Aufgabe 1: Flohmarkt in Langdorf

Teilnahme-ID: 56146

Thomas Pfaller

Inhaltsverzeichnis

1 Lösungsidee

1.1 Abstrahierung der Aufgabenstellung

Jede Voranmeldung hat drei Parameter:

- Standlänge (Meter)
- Beginn (Uhrzeit)
- Ende (Uhrzeit)

Eine optimale Lösung hat die kürzesten Abstände zwischen den verschiedenen Voranmeldungen, sowohl in Bezug auf die Standlängen als auch in Bezug auf die Uhrzeiten. Die Parameter der Voranmeldungen lassen sich auf ein Rechteck übertragen:

- x: Position (Meter)
- y: Beginn (Uhrzeit)
- Breite: Standlänge (Meter)
- Höhe: $Zeitspanne = Ende - Beginn$ (Stunden)

Dadurch lässt sich dieses Problem auf das umfangreich erforschte Problem des Behälterproblems (Bin packing problem) übertragen, wenn man als Behälter/Container ein Rechteck mit der Länge 1000m und Breite 10h verwendet. Genauer gesagt, handelt es sich hier um das offline-rectangle-packing-Problem ohne Rotation und ohne Verschiebung an der x-Achse. Wenn man dieses Problem auf die Maximierung des Flächeninhaltes hin optimiert, werden automatisch die Einnahmen der Flohmarktbesitzer maximiert.
Dieses Problem ist allerdings NP-vollständig [1], wodurch es sich nichtdeterministisch in Polynomialzeit lösen lässt. Es existiert also keine effiziente Lösung für dieses Problem. Da die Beispiele bis zu 735 Rechtecke beinhalten, habe ich eine Brute-Force Lösung ausgeschlossen. Eine etwas optimierte Brute-Force Lösung ist das Branch-and-Bound-Verfahren, welches in [2] verwendet wurde. Dieser Algorithmus verwendet am Anfang einen greedy Algorithmus, um die upper bound zu bestimmen. Deswegen habe ich damit begonnen diesen greedy Algorithmus zu implementieren. Da dieser Algorithmus aber schon sehr gute Lösungen geliefert hat und es wenige Verbesserungsmöglichkeiten der Ausgabe gab, habe ich mich dafür entschieden diesen Algorithmus statt des Branch-and-Bound-Verfahrens zu verwenden, um so eine schnellere Durchlaufzeit und eine günstigere Zeitkomplexität zu erreichen.

1.2 Greedy Algorithmus

Ich verwende den Bottom-Left Algorithmus (BL) [3] mit abnehmender Breite der Rechtecke. Die Rechtecke werden also zuerst an den untersten linken Punkt gesetzt (siehe Abbildung 1). Das Prinzip alle angrenzenden Punkte zu verwenden, anstatt alle möglichen Punkte im Container-Rechteck zu betrachten findet auch in Artikel [4] Verwendung. *Hinweis*: In der gesamten Dokumentation ist die kurze Seite des Container-Rechtecks unten, da auch der Algorithmus von unten nach oben vorgeht. In der graphischen Ausgabe ist das Rechteck aber aus Platzgründen um 90°gedreht.

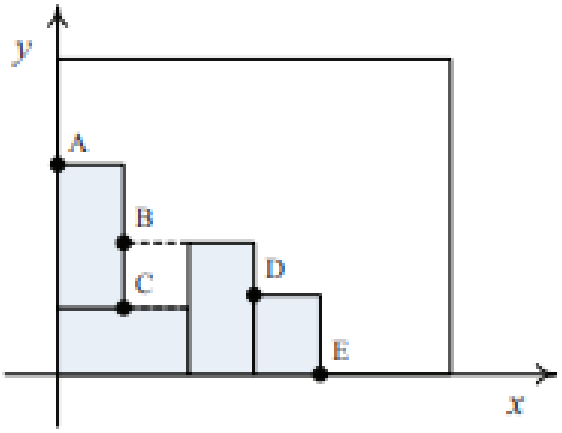


Abbildung 1: BL Platzierung; x := Zeit, y := Standlänge

In diesem Beispiel würde der Punkt E zuerst ausgewählt werden. Wenn es für diesen Punkt kein einsetzbares Rechteck gibt, wird der nächste am meisten links unten liegende Punkt ausgewählt und so weiter. Die Reihenfolge ist also:

E => C => D => B => A

Wenn es auch für den Punkt A kein einsetzbares Rechteck gibt, terminiert der Algorithmus und die gefundene Lösung wird ausgegeben.

