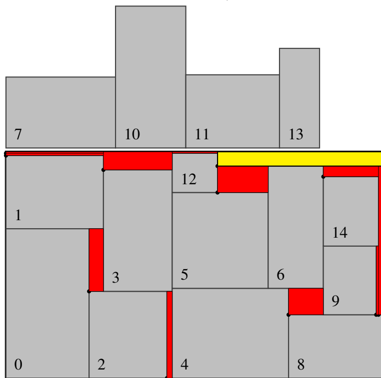
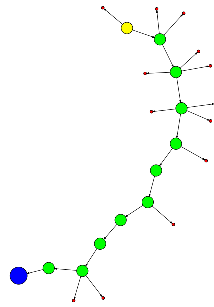


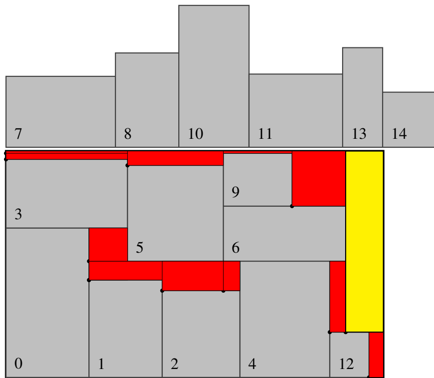
Figure 3. Solutions and search graphs produced by different algorithms (part 1)

BFS "remaining rectangles minus maximal empty space":
Wastes area ratio: 10.82%, visited states: 12, active states: 13.





BFS "remaining rectangles":
Wastes ratio: 22.29%, visited states: 10, active states: 17.



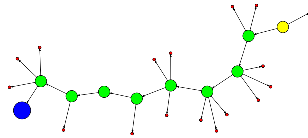


Figure 4. Solutions and search graphs produced by different algorithms (part 2)

3.2. Batch experiment: 100 random problems

In this section we present results averaged over 100 random packing problems. We focus only on Dijkstra's algorithm and A^*_α algorithms for $\alpha \in \{0.5, 0.4, 0.3, 0.2, 0.1\}$.

In each problem given was a set of 20 rectangles with random integer sides in the range $75 \div 200$ and a container of size 500×350. We tested both container types: *any cuts*, *full cuts*. The branching settings were: 2 of remaining rectangles analyzed at each step for *any cuts* container and 1 of remaining rectangles analyzed for *full cuts* container, 2 rotations possible (for both containers), tried at 2 available empty spaces (for both containers). By differentiating the first setting depending on container type we wanted to achieve similar effective branching factors — we remind that *full cuts* variant has an additional implicit branching factor of 2 related to two possible cuts (horizontal and vertical) after each new rectangle is added.

Beneath we present the Java code (using the *SaC* library) which executes the batch experiment. As one can seee there are four nested loops iterating: over 100 random problems, over 2 container types, and over 6 variants of search algorithms (last two loops); this gives in total 1 200 single experiments. The whole execution time was $\approx 0.5h$ on an Intel i7 1.6 GHz CPU with 8 GB RAM.

```
1  int howManyProblems = 100;
2  PackingState[] containerTypes = { new PackingState(), new PackingStateFullCuts() };
3  GraphSearchAlgorithm[] algorithms = { new AStar(), new Dijkstra() };
4  double[] alphas = {0.5, 0.4, 0.3, 0.2, 0.1};
5
6  PackingState.BRANCHING_BY_EMPTY_SPACES = 2;
7  PackingState.USE_FLIPFLOP = true;
8
9  for (int i = 0; i < howManyProblems; i++) {
10     List<Rectangle> problem = randomRectangles(20);
11     for (PackingState containerType : containerTypes) {
12         if (containerType instanceof PackingStateFullCuts)
13             PackingState.BRANCHING_BY_REMAINING_RECTANGLES = 1;
14         else
15             PackingState.BRANCHING_BY_REMAINING_RECTANGLES = 2;
16         packingState.setup(500.0, 350.0, new ArrayList<Rectangle>(problem));
17         for (GraphSearchAlgorithm algorithm : algorithms) {
18             int jLimit = (algorithm instanceof Dijkstra) ? 1 : heuristicsHowFar.length;
19             for (int j = 0; j < jLimit; j++) {
20                 PackingState.HEURISTICS_HOW_FAR = alphas[j];
21
22                 algorithm.setInitial(packingState);
23                 algorithm.setConfigurator(configurator);
24                 algorithm.execute(); //the search (!)
25             }
26         }
27     }
28 }
```

In the Fig. 5 we present the averaged results. The top plot shows qualities of solutions for particular algorithms, the bottom plot shows the number of visited and active states in the graph at the moment the particular algorithm was stopped.

