

# Aufgabe 1: Flohmarkt

Teilnahme-Id: 55628

Bearbeiter dieser Aufgabe:  
Michal Boron

Januar–April 2021

## Inhaltsverzeichnis

<b>1</b>	<b>Lösungsidee</b>	<b>2</b>
1.1	Formulierung des Problems . . . . .	2
1.2	Komplexität des Problems . . . . .	2
1.3	Heuristik . . . . .	3
1.3.1	Konversion der Eingabe . . . . .	3
1.3.2	Greedy-Algorithmus . . . . .	4
1.3.3	Heuristisches Verbesserungsverfahren . . . . .	5
1.4	Diskussion der Ergebnisse . . . . .	7
1.4.1	Grenzen der Heuristik . . . . .	7
1.4.2	Qualität der Ergebnisse . . . . .	8
1.5	Laufzeit . . . . .	11
<b>2</b>	<b>Umsetzung</b>	<b>14</b>
2.1	Klasse <code>Rec</code> . . . . .	14
2.2	Klasse <code>Hole</code> . . . . .	14
2.3	Hilfsfunktionen . . . . .	14
2.4	Klasse <code>Solver</code> . . . . .	14
<b>3</b>	<b>Beispiele</b>	<b>17</b>
3.1	Beispiel 1 . . . . .	17
3.2	Beispiel 2 . . . . .	17
3.3	Beispiel 3 . . . . .	17
3.4	Beispiel 4 . . . . .	17
3.5	Beispiel 5 . . . . .	17
3.6	Beispiel 6 . . . . .	17
3.7	Beispiel 7 . . . . .	18
3.8	Beispiel 8 . . . . .	18
3.9	Beispiel 9 . . . . .	18
3.10	Beispiel 10 . . . . .	18
<b>4</b>	<b>Quellcode</b>	<b>19</b>

# 1 Lösungsidee

## 1.1 Formulierung des Problems

Gegeben sei eine Strecke der Länge  $N$  und ein Zeitraum von  $B$  bis  $E$ . Außerdem gegeben sei eine Liste von  $Z$  Anmeldungen. Die Anmeldungen betreffen die Vermietung eines Teils der Strecke in einer konkreten Zeitspanne. Jede **Anmeldung**  $i$  besteht aus einer Strecke  $s_i$  ( $0 < s_i \leq N$ ), einem Mietbeginn  $b_i$  ( $B \leq b_i < E$ ) und einem Mietende  $e_i$  ( $b_i < e_i \leq E$ ). In diesem Problem werden Strecken in vollständigen Metern behandelt und alle Zeiten werden in vollständigen Stunden angegeben. Obwohl  $N$  auf 1000 Meter,  $B$  auf 8:00 und  $E$  auf 18:00 in der ursprünglichen Aufgabe festgelegt sind, kann die folgende Lösungsidee auf beliebige Größen, die die Aufgabenbedingungen erfüllen, übertragen werden. Das gelieferte Programm kann somit mit unterschiedlichen Werten umgehen.

Die Aufgabe ist ein Optimierungsproblem. Man soll so eine Teilfolge von  $k$  Anmeldungen wählen, dass alle gewählten Strecken in den angegebenen Zeiten vermietet werden können, d.h., für jede Anmeldung eine freie stetige Strecke der angegebenen Länge in der angegebenen Zeitspanne durchgehend zur Verfügung steht, und dazu die Mieteinnahmen möglichst hoch sind, wobei der Preis 1 Euro pro Meter pro Stunde beträgt.

Man kann das Problem auf folgende Weise modellieren. Wir setzen:  $M := E - B$ . Wir bilden ein **großes Rechteck**  $R$  der Größe  $N \times M$ . So kann man analog jede Anmeldung  $i$  als ein **kleineres Rechteck**  $r_i$  der Größe  $s_i \times m_i$  darstellen, wobei  $m_i := e_i - b_i$ .

So können wir die obige Aufgabe umformulieren: Wähle so eine Teilfolge  $Z'$  von Rechtecken aus  $Z$ , die eine Anordnung innerhalb von  $R$  bilden, dass kein Paar der Rechtecke in  $Z'$  sich überdeckt und der Gesamtflächeninhalt aller Rechtecke in  $Z'$  maximal ist. Als Fläche eines kleineren Rechtecks  $r_i$  bezeichnen wir das Produkt  $m_i \times s_i$ .

Genauer gesagt: Wenn wir das große Rechteck und die kleineren platzierten Rechtecken an einem Koordinatensystem abbilden, besitzt jedes Rechteck  $r_i$  in  $Z'$  4 Ecken, die den folgenden Punkten entsprechen:  $(x_i, b_i)$ ,  $(x_i, e_i)$ ,  $(x_i + s_i, e_i)$ ,  $(x_i + s_i, b_i)$ . Man beachte, dass  $b_i$ ,  $e_i$  und  $s_i$  fixiert sind. So ist die Aufgabe, nur  $x_i$  so zu wählen, dass die Bedingungen der Aufgabe erfüllt werden. Wir können uns dieses Problem so vorstellen, dass die Länge  $s_i$  und die Breite  $m_i$  jedes Rechtecks  $r_i$ , sowie seine Anordnung entlang der  $y$ -Achse fixiert sind, und wir das Rechteck nur entlang der  $x$ -Achse zwischen den  $x$ -Werten von 0 und  $N - s_i$  verschieben können.

In den weiteren Betrachtungen nennen wir unsere Aufgabe das FLOHMARKT-PROBLEM.

## 1.2 Komplexität des Problems

In diesem Abschnitt zeigen wir, dass FLOHMARKT-PROBLEM ein Optimierungsproblem ist, das NP-schwer ist. Es ist unmöglich, einen Algorithmus zu bilden, der in Polynomialzeit prüfen würde, ob ein Ergebnis zum FLOHMARKT-PROBLEM optimal ist. Um das zu beweisen, zeigen wir, dass das 0/1-Rucksackproblem zum FLOHMARKT-PROBLEM reduziert werden kann. Das bedeutet: Falls das FLOHMARKT-PROBLEM in Polynomialzeit gelöst werden kann, so auch das Rucksackproblem. Eine Instanz des Rucksackproblems besteht aus einer Liste von Zahlen sowie aus einem Rucksack mit einer fixierten Größe. Das Problem besteht darin, die Zahlen in den Rucksack so zu packen, dass ihre Summe maximal ist und sie die Größe des Rucksacks nicht überschreitet. Das Optimierungsproblem zum 0/1-Rucksackproblem ist NP-schwer.[3]

Gegeben sei eine Instanz des Rucksackproblems. Wir können daraus eine entsprechende Instanz des FLOHMARKT-PROBLEMS auf folgende Weise generieren. Für jede Zahl im Rucksackproblem bilden wir ein Rechteck der Breite 1 und der Länge, die dieser Zahl entspricht. So bilden wir eine Anmeldung im FLOHMARKT-PROBLEM, deren Länge der Zahl aus dem Rucksackproblem entspricht und der Unterschied zwischen dem Beginn und dem Ende der Anmeldung beträgt eine Stunde. Außerdem bilden wir ein großes, umschließendes Rechteck, dessen Länge der Größe des Rucksacks entspricht und dessen Breite ebenfalls 1 beträgt. Somit bilden wir eine Instanz eines Flohmarkts, der eine Stunde dauert und dessen Länge der Größe des Rucksacks entspricht. In dem hierdurch entstandenen Problem wählen wir die Rechtecke so, dass sie über die Grenzen des umschließenden Rechtecks nicht hinausreichen und der Gesamtflächeninhalt der kleineren Rechtecke maximal ist. Insbesondere beachte man, dass man die kleineren Rechtecke nur entlang der längeren Seite des großen Rechtecks bewegen darf. Somit ist dieses spezielle FLOHMARKT-PROBLEM äquivalent zum ursprünglichen 0/1-Rucksackproblem. Wenn wir jede Instanz des FLOHMARKT-PROBLEMS in Polynomialzeit lösen könnten, könnten wir auch jedes 0/1-Rucksackproblem in Polynomialzeit lösen. Somit ist das FLOHMARKT-PROBLEM NP-schwer.

Korf beschreibt ein sehr ähnliches Problem wie das FLOHMARKT-PROBLEM.[4] In seinem Artikel schildert der Autor das *rectangle packing problem*. In diesem Problem gibt es eine Menge von kleineren Rechtecken und man soll ein umschließendes Rechteck mit der minimalen Fläche für alle Rechtecke finden, sodass die Rechtecke sich nicht überdecken. In diesem Problem dürfen die Rechtecke nicht gedreht werden. Der Autor beweist, dass das *rectangle packing problem* NP-vollständig ist.

Da das FLOHMARKT-PROBLEM NP-schwer ist, muss man über einen optimalen Algorithmus nachdenken. Obwohl unser Problem NP-schwer ist, kann es einen parametrisierten Algorithmus geben, der dieses Problem in Pseudopolynomialzeit lösen würde. Das betrifft beispielsweise das Rucksackproblem, das sich durch einen dynamischen Ansatz in Pseudopolynomialzeit lösen lässt.[2]

Allerdings nutzen wir die Hinweise, die Cormen et al. zum Lösen von NP-vollständigen Problemen in ihrem Buch beschreiben. Grundsätzlich gibt es drei Ansätze zum Lösen eines solchens Problems.[1, S. 1106] Erstens, wenn die Eingabe klein genug ist, reicht ein Algorithmus mit einer exponentieller Laufzeit aus. Allerdings lässt sich diese Idee schlecht umsetzen, sobald die Anzahl der kleineren Rechtecke in der Eingabe sich in der Ordnung von Hunderten befindet. Die praktische Laufzeit eines exponentiellen Algorithmus wäre in diesem Fall zu groß. Zweitens beschreiben die Autoren, dass man bestimmte Grenzfälle ausgliedern kann, die sich in Polynomialzeit lösen lassen. Diesen Ansatz verwenden wir bei einigen Beispielen und er wird im Abschnitt 1.4.2 besprochen. Drittens kann man einen Algorithmus liefern, der nahezu optimale Ergebnisse in Polynomialzeit liefert — eine Heuristik.

### 1.3 Heuristik

Wir entwickeln ein heuristisches Verfahren, um diesem Problem zu begegnen. Wir lassen am Anfang einen Greedy-Algorithmus laufen, um ein Ausgangsergebnis zu erzeugen und danach entwickeln wir ein deterministisches Greedy-Verbesserungsverfahren, das die Vorgehensweise eines Bergsteigeralgorithmus (engl. *hill climbing algorithm*) nachahmt. Der Verbesserungsalgorithmus optimiert heuristisch das Ausgangsergebnis, indem er durch mehrmalige Veränderungen der Platzierung ein lokales Maximum sucht.