Da es aber sein könnte, dass es an dem in Abbildung 2 gezeigten Punkt F ein passendes Rechteck gibt, werden auch die Punkte, die nicht an eine Seite eines Rechteckes angrenzen mit berücksichtigt.

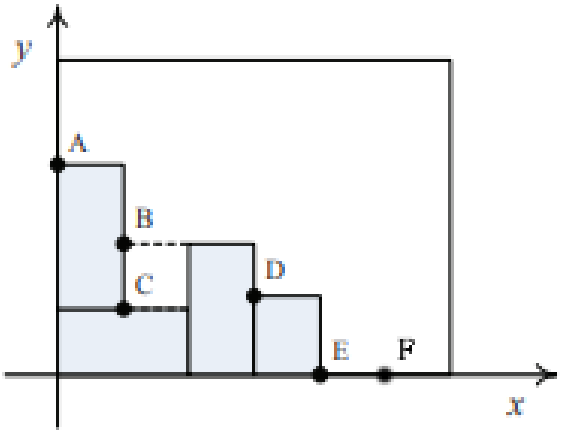


Abbildung 2: BL erweiterte Platzierung

Die neue Reihenfolge lautet dann:

E => F => C => D => B => A

Da für einen Punkt mehrere Rechtecke zur Verfügung stehen können, muss bestimmt werden, welche Rechtecke eine höhere Priorität haben. Dafür gibt es drei Möglichkeiten:

- Rechtecke absteigend nach Fläche sortieren
- Rechtecke absteigend nach Länge sortieren
- Rechtecke absteigend nach Breite sortieren

Ich habe alles ausprobiert und das Sortieren nach Breite (Zeitspanne) hat die besten Ergebnisse geliefert, weshalb ich mich für dieses Verfahren entschieden habe. Durch die Sortierung in absteigender Reihenfolge werden erst die breiteren Rechtecke platziert, was den Vorteil hat, dass kleinere Rechtecke nicht größere blockieren können, was eine bessere Lösung verhindern würde.

Da es einen großen Aufwand darstellen würde den gesamten Inhalt des Container-Rechtecks zu speichern, habe ich mich für die in [5] beschriebene Datenstruktur der sogenannten Skyline entschieden. Diese speichert nur die „Skyline“ des Containers und ist deswegen nur ein Array mit der Größe zehn ($18-8=10$). Die Punkte, an denen Rechtecke platziert werden können, liegen alle auf dieser Skyline (siehe Abbildung 3 und 4).