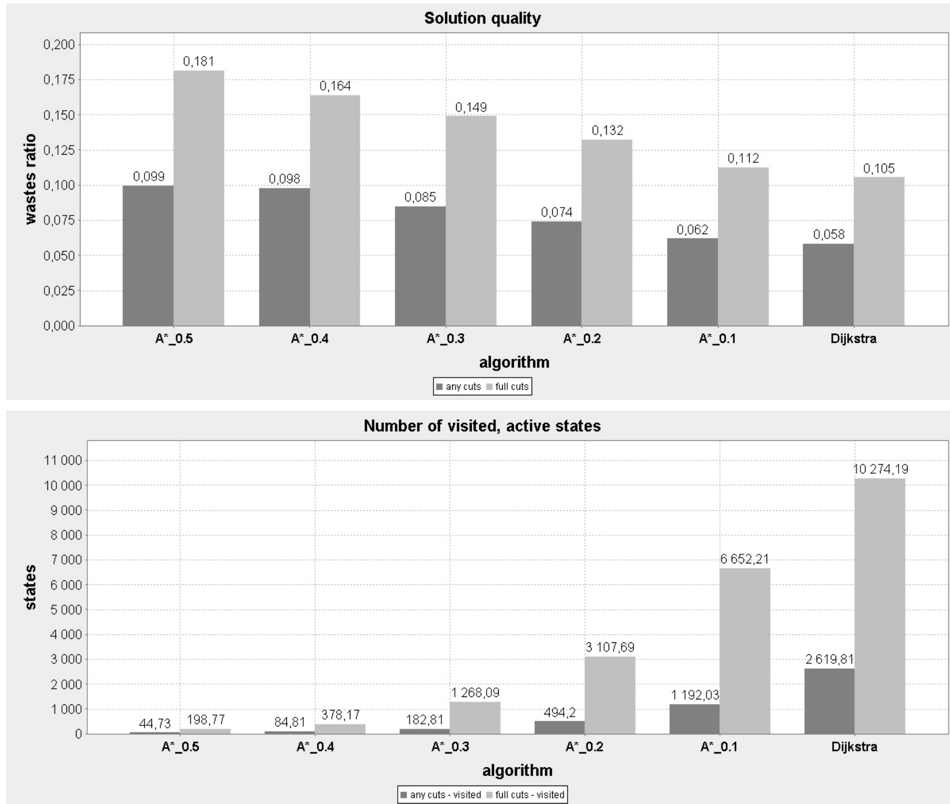


Figure 5. Results averaged over 100 random problems: quality of solutions and number of visited and active states at the stop of search

The first thing to note about the results is that packings for *any cuts* type of container (darker bars) turned out significantly better than for *full cuts* type of container (lighter bars). Typically the ratio of wasted area was approximately two times smaller for the *any cuts* container. This, of course, is a result one could expect. Simultaneously we see that the number of states visited by algorithms is also better (smaller) for the *any cuts* container, which perhaps is a slightly surprising result taking into account similar branching settings.

As regards the comparison of algorithms, the results are compliant with the ones presented for a single problem (in previous section). This means that Dijkstra's algorithm proved best with mean wastes 5.6% for *any cuts* and 10.5% for *full cuts*, whereas successive A^*_α algorithms can be regarded as approximations of best results as α tends to zero. As for the number of states in graphs one can see that for successive algorithms when 'moving away' from Dijkstra's algorithm reduce roughly about twice.

4. Summary

In the paper we have discussed some graph searching algorithms in application to the rectangle packing problem. The presented approach requires providing an implementation of a search state (in this case it is the rectangles container) which can produce descendant states by adding one of remaining rectangles. We propose some heuristics that help to guide the search procedure along more promising paths and to reduce the number of visited states. The presented application was developed as a part of the Java library named *Search and Conquer*.

References

[1] Huang, E. and Korf, R. E., *Optimal Rectangle Packing: An Absolute Placement Approach*, J. Artif. Intell. Res. (JAIR), Vol. 46, 2013, pp. 47–87.

[2] Scheithauer, G. and Sommerweiss, U., *4-Block Heuristic for the Rectangle Packing Problem*, European Journal of Operational Research, Vol. 108, 1996, pp. 509–526.

[3] Korf, R., *Optimal Rectangle Packing: Initial Results.* In: ICAPS, edited by E. Giunchiglia, N. Muscettola, and D. S. Nau, AAAI, 2003, pp. 287–295.

[4] Huang, W. and Chen, D., *An efficient heuristic algorithm for rectangle-packing problem*, Simulation Modelling Practice and Theory, Vol. 15, No. 10, 2007, pp. 1356–1365.

[5] Hart, P., Nilsson, N., and Raphael, B., *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, Systems Science and Cybernetics, IEEE Transactions on, Vol. 4, No. 2, 1968, pp. 100–107.

[6] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[7] Hansson, O., Mayer, A., and Yung, M., *Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models*, Tech. Rep. CUCS-219-85, Department of Computer Science, Columbia University, New York, USA, 1985.

[8] Russell, S. and Norvig, P., *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2002.

[9] Dijkstra, E., *A note on two problems in connexion with graphs*, Numerische Mathematik, Vol. 1, No. 1, 1959, pp. 269–271.