#### 1.3.1 Konversion der Eingabe

Wie schon im Abschnitt 1.1 erwähnt wurde, können die Voraussetzungen bezüglich des FLOHMARKT-PROBLEMS auf andere Größen übertragen werden. Da die Größen des Rechtecks  $R$  sowie des Zeitraums fest sind und auf 1000 Metern bzw. auf den Zeitraum von 8:00 bis 18:00 beschränkt sind, konvertieren wir die Eingabe, indem wir den Beginn  $B$  vom Ende  $E$  subtrahieren und den Beginn des Zeitraumes auf 0 setzen. So bleibt auch der Wert  $M$ , also die Differenz von  $E$  und  $B$ , gleich. Analog müssen wir die Eingabe für die kleineren Rechtecke  $r_i$  entsprechend konvertieren, indem wir von jedem  $b_i$  und  $e_i$  den Wert  $B$  abziehen. Für die Aufgabe selbst hat diese Konversion keine Bedeutung und funktioniert auch, wenn ein angegebener Zeitraum sich vom ursprünglichen Zeitraum unterscheidet.

Mit dieser Konversion können wir ebenfalls mehrtägige Flohmärkte behandeln. Zur Darstellung eines mehrtägigen Flohmarktes kann man die gesamten Öffnungszeiten des Flohmarkts in Stunden angeben, z. B. der Zeitraum eines Flohmarkts, der zwei Tage von 10:00 bis 17:00 dauert, kann als von 10:00 bis 17:00 und von 34:00 bis 41:00 dargestellt werden. Ebenfalls, wenn der Zeitraum eines eintägigen Flohmarkts an einer Stelle unterbrochen ist, etwa dauert der Flohmarkt von 7:00 bis 9:00 und dann von 12:00 bis 15:00, können die beiden Zeiträume in einem Beispiel angegeben werden. Wir können dann diese Instanz lösen, indem wir den ganzen Zeitraum von 7:00 bis 15:00 betrachten und in der Pausenzeit wird keine Anmeldung angenommen. Mehrere unterschiedlichen Flohmärkte mit derselben Länge  $N$  kann man analog kodieren. Beispielsweise gründet man eine Firma, die Flohmärkte an vielen Orten organisiert, aber die Firma verfügt nur über Stände mit einer festen Länge. Dann muss man die Eingabe nur entsprechend kodieren.

Außerdem werden im ursprünglichen Problem alle Zeiten in vollständigen Stunden angegeben. Diese Aufgabe kann sehr leicht zu Zeiten in Minuten ergänzt werden. Das lohnt sich vor allem dann, wenn die Öffnungszeiten zur halben Stunde fällt. Dazu konvertiert man die Eingabe am Anfang auf folgende Weise: Man kann einfach alle Zeiten zu Minuten umrechnen, indem man vollständige Stunden mal 60 multipliziert. Obwohl die weiteren Betrachtungen sich grundsätzlich auf vollständige Stunden beziehen (wie in der Aufgabenstellung), soll man nicht vergessen, dass alle diesen Gedanken sich auf Minuten übertragen lassen.

### 1.3.2 Greedy-Algorithmus

Wir bilden das große Rechteck  $R$  auf ein Koordinatensystem ab. Die Seite der Länge  $N$  verläuft entlang der  $x$ -Achse und die Seite der Länge  $M$  entlang der  $y$ -Achse. Der Wert  $B$  (nach der Konversion) wird entsprechend am Punkt  $(0, 0)$  abgebildet (s. Abb. 1)

Die Größen  $N$  und  $M$  sind im Programm fest, unabhängig davon, wie viel sie betragen. Außerdem wurde im Abschnitt 1.1 festgestellt, dass die Größen  $s_i$ ,  $b_i$  und  $e_i$  des jeweiligen Rechtecks  $r_i$  fest sind und dass wir ein Rechteck  $r_i$  nur entlang der  $x$ -Achse, also entlang der Seite der Länge  $N$  des Rechtecks  $R$ , bewegen dürfen. So bietet sich eine Verteilung der kleineren Rechtecke  $r_i$  auf kleinere **Streifen** der Länge  $N$  im Rechteck  $R$  entlang der  $y$ -Achse an (s. Abb. 1a). Die Breite eines solchen Streifens ist äquidistant für alle Streifen und, da man Stände am Flohmarkt nur zu vollständigen Stunden vermietet, beträgt die Breite eines Streifens 1 Stunde.<sup>1</sup> Ein Streifen ist ein Teil des Rechtecks  $R$  und jeder Streifen ist am Anfang leer, da keine Rechtecke in  $R$  gelegt sind. Legen wir die folgende Schreibweise fest: Ein Streifen im Rechteck  $R$ , der die Stunde  $k$  betrifft, also in der Stunde  $k$  beginnt und in der Stunde  $k + 1$  endet, nennen wir *Streifen  $k$* . Außerdem besitzt jeder Streifen  $k$  eine Liste  $S_k$ . Zu dieser Liste gehören alle kleineren Rechtecke, die zum Streifen  $k$  gehören.

Nach der Konversion der Eingabe bilden wir eine Liste  $Z$ , in der jedes Rechteck  $r_i$  mit seinen genannten Größen  $s_i, b_i, e_i$  gespeichert wird. Dann iterieren wir über jedes Rechteck  $m_i$  in  $Z$  und fügen es in jede Liste  $S_j$  für alle  $j$  hinzu, die die folgende Bedingung erfüllen:  $b_i \leq j < e_i$ . Das bedeutet, dass z. B. ein Rechteck von  $b_i = 1$  (nach Konversion, in vollständigen Stunden) bis  $e_i = 5$  in den folgenden Listen enthalten ist:  $S_1, S_2, S_3, S_4$ . In  $S_5$  wird es nicht enthalten sein, da die Miete mit dem Anfang der 5. Stunde endet. Wie die Listen  $S_j$  implementiert werden, lesen Sie in der Umsetzung.

Nach dieser Vorbereitung der Eingabe erfolgt der Lauf unseres Greedy-Algorithmus, der das Ausgangsergebnis liefert. Wir sortieren die Rechtecke  $r_i$  in jeder Liste  $S_j$  unabhängig voneinander nach folgenden Kriterien in dieser Reihenfolge: 1) fallend nach dem Wert  $e_i$ , 2) aufsteigend nach dem Wert  $b_i$  und 3) fallend nach der Fläche jedes Rechtecks (mehr zu den Sortierkriterien im Abschnitt 1.4.2). Somit sind die ersten Rechtecke in jeder Liste  $S_j$  diejenigen, deren Wert  $e_i$  am größten ist — oft diejenigen, die am breitesten in  $S_j$  sind. Die Reihenfolge wurde so gewählt, damit wir in dieser Reihenfolge versuchen, die Rechtecke aus den Listen in die Streifen im großen Rechteck  $R$  zu platzieren. Die Idee hinter dieser Platzierung ist, dass wir zuerst die breitesten Rechtecke in einer Liste  $S_j$  „links“, also an niedrigeren  $x$ -Werten, platzieren, so weit es geht. Dann füllen wir die Lücken „rechts“ (an größeren  $x$ -Werten) mit schmalere Rechtecken. Die grobe Idee ist, dass wir das Rechteck  $R$  quasi vom Punkt  $(0, 0)$  bis zum Punkt  $(N, E)$  Streifen für Streifen mit immer schmalere Rechtecken füllen.

Bevor wir zur Beschreibung des Algorithmus übergehen, legen wir noch fest, was eine Lücke ist. Als **Lücke** bezeichnen wir eine Strecke zwischen zwei Rechtecke in einem Streifen. Jede Lücke betrifft einen konkreten Streifen  $j$  und hat die Koordinaten  $x_1$  und  $x_2$  (stets:  $x_1 < x_2$ ), die den Eckpunkten der zwei Rechtecken in  $j$  bzw. den Seiten des Rechtecks  $R$  entsprechen. Man beachte insbesondere, dass es keine Lücke zwischen zwei Rechtecken gibt, die eine gemeinsame Seite haben. Als *Größe* oder *Länge* einer Lücke bezeichnen wir:  $x_2 - x_1$ .

Wir verarbeiten Streifen für Streifen in der aufsteigenden Reihenfolge der  $y$ -Werte, beginnend mit dem 0-ten Streifen. Wir iterieren durch jede Liste  $S_j$  und untersuchen jedes Rechteck  $r_i$  in  $S_j$ , ob sein Wert  $b_i$  gleich dem Wert  $j$  ist, also ob das Rechteck (die Anmeldung) im Streifen  $j$  beginnt. Außerdem prüfen wir, ob das Rechteck bereits platziert wurde. Wenn die Werte  $b_i$  und  $j$  übereinstimmen und  $r_i$  noch nicht platziert wurde, suchen wir von  $x = 0$  bis  $x = N$  nach der ersten freien Lücke im Streifen  $j$ , die mindestens so groß ist wie die Länge des Rechtecks  $s_i$ . Wenn es so eine Lücke gibt, legen wir  $r_i$  ins Rechteck  $R$  und gehen zum Rechteck  $r_{i+1}$ . Man beachte, wenn ein Rechteck  $r_i$  in  $R$  platziert wird, muss ein in alle Streifen, zu denen es gehört, eingefügt werden. Auf der Abbildung 1b sieht man den verarbeiteten Streifen  $S_0$ . Insbesondere erkennt man gut die Reihenfolge der Sortierkriterien der Rechtecke im Streifen.

In diesem einfachen Algorithmus nutzt man beim Platzieren eines Rechtecks den Vorteil, dass beim Streifen  $j$  nur ein Rechteck  $r_i$  platziert werden kann, das in diesem Streifen beginnt — es gilt:  $b_i = j$ . Natürlich können andere Rechtecke bereits platziert sein, aber unsere Vorgehensweise sichert uns, dass es für ein Rechteck  $r_i$  eine Lücke der Größe  $s_i$  über diesem Rechteck in den weiteren Streifen  $b_i + 1, b_i + 2, \dots, e_i - 1$  gibt, wenn der Algorithmus entscheidet, dieses Rechteck in  $R$  zu platzieren. Diese Beobachtung ist offensichtlich wahr, da man die Streifen von „unten“ (beginnend mit den niedrigeren  $y$ -Werten im Koordinatensystem) nach „oben“ verarbeitet und bei jedem Streifen  $j$  prüft, ob es eine genügend große Lücke für ein Rechteck  $r_i$  gibt. Wenn es eine Lücke im Streifen  $j$  für ein Rechteck  $r_i$  mit

<sup>1</sup>Wenn man Zeiten zu vollständigen Minuten betrachtet, wird  $R$  analog in äquidistante Streifen mit Breite von 1 Minute aufgeteilt.