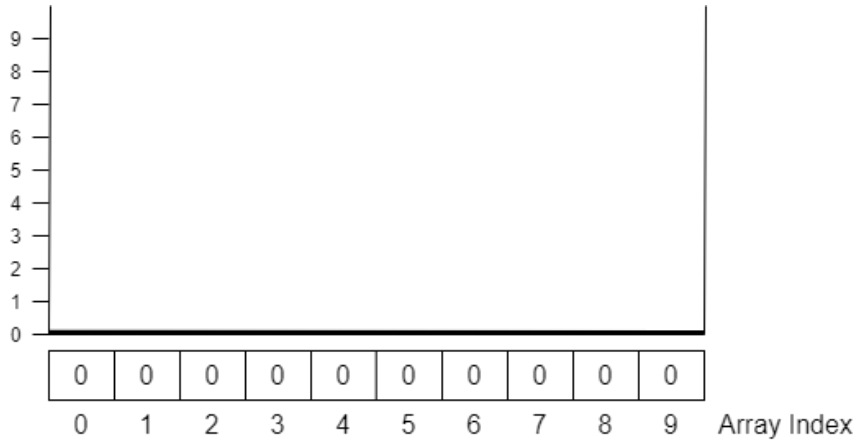


Abbildung 3: Skyline am Anfang

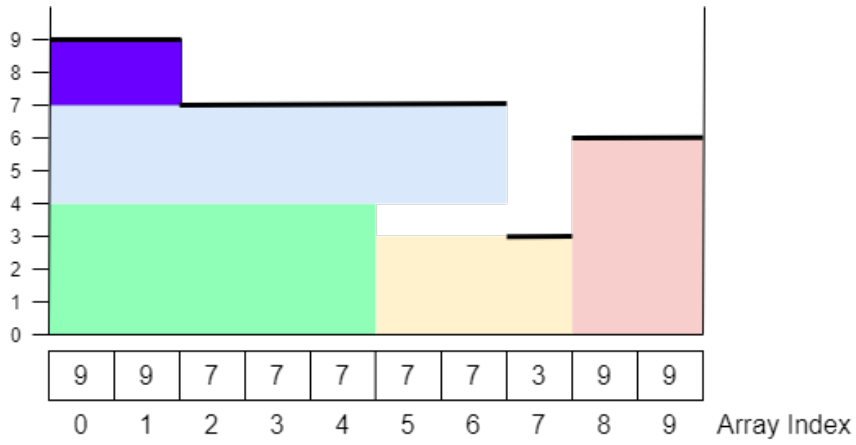


Abbildung 4: Skyline als Array abspeichern

Bei Array Index 7 ist eine Lücke zu sehen. Wenn es kein Rechteck gibt, das in diese Lücke passt, muss die Lücke aufgefüllt werden und zwar zum niedrigsten benachbarten „Turm“ (siehe Abbildung 5). Würde man den höheren Turm auswählen kann es passieren, dass dadurch eine andere Lösung „blockiert“ wird. In diesem Beispiel könnte ein Rechteck, welches bei 7 beginnt, nicht mehr gesetzt werden.

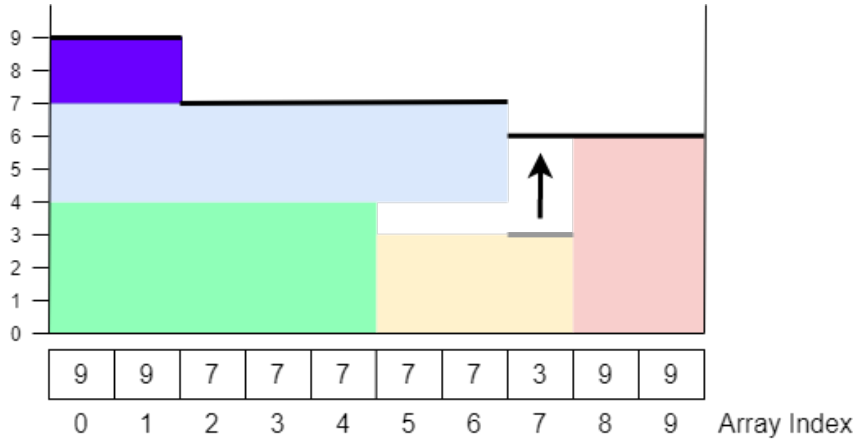


Abbildung 5: Lücke auffüllen

Dadurch kann an der Stelle 7 zum Beispiel auch ein Rechteck mit der Breite drei gesetzt werden (siehe Abbildung 6).