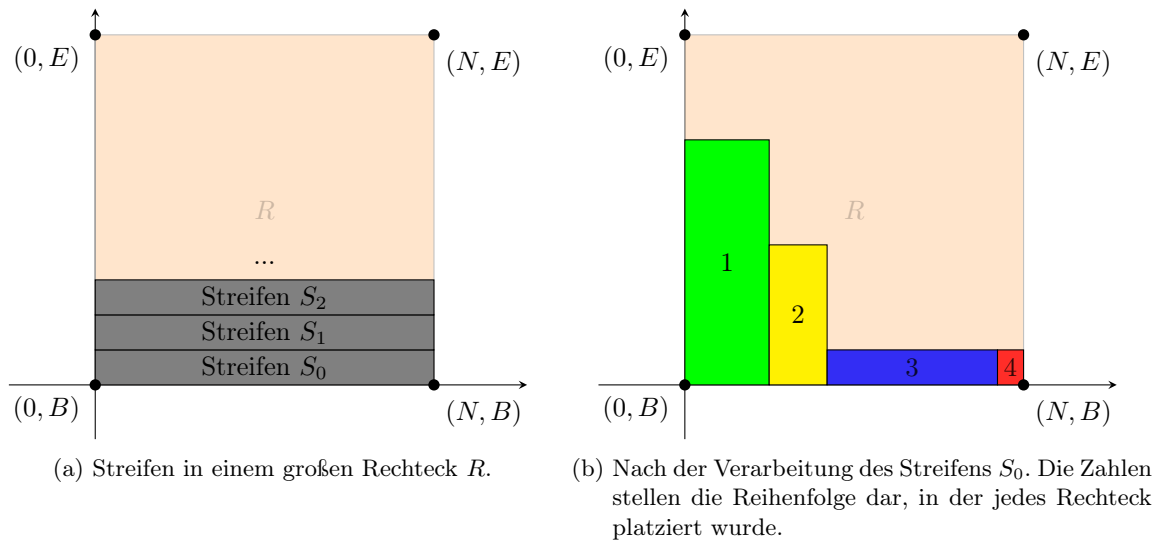


Abbildung 1: Die Abbildung des Rechtecks  $R$  auf einem Koordinatensystem. Die Seiten entlang der  $x$ -Achse haben die Länge  $N$  und die Seiten entlang der  $y$ -Achse haben die Länge  $M$ .

$b_i = j$  nicht gibt, bedeutet das, dass es im Streifen  $j$  und möglicherweise in weiteren Streifen  $j+1, j+2, \dots$  ein Rechteck gibt, das die Platzierung von  $r_i$  in  $j$  unmöglich macht.

Nachdem alle Streifen verarbeitet worden sind, ist unser Ausgangsergebnis erzeugt.

Man kann leicht begründen, dass der vorgestellte Algorithmus als Greedy klassifiziert werden kann. Mit jedem Schritt des Algorithmus wird die aktuell beste Verbesserungsmöglichkeit gewählt. Der Algorithmus nutzt die sortierte Reihenfolge der Rechtecke in den Listen  $S_j$ , um anhand des aktuellen Standes im Streifen eine Entscheidung zu treffen, ob ein Rechteck  $r_i$  in  $R$  platziert werden kann.

### 1.3.3 Heuristisches Verbesserungsverfahren

Man kann leicht feststellen, dass, wenn alle Rechtecke aus  $Z$  im Laufe des Greedy-Algorithmus in  $R$  platziert wurden oder wenn die ganze Fläche von  $R$  bedeckt wurde, das Problem für diese Eingabe optimal gelöst wurde. Allerdings lässt sich nicht nachweisen, dass der vorgestellte Algorithmus stets eine optimale Platzierung liefert. Hingegen kann man sogar feststellen, dass es bessere Ergebnisse gibt als die, die am Anfang geliefert werden.

Wir probieren, das Ausgangsergebnis heuristisch zu verbessern. Bezeichnen wir ab jetzt ein beliebiges Ergebnis, also eine beliebige Anordnung der kleineren Rechtecke innerhalb des großen Rechtecks  $R$ , die unser Programm liefert, als  $C$ . Insbesondere nennen wir unser Ausgangsergebnis  $C_A$ .

Offensichtlich kann man mithilfe des obengenannten Greedy-Algorithmus das Ausgangsergebnis nicht optimieren. Wir haben begründet, dass dieser Algorithmus an jeder Stelle stets die aktuell optimale Variante wählt. Außerdem dürfen wir diesen Algorithmus nicht nochmals nutzen, da wir voraussetzen, dass die Streifen in der aufsteigenden Reihenfolge einer nach dem anderen verarbeitet werden. Dann kann es sein, dass sich eine Lücke der Länge  $\ell$  zwischen zwei Punkten  $(x_j, j)$  und  $(x_j + \ell, j)$  in einem Streifen  $j$  befindet und dass ein Rechteck  $r_i$  mit  $s_i \leq \ell$  theoretisch hineinpassen würde, aber es ist nicht mehr gesichert, dass es die Lücken direkt darüber in oberen Streifen  $j+1, j+2, \dots$  geben würde.

Deshalb führen wir ein neues Verfahren ein. Sei  $C$  eine beliebige Platzierung von Rechtecken innerhalb von  $R$ . Nennen wir  $C$  das *aktuelle Ergebnis*. Die allgemeine Idee lautet: Ein nicht platziertes Rechteck  $r$  wird in  $R$  platziert, ggf. alle Rechtecke, die mit  $r$  kollidieren, werden aus  $R$  entfernt und die neu entstandenen Lücken werden mit nicht gelegten Rechtecken gefüllt. So kommt man auf einen neuen Zustand  $C'$ , also eine neue Platzierung der Rechtecke. Es wird dann überprüft, ob der Gesamtflächeninhalt aller platzierten Rechtecke in der Platzierung  $C'$  größer ist als der in der Platzierung  $C$ . Wenn ja, wird  $C'$  das aktuelle Ergebnis und der Vorgang wiederholt sich, bis es nicht mehr möglich ist, einen Zustand  $C$  weiter zu verändern. Wir sehen, dieser Ansatz nimmt viel Inspiration aus der Idee der Bersteigeralgorithmus, dennoch lässt er sich schlecht als einer klassifizieren. Unser Algorithmus ist zu deterministisch, es fehlen starke Mutationen in seinen Lauf, die das Ergebnis deutlich beeinflussen könnten, und es fehlen Anwendungen der Randomisierung.

Wie kommt es zur Veränderung der Platzierung und wann bestimmen wir, dass es unmöglich ist, einen

Zustand weiter zu verändern? Unser Verbesserungsverfahren arbeitet mit Lücken, die nach der Platzierung  $C_A$  entstehen. Die Idee ist, man findet eine Lücke in einem Streifen und man legt ein noch nicht platziertes Rechteck  $r$  in die Lücke, ggf. muss man die Rechtecke, die mit  $r$  kollidieren, aus der Platzierung entfernen und somit entstehen neue Lücken, die mit anderen nicht gelegten Rechtecken gefüllt werden können. So kommt man auf ein neues Ergebnis. Man hört auf, wenn es keine Lücken mehr gibt, für die ein Rechteck zum Platzieren zur Verfügung steht.

Zuerst muss man die nicht platzierten Rechtecke für jeden Streifen bestimmen. So legen wir für jeden Streifen  $j$  eine Liste  $U_j$  fest, in der sich alle Rechtecke aus dem Streifen  $j$  befinden, die nicht platziert sind. Die Liste  $U_j$  muss man auf eine bestimmte Weise ordnen. Jedes Rechteck  $r_i$  in jeder solchen Liste sortieren wir nach diesen Kriterien: 1) aufsteigend nach der Länge  $s_i$  und 2) aufsteigend nach dem Beginn  $b_i$ . Die Entscheidung, diese Sortierkriterien zu wählen, ergibt sich experimentell und wird im Abschnitt 1.4.2 diskutiert.

Danach muss man die Lücken in jedem Streifen in einer Platzierung finden. So legen wir eine Liste  $H_j$  für jeden Streifen  $j$  fest, in der sich alle Lücken aus diesem Streifen befinden. Man findet sie, indem man durch jedes im Streifen  $j$  platzierte Rechteck  $r_i$  iteriert und jeweils überprüft, ob das Rechteck  $r_i$  mit dem Rechteck  $r_{i+1}$  eine gemeinsame Seite haben. Wenn nicht, gibt es eine Lücke zwischen den Rechtecken  $r_i$  und  $r_{i+1}$ . Dazu muss man auch untersuchen, ob das erste Rechteck und das letzte Rechteck im Streifen direkt an den Seiten des großen Rechtecks liegen. Wenn nicht, entstehen auch Lücken zwischen den Rechtecken und den Seiten des großen Rechtecks  $R$ .

Danach werden alle Listen  $H_j$  zu einer Liste  $H$  zusammengebracht. Diese Liste muss auch auf eine bestimmte Weise geordnet sein. Experimentell ergeben sich die folgenden Sortierkriterien für jede Lücke  $L_i$ : 1) fallend nach der Größe der Lücke  $L_i$  und 2) aufsteigend nach dem Index des Streifens. Diese Entscheidung wird ebenfalls im Abschnitt 1.4.2 diskutiert.