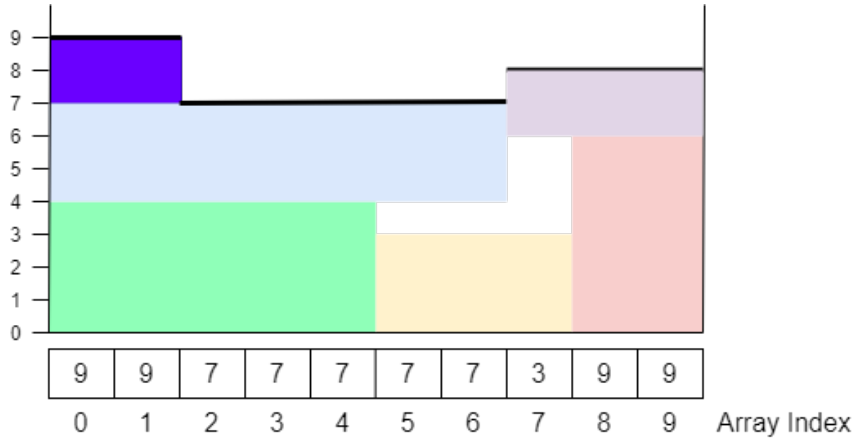


Abbildung 6: Neue Möglichkeit für die Platzierung der Rechtecke

1.3 Ausgabe

Damit die Flohmarktbesitzer ein guten Überblick darüber haben, an welcher Stelle sich ein Stand in welchem Zeitraum befindet, habe ich eine graphische Ausgabe implementiert. Außerdem wird der gesamte Flächeninhalt, also die Einnahmen ausgegeben. Damit die Flohmarktbesitzer wissen welchen Ständen sie zusagen und welchen Ständen sie absagen müssen, werden auch die verwendeten und die nicht verwendeten Stände ausgegeben.

2 Zeit- und Speicherkomplexität

Die Zeitkomplexität eines Bottom-Left-Algorithmus beträgt $O(n) = n^2$ und die Speicherkomplexität $O(n) = n$ [6], wobei n die Anzahl an Rechtecken ist. Durch die Regel, dass Rechtecke nicht horizontal (Uhrzeit) verschoben werden können, kommen für einen Punkt im Durchschnitt $n/10$ Rechtecke in Frage. Aus diesem Grund habe ich die Rechtecke in einem Dictionary abgespeichert, sodass man die Rechtecke, die bei einer bestimmten Uhrzeit beginnen, direkt aufrufen kann. Sortiert werden die Rechtecke nur einmal am Anfang mit dem Timsort-Algorithmus, welcher eine Zeitkomplexität von $O(n) = nlog(n)$ besitzt [7], was wir jedoch vernachlässigen können, da $O(n) = n^2$ überwiegt. Die Durchlaufzeit ist natürlich auch von der Größe des Containers abhängig, da sich diese Größe aber nicht verändert, ist sie für die Landau-Notation vernachlässigbar.

3 Durchlaufzeit

Die Durchlaufzeit des Algorithmus ohne graphische Ausgabe und Konsolenausgabe beschränkt sich für alle Beispiele auf 4ms. Verbessert werden könnte die Durchlaufzeit zum Beispiel indem man die Liste, in der die Punkte gespeichert sind, als Fibonacci heap implementiert, sodass die Punkte immer sortiert sind. Dadurch müssen sie zu Beginn der Auswahl eines Rechtecks nicht jedes mal neu sortiert werden. Dies würde die Zeitkomplexität für das Suchen des niedrigsten Wert von $O(k) = k$ auf $O(k) = log(k)$ reduzieren [8], wobei k hier die Breite $(18 - 8 = 10)$ darstellt.

4 Umsetzung

Nachdem in der Methode *read()* die Beispieldatei eingelesen wurde, werden die Stände in der Liste *shops* sortiert und der Methode *solve()* übergeben. In dieser Methode werden erstmal alle Stände eingesetzt, welche von 8 bis 18 Uhr gebucht sind.

```
while shops[0][0] == 10:
    shop = shops.pop(0)
    res.append([border, 0, shop[1], shop[0]])
    border += shop[1]
```

Anschließend wird das dictionary *rect_dict*, in dem sich die Rechtecke nach der Größe sortiert befinden, und die Liste *skyline* initialisiert. Die keys (Schlüssel) des dictionarys geben die Anfangszeit an.

```
skyline = [border] * 10
rect_dict = defaultdict(list)
for s in shops:
    s.insert(0, s[0]*s[1])
    rect_dict[s[3]].append(s)
```