---

**Algorithmus 1** Das heuristische Verbesserungsverfahren
 

---

**Eingabe:**  $R$  — das große Rechteck,  $Z$  — die Liste mit allen kleineren Rechtecken.

```

1:  $C_A \leftarrow \text{Greedy}(R, Z)$ 
2:  $H \leftarrow \text{getAllHoles}(C_A)$ 
3:  $U \leftarrow \text{getRecs}(C_A)$ 
4:  $C \leftarrow C_A$ 
5:  $itH \leftarrow H.begin$ 
6:  $j \leftarrow itH.streifen \quad \triangleright j$  ist der Index des Streifens, in dem sich die Lücke zum Iterator  $itH$  befindet
7:  $itR \leftarrow U_j.begin$ 
8: while  $itH \neq H.end$  do
9:    $\Sigma_C \leftarrow |C| \quad \triangleright$  der Gesamtflächeninhalt aller platzierten Rechtecke in  $C$ 
10:   $C' \leftarrow \text{next}(C, itH, itR)$ 
11:   $\Sigma_{C'} \leftarrow |C'|$ 
12:   $itR \leftarrow itR + 1$ 
13:  if  $itR = U_j.end$  then
14:     $itH \leftarrow itH + 1$ 
15:     $j \leftarrow itH.streifen$ 
16:     $itR \leftarrow U_j.begin$ 
17:  end if
18:  if  $\Sigma_{C'} > \Sigma_C$  then
19:     $H \leftarrow \text{getAllHoles}(C')$ 
20:     $U \leftarrow \text{getRecs}(C')$ 
21:     $itH \leftarrow H.begin$ 
22:     $k \leftarrow itH.streifen$ 
23:     $itR \leftarrow U_k.begin$ 
24:     $C \leftarrow C'$ 
25:  end if
26: end while
```

---

Der Algorithmus 1 zeigt eine vereinfachte Vorgehensweise des Verbesserungsverfahrens. Die Funktion  $\text{getAllHoles}(C)$  findet alle Lücken in allen Streifen im Rechteck  $R$  in der aktuellen Platzierung  $C$  und bestimmt die Liste  $H$ . Die Funktion  $\text{getRecs}(C)$  findet alle nicht platzierten Rechtecke und verteilt sie auf die Listen  $U_j$  für jeden Streifen  $j$ . Die Funktion  $\text{next}(C, itH, itR)$  besteht in diesem Algorithmus selbst

aus drei Funktionen, die im Folgenden beschrieben werden.

Die erste Funktion bestimmt die nächste Lücke, in die ein nicht platziertes Rechteck eingefügt wird. Es gibt einen Iterator  $itH$ , der am Anfang am Beginn der Liste  $H$  gesetzt wird. Im Laufe des Algorithmus bewegt sich der Iterator und zeigt auf nächste Lücken. Eine Lücke bezeichnen wir als *geeignet*, wenn sie sich in einem Streifen  $j$  befindet, zu dem die entsprechende Liste  $U_j$  nicht leer ist, d.h., es gibt mindestens ein Rechteck, das in diese Lücke eingefügt werden kann. Somit bewegt sich der Iterator in der Liste  $H$  und wenn er auf eine geeignete Lücke stößt, wird diese Lücke durch die folgenden zwei Funktionen bearbeitet. Insbesondere beachte man, dass die While-Schleife in Zeile 8 abbricht, wenn der Iterator  $itH$  zum Ende der Liste  $H$  gelangt, also dann, wenn es keine geeigneten Lücken mehr gibt.

Die nächste Funktion wählt ein Rechteck  $r$ , das in eine gewählte geeignete Lücke  $L$  eingefügt wird. Da die gewählte Lücke sich in einem Streifen  $j$  befindet, stammt  $r$  aus der Liste  $U_j$ . Es gibt auch einen internen Iterator  $itR$  für die Liste  $U_j$ , der bei jeder neuen Lücke an den Anfang der Liste gesetzt wird. Wenn ein Rechteck  $r$  in einem Lauf  $t$  der While-Schleife in  $L$  eingefügt wird, wird  $itR$  danach inkrementiert und im darauf folgenden Lauf der Schleife  $t + 1$  wird ein unterschiedliches Rechteck in die Lücke  $L$  gelegt. Wenn der Iterator  $itR$  bis zum Ende der Liste  $U_j$  gelangt, wird der Iterator  $itH$  in der Liste  $H$  inkrementiert und somit eine neue geeignete Lücke gesucht.

Die letzte Funktion nimmt das unter dem Iterator  $itR$  stehende Rechteck  $r$  und legt es in die unter dem Iterator  $itH$  stehende Lücke  $L$ . Diese Funktion bereitet eine neue Platzierung  $C'$  vor. Seien die Koordinaten der Lücke  $x_1$  und  $x_2$  und der Streifen, in dem sich  $L$  befindet, sei  $j$ . Seien  $x_r$  und  $x_r + s_r$  die  $x$ -Koordinaten von  $r$ . Das Rechteck  $r$  wird in  $R$  so gelegt, dass  $x_2 = x_r + s_r$ . Auf diese Weise beträgt der Wert  $x_r := x_2 - s_r$ .<sup>2</sup> Selbstverständlich kann es an dieser Stelle zu Kollisionen kommen — Rechtecke können sich überdecken. Vor dem Platzieren prüft man nicht, ob es durchgehend eine Lücke zwischen den Werten  $x_r$  und  $x_r + s_r$  in allen Streifen  $k$  gibt, wobei  $b_r \leq k < e_r$ . Deshalb entfernt man nun alle Rechtecke, die mit  $r$  kollidieren, also all diejenigen, die zumindest zum Teil zwischen  $x_r$  und  $x_r + s_r$  in einem Streifen  $k$  liegen. So entstehen auch neue Lücken, deshalb versuchen wir an dieser Stelle die Lücken mit anderen Rechtecken zu füllen. Dazu verarbeiten wir alle Streifen  $k$  ( $b_r \leq k < e_r$ ), indem wir jedes in  $C'$  noch nicht gelegte Rechteck  $r_i$  in jeder Liste  $S_k$  (in der ursprünglichen Reihenfolge) untersuchen. Wie beim Greedy-Algorithmus am Anfang versuchen wir, ein Rechteck  $r_i$  im Streifen  $g$  zu legen nur, wenn  $g = b_i$  (und wenn es eine Lücke zwischen  $x_{r_i}$  und  $x_{r_i} + s_i$  gibt). Allerdings müssen wir die Streifen  $b_i + 1, b_i + 2, \dots, e_i - 1$  vor dem Platzieren prüfen, ob es in ihnen durchgehend Lücken zwischen  $x_{r_i}$  und  $x_{r_i} + s_i$  gibt. Nur, wenn in allen Streifen  $b_i, b_i + 1, \dots, e_i - 1$  diese Lücken bestehen, kann das Rechteck  $r_i$  in die Platzierung  $C'$  eingefügt werden. Nachdem alle durch Kollision betroffenen Streifen verarbeitet worden sind, ist die Platzierung  $C'$  fertig.

Dann erfolgt der Vergleich in Zeile 18. Wenn der Gesamtflächeninhalt der Platzierung  $C'$  streng größer ist als der Gesamtflächeninhalt der Platzierung  $C$ , wird die neue Platzierung vom Algorithmus akzeptiert und gilt als die aktuelle Platzierung  $C$ . Danach muss man offensichtlich alle Lücken und alle nicht gelegten Rechtecke neu bestimmen. Die Iteratoren  $itH$  bzw.  $itR$  werden auf  $H.begin$  bzw. auf  $U_k.begin$  gesetzt.

Man kann leicht begründen, dass das Programm auf jeden Fall anhält. Durch den Quasibergsteigeralgorithmus wird ein lokales Maximum gefunden, das bestenfalls auch das Optimum ist. Jedes lokale Maximum kann das Optimum nicht überschreiten. Das Optimum ist auf den Flächeninhalt des großen Rechtecks  $R$  beschränkt. Wenn ein lokales Maximum erreicht wird, kann der Algorithmus keine neuen Platzierungen annehmen. Wenn ein lokales Maximum erreicht wird, werden alle geeigneten Lücken und dazu jeweils alle passenden Rechtecke ausprobiert, aber das Programm nimmt keine der neuen Kombinationen an, da sie keinen größeren Flächeninhalt bilden als der des lokalen Maximums. Somit hält das Programm auf jeden Fall an.

## 1.4 Diskussion der Ergebnisse

### 1.4.1 Grenzen der Heuristik

Im Abschnitt 1.2 wurde bewiesen, dass das FLOHMARKT-PROBLEM NP-schwer ist. Es gibt faktoriell viele möglichen Anordnungen der Rechtecke. Für  $|Z|$  in der Ordnung von ca. 700 wird eine Brute-Force-Lösung deshalb nicht in einer akzeptablen Zeit gelöst. Die Komplexität des FLOHMARKT-PROBLEMS setzt Schranken auf die optimale Lösung des Problems. Man kann keinen in Polynomialzeit laufenden Algorithmus entwickeln, der jede Instanz dieses Problems lösen würde. Das ist der Hauptgrund, warum wir eine Heuristik verwenden. Allerdings, da eine Heuristik nur ein Approximationsalgorithmus ist und nur nahezu optimale Ergebnisse liefert, muss es Kompromisse geben. Dieser Kompromiss betrifft vor allem

<sup>2</sup>Die Situation, in der  $x_r < 0$  gilt, wird in der zweiten Funktion ausgeschlossen. Wenn es dazu käme, wird der Iterator  $itR$  inkrementiert und das nächste Rechteck gewählt.

die Laufzeit und dafür, dass das hier vorgestellte Verfahren in Polynomialzeit läuft, trifft das Programm an vielen Stellen vereinfachte Entscheidungen, die nicht zum optimalen Ergebnis führen können. In diesem Abschnitt diskutieren wir nur über die Grenzen der beiden Greedy-Algorithmen im Programm — über den Greedy-Algorithmus am Anfang und über das heuristische Verbesserungsverfahren — und im Abschnitt 1.4.2 besprechen wir die Ergebnisse.

Im Greedy-Algorithmus am Anfang liegt die Schwierigkeit darin, dass die Platzierung der Rechtecke grundsätzlich von ihrer Reihenfolge in Listen  $S_j$  abhängt. Diese hängt dann von den Sortierkriterien ab. Obwohl dank der gewählten Sortierkriterien optimale oder sehr gute Ergebnisse bei vielen Beispielen herauskommen, ist das nicht der Fall bei allen Beispielen (mehr dazu im Abschnitt 1.4.2). Auf jeden Fall liefert der Greedy-Algorithmus am Anfang kein optimales Ergebnis zum Beispiel 2, weil dieses Ergebnis im Laufe des Verbesserungsverfahrens optimiert wird.

Im Verbesserungsverfahren wurden mehrere Kompromisse zugunsten der Laufzeit gemacht. Vor allem liegt die Schwierigkeit im Verfahren selbst — warum wird ein Verfahren verwendet, das einen Bergsteigeralgorithmus nachahmt und nicht z.B. ein Verfahren mit der simulierten Abkühlung oder ein ganz anderer heuristischer Ansatz? Außerdem liegen die Schwierigkeiten des Verbesserungsverfahrens auch an den Reihenfolgen der Listen  $U_j$  und der Liste  $H$ . Zum Sortieren dieser Listen nutzt man auch festgelegte Sortierkriterien, die nicht zwingend das optimale Ergebnis liefern müssen. Dazu liegt das Problem auch beim Platzieren eines Rechtecks in eine Lücke. Wir entscheiden uns, das Rechteck an die Koordinate  $x_2$  der Lücke zu legen. Man könnte hier eine andere Vorgehensweise anwenden, z. B. man könnte das Rechteck an die Koordinate  $x_1$  orientieren; man könnte auf eine besondere Weise vorgehen, wenn es Rechtecke gibt, die deutlich kleiner sind als eine Lücke; oder man könnte die Rechtecke in darüber und darunter stehenden Streifen in Betracht ziehen, was im Programm nicht gemacht wird. Wie beim Greedy-Algorithmus am Anfang, hängt die Reihenfolge der Rechtecke beim Ausfüllen der Lücken von der Reihenfolge in den Listen  $S_j$  ab. Weiterhin ist die Weise, auf die Lücken bestimmt werden, sehr stark vereinfacht. Man nimmt keinen Bezug darauf, wie breit eine Lücke ist, also wie viele Streifen sie umfasst. Das ist auf jeden Fall eine Schwachstelle des Programms.