In der Methode *greedy()* findet dann der eigentliche Algorithmus statt.

```
space = greedy(rect_dict, skyline, res)
```

Es werden zuerst die Punkte an der Skyline initialisiert, an denen Rechtecke gesetzt werden können. Der While-loop terminiert, wenn keine Rechtecke mehr übrig sind (also wenn das dictionary *rect_dict* leer ist) oder wenn die Methode zum Auswählen des nächsten Rechtecks (*feasible_rectangle()*) für keinen Punkt ein einsetzbares Rechteck findet. Anschließend wird die Skyline und die Punkte aktualisiert und zwar nur in den Spalten, wo das Rechteck gesetzt wurde. Zusätzlich wird das Rechteck auch in der Lösung *res* abgespeichert.

```
points = [[skyline[0], i] for i in range(10)]

while rect_dict:
    pos, rectangle = feasible_rectangle(points, rect_dict, skyline)
    if rectangle == 0:
        return space
    for i in range(rectangle[3], rectangle[4]):
        skyline[i] = (pos[0] + rectangle[2])
        points[i] = [skyline[i], i]
    space += rectangle[0]
    res.append([pos[0], pos[1], rectangle[2], rectangle[1]])
return space
```

Die Methode *feasible_rectangle()* wählt ein neues einsetzbares Rechteck samt der Position aus, oder gibt null zurück, wenn es kein einsetzbares Rechteck findet.

```
# returns a rectangle and its position
def feasible_rectangle(points, queue, skyline, feasible_pos=0):
    if feasible_pos == 10:
        return 0, 0
    position = 0
    p = points.copy()
    p.sort()    # getting Bottom left most points
    for e in p:
        if len(queue[e[1]]) != 0:
            if e[1] >= feasible_pos:
                position = e
                break
    if position == 0:
        return 0, 0
```

Diese Methode ist rekursiv implementiert, denn wenn es für einen Punkt kein einsetzbares Rechteck gibt, wird die Methode noch einmal aufgerufen. Es wird jedoch eine Position mehr ausgeschlossen, durch die Inkrementierung von *feasible_position*. Sobald *feasible_position* das Limit von 10 erreicht wird, gibt es keine Position, an die ein Rechteck passt und es wird 0 zurückgegeben. Der for-loop geht alle Rechtecke durch, bis es ein Rechteck gibt, welches an die Stelle *position* passt. Falls kein Rechteck passt wird die Methode nochmal aufgerufen. Es wird unterschieden ob die Rechtecke wegen der Höhe nicht hineinpassen oder wegen der Blockade durch ein anderes Rechteck. Bei letzterem muss dann die skyline erhöht werden mit der Methode *close_gap()*.

```
count = 0
for r in queue[position[1]]:
    gap = 0
    for e in range(r[3] + 1, r[4]):
        if skyline[e] > position[0]:
            gap = e  # right side of gap
            break
    if not gap:
        if position[0] + r[2] > 1000:
            count += 1
            continue
        queue[position[1]].remove(r)
        return position, r
# if not all rectangles exceed the limit 1000
if count != len(queue[position[1]]):
```

```
        close_gap(position, skyline, gap, points)
    return feasible_rectangle(points, queue, skyline, feasible_pos+1)
```

Die Methode *close_gap()* überprüft zunächst, ob sich die Position am linken oder am rechten Rand befindet, da in diesen Fällen nur in eine Richtung nach dem niedrigsten „Turm“ gesucht werden kann. Im Normalfall wird die vorherige Position mit die des Spaltendes verglichen und der niedrigste in *lowest* gespeichert. Anschließend wird die *skyline* von der Position bis zum rechten Spaltende auf die niedrigste Höhe von beiden „Türmen“ erhöht.