### 1.4.2 Qualität der Ergebnisse

Im vorangegangenen Abschnitt werden die Grenzen der Heuristik erkannt. In diesem Abschnitt diskutieren wir die Qualität der erhaltenen Ergebnisse.

Zuerst bestimmen wir die Kriterien, unter denen wir ein Ergebnis auswerten:

- Das Verhältnis vom Gesamtflächeninhalt der ins große Rechteck gelegten Rechtecke zum Gesamtflächeninhalt aller Rechtecke.
- Das Verhältnis vom Gesamtflächeninhalt der ins große Rechteck gelegten Rechtecke zum Flächeninhalt des großen Rechtecks.
- Die praktische Laufzeit des Programms für ein Ergebnis.

Diese Kriterien wurden in Bezug auf die Aufgabenstellung gewählt, „um den Organisatoren des Flohmarkts zu helfen.“ Das erste Kriterium gibt den Veranstaltern den Einblick darin, wie viel sie in Bezug auf die Menge des verfügbaren Geldes verdienen — alle Mieter wollen den Veranstaltern Geld anbieten, aber es hängt von den Organisatoren ab, welche Anmeldungen sie ablehnen und welche annehmen. Das zweite Kriterium gibt den Einblick darin, wie viel Geld die Veranstalter verdienen in Bezug auf den verfügbaren Platz. Die beiden ersten Kriterien liefern natürlich auch die Erkenntnis über die Verluste, die mit der Auswahl an Anmeldungen verbunden sind. Das dritte Kriterium spielt für die Veranstalter eine praktische Rolle. Sehr wenige Personen würde ein Ergebnis interessieren, das vielleicht um ein paar Promille besser ist, aber dessen Bestimmung mehrere Stunden (oder Tage!) dauert. Deshalb wurde auch die Brute-Force-Lösung ausgeschlossen.

Tabelle 1 stellt die Ergebnisse zu den Beispielen von 1 bis 7 aus der BWINF-Webseite dar. In den ersten zwei Spalten befinden sich entsprechend die Sortierkriterien für Liste  $H$  und die Listen  $U_j$ . Die Sortierkriterien werden in der Tabelle 2 erläutert. Die erste Zeile in der Tabelle stellt die Ergebnisse bevor dem Lauf des Verbesserungsverfahrens dar. Die Zahlen zwischen den Spalten „Bsp. 1“ und „Bsp. 7“ sind die Ergebnisse in  $[m \times h]$  (nach der Aufgabenstellung; auch in Euro), die das Programm unter Verwendung von den angegebenen Sortierkriterien liefert. Dazu ist zu beachten, dass der Flächeninhalt des Rechtecks  $R$  in allen abgebildeten Beispielen 10000 beträgt.

Man kann leicht feststellen, dass der Greedy-Algorithmus sehr gute Ergebnisse liefert, die manchmal nur um winzige Prozentpunkte durch den Verbesserungsalgorithmus verbessert werden.



Kriterien $H$	Kriterien $U_j$	Bsp. 1	Bsp. 2	Bsp. 3	Bsp. 4	Bsp. 5	Bsp. 6	Bsp. 7
<i>Vor dem Verbesserungsverfahren</i>		8028	9056	8778	7370	7962	10000	9959
<b>greaterHolesSize</b>	<b>smallerSize</b>	8028	9077	8778	7370	8705	10000	9979
	<b>greaterSize</b>	8028	9077	8778	7370	8705	10000	9973
	<b>smallerArea</b>	8028	9077	8778	7370	8705	10000	9978
	<b>greaterArea</b>	8028	9077	8778	7370	8705	10000	9973
	<b>greaterEnd</b>	8028	9077	8778	7370	8705	10000	9973
<b>smallerHolesSize</b>	<b>smallerSize</b>	8028	9077	8778	7370	8599	10000	9980
	<b>greaterSize</b>	8028	9077	8778	7370	8599	10000	9973
	<b>smallerArea</b>	8028	9077	8778	7370	8599	10000	9980
	<b>greaterArea</b>	8028	9077	8778	7370	8599	10000	9973
	<b>greaterEnd</b>	8028	9077	8778	7370	8599	10000	9991

Tabelle 1: Die Ergebnisse zu den Beispielen 1–7. Die erste Zeile enthält die Ausgangsergebnisse, bevor man den Verbesserungsalgorithmus laufen lässt. Die ersten zwei Spalten stellen die verwendeten Sortierkriterien für die Liste  $H$  und die Listen  $U_j$  dar. Die Spalten von Bsp. 1 bis Bsp. 7 sind die Ergebnisse in  $[m \times h]$  nach dem Lauf des Verbesserungsalgorithmus. Man beachte, dass die Kriterien, die für den Lauf des Programms und die im Abschnitt 1.3.3 beschrieben werden, diejenigen aus der zweiten Zeile sind. (s. Tabelle 2 für die Erklärungen der Sortierkriterien.)

Name	1. Kriterium	2. Kriterium	3. Kriterium
<b>greaterHolesSize</b>	Lückengröße fallend	Streifenindex aufsteigend	
<b>smallerHolesSize</b>	Lückengröße aufsteigend	Streifenindex aufsteigend	
<b>smallerSize</b>	$s_i$ aufsteigend	$b_i$ aufsteigend	
<b>greaterSize</b>	$s_i$ fallend	$b_i$ aufsteigend	
<b>smallerArea</b>	$s_i \times m_i$ aufsteigend	$b_i$ aufsteigend	
<b>greaterArea</b>	$s_i \times m_i$ fallend	$b_i$ aufsteigend	
<b>greaterEnd</b>	$e_i$ fallend	$b_i$ aufsteigend	$s_i \times m_i$ fallend

Tabelle 2: Die ersten zwei Zeilen sind die Sortierkriterien nur für die Liste  $H$  mit Lücken. Die restlichen Zeilen sind die Sortierkriterien für die Listen  $U_j$  für jedes Rechteck  $r_i$ .

Im Beispiel 1 werden alle Rechtecke bereits beim Lauf des Greedy-Algorithmus am Anfang platziert. Wenn man den Gesamtflächeninhalt aller Rechtecke aus diesem Beispiel ausrechnet, kommt man auf die Zahl 8028. Das ist also das optimale Ergebnis. Ebenfalls ist beim Beispiel 6 sehr leicht zu erkennen, dass der Greedy-Algorithmus das optimale Ergebnis liefert, da alle Rechtecke aus dem Beispiel platziert und die Fläche des großen Rechtecks vollständig bedeckt wurden.

Das Beispiel 3 ist ein sehr interessanter Fall. Weder werden alle Rechtecke aus diesem Beispiel platziert (das ist sowieso unmöglich, da der Gesamtflächeninhalt aller Rechtecke 10010 beträgt), noch ist die Fläche des großen Rechtecks völlig bedeckt. Allerdings, wenn wir die Ergebnisse jedes Streifens einzeln betrachten, stellen wir fest, dass der Flohmarkt von 11:00 bis 17:00 vollständig durchgehend ausgebucht ist, also beträgt der Flächeninhalt jedes Streifens von 3 bis 8 (einschließlich) 1000. Das bedeutet, man kann das Ergebnis für diese Streifen nicht verbessern. Man kann auch feststellen, dass alle Rechtecke, die zu den Streifen 0, 1, 2, 9 gehören, platziert wurden. Das bedeutet, dass die restliche Fläche mit keinen Rechtecken bedeckt werden kann. Das bedeutet, dass 8778 das optimale Ergebnis für dieses Beispiel ist, weil man dieses Ergebnis nicht verbessern kann.

Die Situation mit dem Beispiel 4 sieht ähnlich aus. Obwohl nicht alle Rechtecke platziert werden (wieder beträgt der Gesamtflächeninhalt aller Rechtecke mehr als 10000) und es noch viel freien Platz im großen Rechteck gibt (mehr als ein Viertel des Flächeninhalts), ist dieses Ergebnis optimal. Man kann per Hand prüfen, dass jede Kombination mit den zwei nicht gelegten Rechtecken kein besseres Ergebnis ergibt.

Dann bleiben noch die Ergebnisse zu den Beispielen 2, 5 und 7. Da die Anzahl an kleineren Rechtecken zu groß ist, um die Ergebnisse per Hand zu prüfen und wir keinen Brute-Force-Algorithmus laufen lassen, kann man schwer sagen, wie weit die Ergebnisse von den Optima abweichen. Zur Auswertung dieser Ergebnisse verwenden wir unsere festgelegten Kriterien.

Das Ergebnis zum Beispiel 2 vor dem Lauf des Greedy-Algorithmus am Anfang beträgt 9056. Dieser Gesamtflächeninhalt entspricht 90,5% des Flächeninhalts des großen Rechtecks und ebenfalls 90,5% des Gesamtflächeninhalts aller Rechtecke in diesem Beispiel. Man beachte, dass der Gesamtflächeninhalt

aller Rechtecke 10000 überschreitet. Durch den Verbesserungsalgorithmus steigt das Anfangsergebnis auf 9077, also eine Verbesserung um 0,2 Prozentpunkte. In diesem Beispiel gibt es 603 Rechtecke und das sind Anmeldungen, deren Wert  $s_i$  hauptsächlich zwischen 1 und 6 liegt. Die praktische Laufzeit des Algorithmus liegt im Bereich von 20 Sekunden auf einem modernen Rechner.<sup>3</sup> Der Wert 90,7% ist akzeptabel in Bezug auf die möglichen Einkommen und auf den verfügbaren Platz und aus praktischer Sicht ist es schwer, sich ein besseres Ergebnis innerhalb von 20 Sekunden zu wünschen, solange die Ermittlung eines qualitativen Ergebnisses per Hand sehr lange dauern würde. Es ist anhand der Tabelle 1 festzustellen, dass das Ergebnis sich nach dem Lauf des Verbesserungsalgorithmus unter Verwendung von beliebigen Kriterien um denselben Wert verbessert.