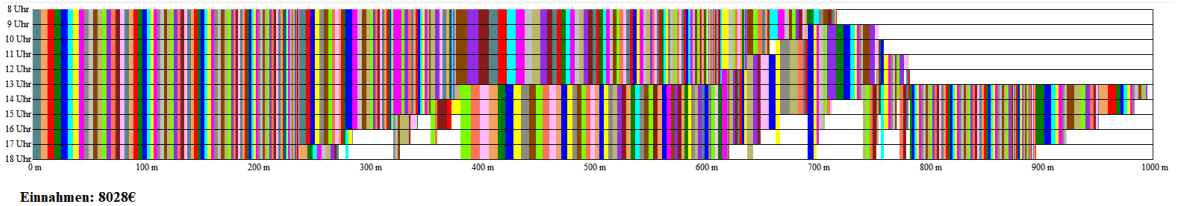
```
# fills the gap two the lowest nearest tower
def close_gap(position, skyline, gap, points):
    if position[1] == 0:
        lowest = skyline[gap]
    else:
        if position[1] == 9:
            skyline[9] = skyline[8]
            points[9] = [skyline[8], 9]
            return
        if skyline[position[1]] == skyline[position[1]-1]:
            skyline[position[1]] = skyline[gap]
            points[position[1]] = [skyline[gap], position[1]]
            return
        lowest = min(skyline[position[1] - 1], skyline[gap])
    for i in range(position[1], gap):
        skyline[i] = lowest
        points[i] = [skyline[i], i]
    return
```

5 Beispiele

In der Konsolenausgabe werden noch die platzierten und die nicht platzierten Stände ausgegeben. Aus Platzgründen sind diese Daten hier nicht abgebildet.