Das Anfangsergebnis zum Beispiel 7 ist 9959 und beträgt genau 99,59% des Flächeninhalts des Rechtecks  $R$  und des Gesamtflächeninhalts aller Rechtecke. In diesem Beispiel gibt es 566 Anmeldungen, davon haben die meisten den Wert  $s_i$  im Bereich von 1 bis 6, aber im Vergleich zum Beispiel 2 gibt Rechtecke, die einen Wert  $s_i$  im Bereich 10–40 besitzen. Wenn man die Sortierkriterien `greaterHolesSize` und `smallerSize` wählt, also diejenigen, die für das Programm gewählt wurden und die im Abschnitt 1.3.3 beschrieben werden, liefert das Verbesserungsverfahren ein Ergebnis um 0,2 Prozentpunkte besser. Wir stellen fest, die anderen Kombinationen der Sortierkriterien und insbesondere die Kombination aus der letzten Zeile der Tabelle ein besseres Ergebnis liefert. Das bedeutet, dass der Wert 9979 auf jeden Fall nicht optimal ist. Allgemein ist der Wert 9979 aus praktischer Sicht völlig akzeptabel. Es ist kaum möglich, innerhalb von einer Sekunde so einen Wert zu erreichen, wenn man eine Platzierung der Rechtecke per Hand bestimmt. Aus praktischer Sicht ist der Wert 9991 auch nicht von einem bedeutenden Unterschied zu 9979 und sind die beiden Werte sehr nah am Optimum.

Als letztes bleibt das Beispiel 5, das 25 große Rechtecke umfasst, und zu dem der Greedy-Algorithmus am Anfang das Ergebnis 7962 liefert. Der Gesamtflächeninhalt aller Rechtecke beträgt mehr als 300000 und somit entspricht das Ergebnis 79,6% des Flächeninhalts des Rechtecks  $R$  und 25,7% des Gesamtflächeninhalts aller Rechtecke. Das Verbesserungsverfahren unter Verwendung von den Sortierkriterien `greaterHolesSize` liefert einen Wert 8705, also ist das eine Verbesserung um 7,4% Prozentpunkte. Hingegen beträgt das verbesserte Ergebnis nur 8599, wenn man die Sortierkriterien `smallerHolesSize` verwendet. Es ist schwer zu beurteilen, wie weit das Ergebnis vom Optimum abweicht. Allerdings ist 87% des Flächeninhalts des großen Rechtecks, also des verfügbaren Platzes, noch in Ordnung und so ein Wert ist aus praktischer Sicht wünschenswert. Nebenbei bemerkt liegt dieser Wert unter dem Grundfreibetrag, was bedeutet, dass die Veranstalter eigentlich keine Steuer abführen müssen, wenn sie dieses Jahr keine weiteren Einnahmen planen. Wenn der Wert über 9400 Euro liegen würde, müsste man noch die Steuer bezahlen und die Einkommen wären geringer :-).

Die deutlichste Verbesserung im Beispiel 5 liegt an der Entscheidung, die Sortierkriterien `greaterHolesSize` und `smallerSize` zu wählen. Eventuell könnte man den Algorithmus so entwickeln, dass man ihn 10-mal laufen lässt und in jedem Lauf andere Kriterien wählt. Dann könnte einfach das Maximum aus diesen Ergebnissen gezogen werden. In vielen Heuristiken ist es genau die Idee, verschiedene Methoden zusammen zu verknüpfen. Bei allen Beispielen außer 2 dauert die Laufzeit des ganzen Programms etwa eine Sekunde, was praktisch eine sehr gute Laufzeit darstellt, und die Qualität der Ergebnisse ist in diesem Zusammenhang sehr gut, wenn man dies mit einer faktoriellen Laufzeit eines Brute-Force-Lösung vergleicht. Die durch den Bergsteigeralgorithmus gefundenen Maxima liegen höchstwahrscheinlich sehr nah an den Optima.

Zum Vergleich wurde auch ein Ansatz mit der simulierten Abkühlung und ein Ansatz, in dem die Reihenfolgen der Rechtecke in den Listen zufällig war, ausprobiert, aber die Ergebnisse waren allgemein schlechter oder genauso gut wie die, die der Quasibergsteigeralgorithmus liefert. Die Ergebnisse mit den zufälligen Reihenfolgen werden im Folgenden beschrieben.

Anstatt dass die obengenannten Sortierkriterien verwendet werden, um die Liste  $H$  und die Listen  $U_j$  zu sortieren, kann man die Lücken und die Rechtecke in diesen Listen zufällig ordnen. Dazu wurde der eingebaute Zufallszahlengenerator `random_device` und die Funktion `shuffle` verwendet. In der Tabelle 3 sind die Ergebnisse zu den Beispielen 5 und 7 zu sehen. Wie man feststellen kann, sind die Maxima aus allen Kombinationen nicht größer als die Ergebnisse in der Tabelle 1, aber die Minima sind nicht größer als die Minima der Ergebnisse ohne Zufallszahlengenerator. Dank der Zufälligkeit kann man zum Beispiel 7 ein um 12 besseres oder ein um 11 schlechteres Ergebnis im Vergleich zur Kombination `greaterHolesSize`—`smallerSize` erzielen. Das Ergebnis zum Beispiel 5 kann dank der zufälligen Reihenfolge nicht verbessert werden, aber kann im schlimmsten Fall um 193, also um 1,8 Prozentpunkte, verschlechtert werden. Die mittels des Zufallszahlengenerators generierten Ergebnisse zu den anderen Beispielen sind dieselben wie in der Tabelle 1.

<sup>3</sup>Die genaue Laufzeit ist nicht interessant, ohne dass die technischen Spezifikationen des Rechners angegeben werden.

		Beispiel 5			Beispiel 7		
Kriterien $H$	Kriterien $U_j$	Min.	Max.	Mittelwert	Min.	Max.	Mittelwert
random	smallerSize	8512	8705	8629,383	9968	9991	9978,654
	greaterSize	8599	8705	8685,178	9968	9991	9976,078
	smallerArea	8512	8705	8621,999	9968	9988	9978,53
	greaterArea	8599	8705	8697,474	9968	9991	9975,984
	greaterEnd	8599	8705	8698,322	9969	9991	9976,853
	random	8512	8705	8663,05	9968	9991	9977,131
greaterHolesSize	random	8705	8705	8705	9972	9979	9974,091
smallerHolesSize	random	8599	8599	8599	9973	9991	9981,197

Tabelle 3: Die Ergebnisse zu den Beispielen 5 und 7 unter Verwendung vom eingebauten Zufallszahlengenerator `random_device`. Es wurden 1000 Ergebnisse zu jeder Kombination generiert. In den 3. und 4. bzw. in den 6. und 7. Spalten befinden sich die Minima und Maxima der generierten Ergebnisse zur jeweiligen Kombination. In der 5. bzw. in der 8. Spalte befinden sich die arithmetischen Mittel aus allen generierten Ergebnissen zur jeweiligen Kombination.

## 1.5 Laufzeit

- $M$  — die Breite des großen Rechtecks  $R$ , die Anzahl der Streifen
- $n$  —  $|Z|$ , also die Anzahl der kleineren Rechtecke

Die Größe  $M$  in der Aufgabe tritt in vollständigen Stunden auf. Wenn die Eingabe zu Minuten konvertiert wird, wird diese Variable in vollständigen Minuten betrachtet. Im Abschnitt 1.3.2 wird beschrieben, dass die Breite jedes Streifens 1 Stunde bzw. 1 Minute entspricht. Somit kann man die Größe  $M$  auch als die Anzahl der Streifen betrachten.

- Vorbereitung der Eingabe:  $O(M \cdot n \log n)$  (worst-case)
  - Einlesen aller Rechtecke und Erstellung der Liste  $Z$ :  $O(n)$
  - Erstellung von Listen `placed`, `unusedRectangles` und `holes` für jeden Streifen (s. Umsetzung):  $O(M)$
  - Verteilung jedes Rechtecks auf die Listen  $S_j$ , zu denen sie gehören:  $O(M \cdot n)$  (worst-case)  
Im schlimmsten Fall gehört jedes Rechteck zu jeder Liste  $S_j$ , wenn jede Anmeldung den ganzen Zeitraum eines Flohmarkts betrifft. So muss jedes Rechteck jeder Liste  $S_j$  hinzugefügt werden.
  - Sortierung der Listen  $S_j$ :  $O(M \cdot n \log n)$  (worst-case)  
Es gibt  $M$  Listen und in jeder Liste kann es im schlimmsten Fall alle Rechtecke geben. Die linear-logarithmische Laufzeit ist durch das Sortieren verursacht.
- Der Greedy-Algorithmus am Anfang:  $O(n(n + M \log n))$  (worst-case, amortisierte Laufzeit)  
Obwohl man die Funktion zur Verarbeitung eines Streifens  $M$ -mal laufen lässt, kann eine Laufzeitanalyse pro Lauf dieser Funktion zu pessimistisch sein. Es ist unmöglich, dass ein Platz für  $n$  Rechtecke in jedem Streifen  $M$ -mal gesucht wird, da wir voraussetzen, dass jedes Rechteck  $r_i$  nur im Streifen  $b_i$  gelegt werden kann und außerdem können  $n$  Rechtecke nicht  $M$ -mal platziert werden, da jedes Rechteck nur einmal gelegt werden kann. Stattdessen analysieren wir die amortisierte Laufzeit für das Platzieren jedes Rechtecks allein, deshalb wird die endliche Laufzeit mit dem Faktor  $n$  multipliziert, da man im schlimmsten Fall alle  $n$  Rechtecke in  $R$  legen muss.
  - Das Finden der passenden, freien  $x$ -Koordinaten:  $O(n)$  (worst-case)  
Im schlimmsten Fall muss man in einem Streifen über  $n - 1$  Rechtecke iterieren, um einen freien Platz für ein Rechteck zu finden.
  - Das Finden der genauen Stelle in den restlichen Streifen:  $O(M \log n)$  (worst-case)  
Nicht in jedem Streifen müssen sich dieselben Rechtecke befinden und ein Rechteck kann zu mehreren Streifen gehören. In jedem Streifen muss man die genaue Position zum Platzieren des Rechtecks finden. Da man im Programm `set` verwendet, sind alle Rechtecke im Streifen immer in aufsteigender Reihenfolge ihrer Koordinaten  $x_i$ . So kann ein Rechteck mittels der eingebauten Funktion `insert` in `set` eingefügt werden. Die Einfügen-Operation erfolgt

in logarithmischer Laufzeit bezüglich der Anzahl der Rechtecke im Streifen:  $O(\log n)$ .<sup>4</sup> Im schlimmsten Fall gehört ein Rechteck zu allen Streifen, deshalb muss die endliche Laufzeit mal  $M$  multipliziert werden:  $O(M \log n)$ .

- Ein Lauf des Verbesserungsalgorithmus pro Paar Lücke–Rechteck:  $O(n(n + M \log n))$  (worst-case, amortisierte Laufzeit)
  - Berechnung des Gesamtflächeninhalts aller platzierten Rechtecke:  $O(n)$  (worst-case)  
Im schlimmsten Fall können alle Rechtecke ins große Rechteck  $R$  gelegt werden.
  - Bestimmung aller nicht gelegten Rechtecke:  $O(M \cdot n \log n)$  (worst-case)  
Im schlimmsten Fall gibt es ein gelegtes Rechteck, das genauso groß ist wie  $R$ , und somit alle  $n - 1$  restlichen kleineren Rechtecke zu Listen  $U_j$  gehören. Im schlimmsten Fall können alle diese restlichen Rechtecke zu allen Listen  $U_j$  gehören. Die linear-logarithmische Laufzeit ist dem Sortieren der Listen  $U_j$  geschuldet.
  - Auffinden aller Lücken:  $O(M \cdot n \log n)$  (worst-case)  
Im schlimmsten Fall gibt es in jedem Streifen  $n + 1$  Lücken, wenn kein Paar der Rechtecke in demselben Streifen eine gemeinsame Seite hat — dann gibt es Lücken auf beiden Seiten jedes Rechtecks. Dazu kann es im schlimmsten Fall  $n$  Rechtecke in jedem Streifen geben. Die linear-logarithmische Laufzeit ist mit dem Sortieren der Liste  $H$  verbunden.
  - Entfernung aller kollidierenden Rechtecke:  $O(M \cdot n)$  (worst-case)  
Beim Einfügen eines Rechtecks  $r$  in eine Lücke muss man in allen Streifen, zu denen  $r$  gehört, prüfen, ob es Kollisionen gibt. Im schlimmsten Fall gehört ein Rechteck zu allen  $M$  Streifen und man muss in allen Streifen  $n - 1$  Rechtecke entfernen.
  - Einfügen neuer Rechtecke in neue Lücken:  $O(n(n + M \log n))$  (worst-case, amortisierte Laufzeit)  
In diesem Teil lohnt es sich mehr, eine amortisierte Laufzeitanalyse pro Rechteck durchzuführen. Im schlimmsten Fall muss man  $n - 1$  Rechtecke ins Rechteck  $R$  einfügen, deshalb wird die endliche Laufzeit mit dem Faktor  $n$  multipliziert. Im schlimmsten Fall gehört jedes Rechteck zu jedem der  $M$  Streifen. Beim Einfügen jedes Rechtecks  $r$  muss man zuerst die potenziellen Koordinaten für  $r$  im Streifen  $b_r$  finden. Dazu kann man im schlimmsten Fall über  $n - 1$  Rechtecke iterieren:  $O(n)$ . Dann muss man in jedem Streifen, zu dem  $r$  gehört, prüfen, ob es genug Platz dafür gibt. Das erfolgt mittels der eingebauten Funktion `upper_bound`, die in logarithmischer Zeit bezüglich der Anzahl der Rechtecke im Streifen läuft:  $O(\log n)$ .<sup>5</sup> Dann wird das Rechteck allen Streifen hinzugefügt, zu denen es gehört. Diese Operation erfolgt mithilfe der Funktion `insert`, die in logarithmischer Zeit läuft:  $O(M \log n)$ . Somit ergibt sich pro Rechteck  $r$  die Laufzeit:  $O(n + M \log n)$ .
  - Zurücksetzung der ursprünglichen Platzierung:  $O(M \cdot n)$  (worst-case)  
Wenn eine Platzierung einen niedrigeren Flächeninhalt besitzt als die ursprüngliche Anordnung, wird diese Platzierung vom Algorithmus abgelehnt und die ursprüngliche Anordnung wird zurückgesetzt. Dazu muss man den Inhalt aus allen  $M$  Streifen kopieren, wobei sich in jedem Streifen höchstens  $n$  Rechtecke befinden können.

Wie man im Algorithmus 1 erkennt, unterscheidet man zwischen zwei möglichen Läufen des Verbesserungsalgorithmus in der While-Schleife: Wenn ein neues Ergebnis angenommen wird und falls nicht. Wenn ein Ergebnis angenommen wird, muss man zusätzlich die Funktionen zur Bestimmung aller Lücken und aller nicht gelegten Rechtecke laufen lassen. Wie wir in den Betrachtungen zur Laufzeitanalyse zum Verbesserungsalgorithmus pro ein Paar Lücke–Rechteck sehen, besitzen diese zwei Funktionen niedrigere Laufzeiten als die Gesamtlaufzeit der restlichen Prozesse, deshalb unterscheiden wir nicht zwischen den Laufzeiten der While-Schleife. Somit ergibt sich die folgende Laufzeit für ein Paar von einer Lücke  $L$  und einem Rechteck  $r$ , wobei  $r$  in  $L$  eingefügt werden soll:  $O(n(n + M \log n))$  (worst-case).

Nun bestimmen wir die Anzahl an Paaren Lücke–Rechteck, die vom Verbesserungsalgorithmus verarbeitet werden. Die Funktion zur Bestimmung aller nicht gelegten Rechtecke kann höchstens  $n - 1$  Rechtecke und die Funktion zum Auffinden aller Lücken kann höchstens  $M \cdot n$  Lücken finden. Allerdings, wenn  $n - 1$  Rechtecke nicht gelegt sind, gibt es höchstens  $2M$  Lücken. Wenn die Anzahl der Lücken wächst, sinkt die Anzahl der nicht gelegten Rechtecke. Deshalb gibt es am meisten Möglichkeiten, wenn die beiden Anzahlen halbiert werden. So gibt es maximal  $O(M \cdot n/2 \cdot n/2) = O(M \cdot n^2)$  Möglichkeiten.

<sup>4</sup><https://en.cppreference.com/w/cpp/container/set/insert>

<sup>5</sup>[https://en.cppreference.com/w/cpp/algorithm/upper\\_bound](https://en.cppreference.com/w/cpp/algorithm/upper_bound)

Im schlimmsten Fall muss man alle diesen Kombinationen durchgehen, bis man zu einem Ergebnis gelangt, das vom Verbesserungsalgorithmus angenommen wird. So legen wir fest, die Laufzeit des Verbesserungsalgorithmus für jede Kombination der Lücken und Rechtecke aus einem Paar von Listen  $H$  und  $U_j$  beträgt im worst-case:

$$O(n(n + M \log n) \cdot M \cdot n^2) = O((n^2 + M \cdot n \log n) \cdot M \cdot n^2) = O(M \cdot n^4 + M^2 \cdot n^3 \log n).$$

Es bleibt noch die Abschätzung, wie viel mal man die Listen  $H$  und  $U_j$  bestimmt. Allgemein kann man die Laufzeit eines Bergsteigeralgorithmus schwer abschätzen. Allerdings ist ein Ergebnis auf den Flächeninhalt des großen Rechtecks  $R$  beschränkt und die Einheiten in dieser Aufgabe sind ganzzahlig (vollständige Meter, vollständige Stunden/Minuten). So können wir feststellen, dass die Ergebnisse des Verbesserungsalgorithmus mit jeder neuen Bestimmung der Listen  $H$  und  $U_j$  eine streng monoton wachsende Funktion bilden. Es gibt also eine feste Anzahl an möglichen Verbesserungen. Theoretisch könnte es einen Fall geben, in dem das Anfangsergebnis nach dem Lauf des Greedy-Algorithmus 1 beträgt und in dem dieses Ergebnis mit jeder neuen Bestimmung der Listen  $H$  und  $U_j$  um 1 verbessert wird, aber diese Situation ist aus praktischer Sicht kaum vorstellbar. Deshalb führen wir eine Variable  $k$  ein, deren Wert zwischen 1 und  $N \times M$  liegt, die dafür steht, wie oft die Listen  $H$  und  $U_j$  bestimmt werden müssen. In den Beispielen 1–7 überschreitet  $k$  10 nicht. So beträgt die finale Laufzeit im worst-case:

$$O(k(M \cdot n^4 + M^2 \cdot n^3 \log n)).$$

## Literatur

- [1] T.H. Cormen u. a. *Introduction To Algorithms. Third edition*. Introduction to Algorithms. MIT Press, 2009. ISBN: 9780262533058.
- [2] Marek Cygan u. a. *Parameterized Algorithms*. Springer, 2016.
- [3] Michael R. Garey und David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman und Company, 2009.
- [4] Richard E. Korf. „Optimal Rectangle Packing: Initial Results“. In: *American Association for Artificial Intelligence* (2003).

## 2 Umsetzung

### 2.1 Klasse Rec

Rechtecke werden im Programm als Objekte der Klasse **Rec** dargestellt. Diese Klasse besitzt 5 Attribute. Es gibt: **b**, **e** und **size**, die den Werten  $b_i$ ,  $e_i$  und  $s_i$  eines Rechtecks  $r_i$  entsprechen. Außerdem gibt es die Attribute **x1** und **x2**, die den  $x$ -Koordinaten eines Rechtecks entsprechen. Wenn ein Rechteck gelegt wird, werden die beiden Koordinaten festgelegt und es gilt dann immer:  $x1 + size = x2$ . Wenn ein Rechteck nicht platziert ist, sind die beiden Variablen auf  $-1$  gesetzt.

Außer den standardmäßigen Getter- und Setter-Methoden gibt es auch eine Methode **getArea()**, die den Flächeninhalt des Rechtecks ausgibt, indem  $(e - b) \times m$  ausgerechnet wird.

### 2.2 Klasse Hole

Die Lücken stellt man als Objekte der Klasse **Hole** dar. In dieser Klasse gibt es 3 Attribute: **x1**, **x2** und **stripe**. Sie stellen die Koordinaten  $x_1$  und  $x_2$  einer Lücke dar und **stripe** ist der Index des Streifens, in dem die Lücke auftritt.

### 2.3 Hilfsfunktionen

Es gibt drei Hilfsfunktionen, die die Eingabe verarbeiten und die zu keiner Klasse gehören.