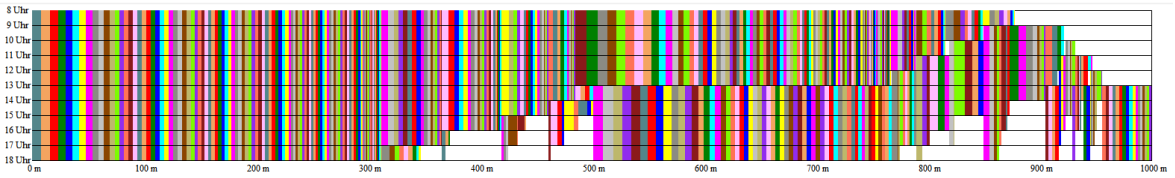
flohmark1.txt

Einnahmen: 8028€



flohmark2.txt

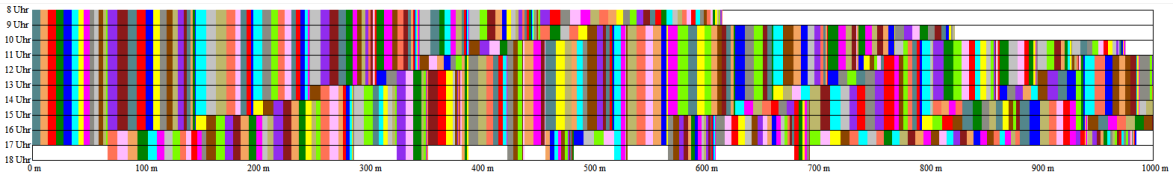
Einnahmen: 8992€



Einnahmen: 8992€

flohmark3.txt

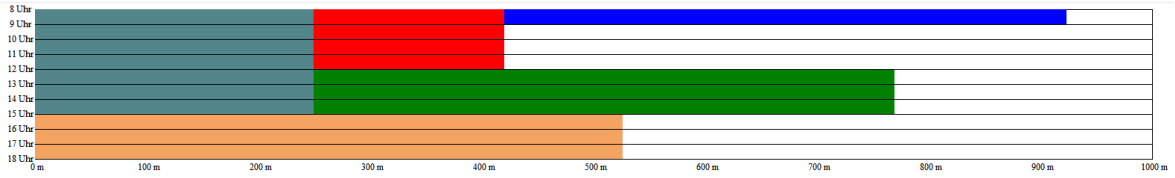
Einnahmen: 8763€



Einnahmen: 8763€

flohmark4.txt

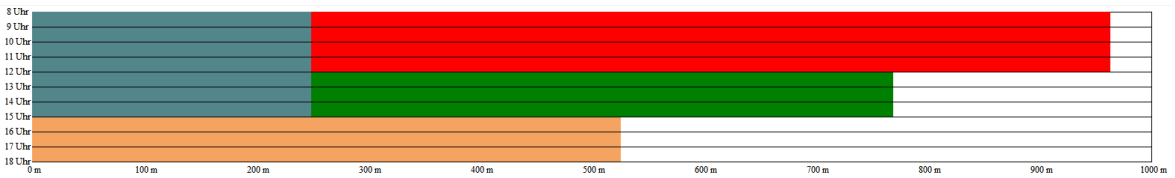
Einnahmen: 6068€



Einnahmen: 6068€

flohmark5.txt

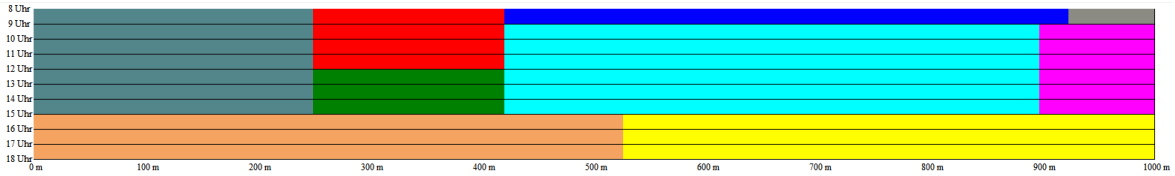
Einnahmen: 7737€



Einnahmen: 7737€

flohmark6.txt

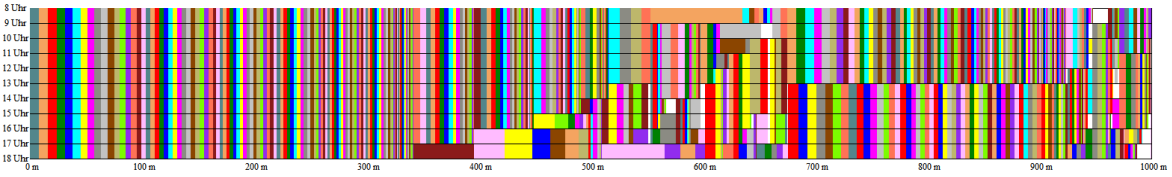
Einnahmen: 10000€



Einnahmen: 10000€

flohmark7.txt

Einnahmen: 9885€



Einnahmen: 9885€

6 Literatur

[1] E. D. Demaine, M. L. Demaine, (2007). *Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity.*
https://link.springer.com/article/10.1007/s00373-007-0713-4

[2] M. Hifi, (1998). *A best-first branch-and-bound algorithm for orthogonal rectangular packing problems.*
https://www.sciencedirect.com/science/article/abs/pii/S0969601698000264

[3] W. Huang, T. Ye (2011). *Bottom-Left Placement Theorem for Rectangle Packing.*
https://arxiv.org/pdf/1107.4463.pdf

[4] M. A. Boschetti, L. Montaletti, (2010). *An Exact Algorithm for the Two-DimensionalStrip-Packing Problem.*
https://www.researchgate.net/publication/220244051_An_Exact_Algorithm_for_the_Two-Dimensional_Strip-Packing_Problem

[5] E. K. Burke, G. Kendall, G. Whitwell (2004). *A New Placement Heuristic for the OrthogonalStock-Cutting Problem.*
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.8162&rep=rep1&type=pdf

[6] B. Chazelle (1983). *The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation.*
https://www.cs.princeton.edu/~chazelle/pubs/blbinpacking.pdf

[7] N. Auger, V. Jugé, C. Nicaud, C. Pivoteau (2018). *On the Worst-Case Complexity of TimSort.*
https://drops.dagstuhl.de/opus/volltexte/2018/9467/pdf/LIPIcs-ESA-2018-4.pdf

[8] M. L. Fredman, R. E. Tarjan (1984). *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms.*
http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Fibonacci-Heap-Tarjan.pdf