Es gibt eine Funktion **timeToMinutes()**, die zur Umwandlung der Zeitangaben dient. Sie nimmt einen String als Argument, das entweder im Format "H" oder "H:MM" ist. H ist dabei ein beliebiger Integer, der Stunden angibt, und MM ist eine zweistellige Zahl, die Minuten angibt. Wenn die Eingabe Minuten enthält, wird die als String eingegebene Zeit zu Minuten umgewandelt, indem H mal 60 multipliziert wird. Sonst erfolgt keine Konversion und H wird nur zu einem Integer umgewandelt. Ausgegeben wird ein Paar bestehend aus der umgewandelten Zeit und einem Integer, in dem das Eingabeformat kodiert wird: 0 steht für Stunden und 1 für Minuten.

Die Funktion **processInput()** verarbeitet den String mit den Öffnungszeiten des Flohmarkts. Diese Funktion nimmt als Argument einen String, der aus  $2k$  Zahlen besteht. Die Funktion teilt den String in eigenständige Uhrzeiten auf. Jede Uhrzeit wird mithilfe der Funktion **timeToMinutes()** zu Stunden bzw. zu Minuten umgewandelt. Dann wird diese Folge von Zahlen als ein **vector<int>** mit dem Eingabeformat (wie oben) ausgegeben.

Es gibt noch die Funktion **calculateRecess()**. Diese Funktion nimmt als Argument einen **vector** von Integers, in dem die Uhrzeiten gespeichert sind. In diesem **vector** steht jede Zahl mit einem geraden bzw. ungeraden Index für die Öffnungs- bzw. Schließungszeit.<sup>6</sup> Ausgegeben wird ein Integer — die Summe der Unterschiede zwischen jedem  $k$ -ten Schließungs- und jedem  $k+1$ -ten Öffnungszeitpunkt. Diese Summe ist wichtig, weil man die Schließungszeit des Flohmarkts vom Endergebnis abziehen soll, wenn der Zeitraum des Flohmarkts unterbrochen ist.

### 2.4 Klasse Solver

In dieser Klasse wird das vorgestellte Verfahren implementiert.

Die Attribute **N** und **M** entsprechen den Zahlen  $N$  und  $M$  aus dem Abschnitt 1.1. Es gibt das Attribut **START**, das dem Wert  $B$  entspricht. Die Variable **recess** entspricht der Dauer der Schließungszeit des Flohmarkts, wenn sein Zeitraum unterbrochen ist. Der Integer **format** speichert das Zeitformat, das in der Eingabedatei auftritt. Die Kodierung ist dieselbe wie oben.

An dieser Stelle muss man erwähnen, dass kleinere Rechtecke im Programm als Zeiger zu den Objekten von **Rec** dargestellt werden. Diese Entscheidung wurde aus diesem Grund getroffen, damit es nur eine Kopie eines Rechtecks in der Klasse **Solver** gibt und damit jeder Prozess im Programm einen direkten Zugriff auf jedes Rechteck hat.

In der Klasse **Solver** gibt es einen **vector<Rec\*>** namens **rectangles**, der alle Rechtecke aus diesem Beispiel speichert und der der Liste  $Z$  aus dem Abschnitt 1.1 entspricht. Es gibt zwei **vector** von **vector<Rec\*>** mit den Namen **rectangles\_stripes** und **unusedRectangles**. Die beiden haben die Größe  $M$  und der erste entspricht den Listen  $S_j$  und der zweite den Listen  $U_j$ . Dazu gibt es einen **vector** von **set<Rec\*>** der Größe  $M$  namens **placed**, in dem alle Rechtecke auftreten, die in den Streifen platziert sind. Wir verwenden **set<Rec\*>** aus dem Grund, dass die Rechtecke in diesem Container automatisch in der sortierten Reihenfolge der Koordinaten **x2** behalten werden. Außerdem kann man Rechtecke in

---

<sup>6</sup>Indexierung ab 0.

**set** sehr einfach einfügen. Es gibt noch einen **vector** von **vector<Hole>** der Größe **M** namens **holes**, in dem alle Lücken aus jedem Streifen gespeichert werden. Dazu gibt es noch einen **vector** mit dem Namen **all\_holes**, in dem alle Lücken aus allen Streifen hinzugefügt werden.

Die Methode **readFile()** liest die Daten aus der Textdatei ein. Diese Methode nimmt als Argument einen String mit der Adresse der Datei in dem Rechner. Das Eingabeformat in den Textdateien mit den Beispielen wurde geändert. Die erste Zeile beinhaltet die Zahl **N**. Die zweite Zeile enthält eine Folge von Öffnungszeiten des Flohmarkts. Die Uhrzeit mit einem geraden Index ist jeweils der Beginn und die Uhrzeit mit einem ungeraden Index ist jeweils die Schließungszeit. Die dritte Zeile enthält die Anzahl der Anmeldungen **n** und es folgen **n** Zeilen mit den Anmeldungen. So wird zuerst die Zahl **N** eingelesen. Danach folgt ein String mit einer geraden Anzahl an Öffnungs- und Schließungszeiten des Flohmarkts. Dies wird mittels der Funktion **processInput()** eingelesen. Dann wird die gesamte Pausenzeit des Flohmarkts mithilfe der Funktion **calculateRecess()** ermittelt und in der Variable **recess** gespeichert. Dann werden die Variablen **M** und **START** bestimmt. Es folgen danach die **n** Anmeldungen, die jeweils mittels der Funktion **timeToMinutes()** verarbeitet und in **rectangles** gespeichert werden.

Im Programm gibt es die Methode **run()**, die den Lauf des Programms ausführt. In dieser Funktion wird die Methode **distributeToStripes()** ausgeführt, die alle Rechtecke aus **rectangles** auf die **vector rectangles\_stripes** verteilt und die diese Listen danach unter Verwendung von den Sortierkriterien **greaterEnd** einzeln sortiert. Danach folgt der Lauf der Methode **processStripe()** für jeden Streifen. Hier werden die Streifen im Greedy-Algorithmus verarbeitet. Es wird durch jedes Rechteck **r** in jedem **vector rectangles\_stripes** iteriert. Es wird geprüft, ob die Koordinaten des Rechtecks **r** **-1** betragen und ob das Rechteck in dem verarbeiteten Streifen beginnt. Dann wird für **r** nach einer Lücke im Streifen in **placed** gesucht. Die Funktion **findNearestHole()** gibt die Koordinate **x1** für **r** aus, wenn es Platz für dieses Rechteck gibt. Sonst wird **-1** ausgegeben. Wenn **x1** ausgegeben wird, wird dann auch **x2** gesetzt und das Rechteck **r** wird in alle Streifen in **placed** eingefügt, zu denen es gehört.

Nachdem die Methode **processStripe()** für alle Streifen ausgeführt worden ist, wird der Gesamtflächeninhalt aller platzierten Rechtecke mittels der Methode **calculateAreaUsed()** bestimmt und es wird geprüft, ob die Koordinaten aller Rechtecke in **rectangles** größer sind als **-1**. Wenn entweder der Gesamtflächeninhalt gleich dem Produkt  $N \times M$  ist oder wenn alle Rechtecke platziert wurden, wird das Programm an dieser Stelle abgebrochen. Wenn nicht, wird die Methode **runOptimization()** laufen gelassen, die das Verbesserungsverfahren ausführt.

In der Methode **runOptimization()** gibt es eine Variable **area**, in der der Gesamtflächeninhalt einer Platzierung gespeichert wird, und eine boolsche Variable **result**, die dafür steht, ob eine neue Platzierung vom Algorithmus angenommen wird. Am Anfang hat **result** den Wert **true**, da das Ausgangsergebnis angenommen werden muss. Außerdem gibt es ein Objekt **hole** der Klasse **Hole**, das der aktuellen Lücke entspricht, in die ein nicht gelegtes Rechteck eingefügt wird. Dazu gibt es ein Paar aus **Rec\*** und **pair<int, int>** namens **rep**. In dieser Variable wird ein Rechteck zum Einfügen in die Lücke **hole** gespeichert. **pair<int, int>** enthält die Koordinaten der Lücke, in die **Rec\*** eingefügt werden soll. Weiterhin gibt es zwei Iteratoren **itH** und **itR** wie im Algorithmus 1.

Dann beginnt die Do-While-Schleife, die solange läuft, bis es eine Lücke gibt, in die ein Rechteck eingefügt werden könnte. In der Schleife gibt es eine Fallunterscheidung: Wenn **result** den Wert **true** hat und wenn nicht. Wenn die ursprüngliche Platzierung vom Algorithmus angenommen wurde, werden zuerst der Wert **area** mittels der Funktion **calculateAreaUsed()**, die Listen **unusedRectangles** mittels der Methode **determineUnused()** und die Liste **all\_holes** mittels der Funktion **findHoles()** bestimmt. Da wir die Lücken und die nicht gelegten Rechtecke gerade bestimmten, werden **itH** und **itR** an die Anfänge der Liste **all\_holes** und der Liste **unusedRectangles** im Streifen, zu dem die Lücke unter dem Iterator **itH** gehört, gesetzt. Dann wird die nächste Lücke **hole** mittels der Methode **findNextHole()** bestimmt und das nächste Rechteck **rep** mittels der Methode **findReplacement()**. Bevor man **rep** in **hole** einfügt, muss man prüfen, ob die Ausgabe der beiden genannten Funktionen richtig ist. Es kann beispielsweise sein, dass **hole** in so einem Streifen liegt, in dem alle Rechtecke platziert wurden und somit kann man kein Rechteck in **hole** einfügen. Dazu ist die folgende While-Schleife. Wenn mit dem Rechteck oder mit der Lücke etwas falsch ist, wird in **pair<int, int>** in **rep** ein negativer Wert ausgegeben. Der Wert **-1** steht dafür, dass es keine Lücken mehr gibt, in die ein Rechteck eingefügt werden kann, also wird an dieser Stelle die Do-While-Schleife abgebrochen. Der Wert **-2** steht dafür, dass der Iterator ans Ende von **unusedRectangles** in diesem Streifen gelangt ist und dass der Iterator **itH** inkrementiert werden muss und eine neue Lücke gefunden werden muss. Der Wert **-3** bedeutet, dass das Rechteck in **hole** nicht eingefügt werden kann, da die Koordinate **x2** der Lücke kleiner ist als die Länge des Rechtecks **size**. So würde das Rechteck über die Grenzen des großen Rechtecks hinausreichen, wenn es in diese Lücke platziert würde. Diese While-Schleife wiederholt sich solange, bis eine Lücke gefunden wird, in

die man ein Rechteck platzieren kann. Dann wird die Methode `removeCollisions` ausgeführt, die die mit dem eingefügten Rechteck kollidierenden Rechtecke aus der Platzierung entfernt und in der eine neue Platzierung mit neuen Rechtecken generiert wird. Diese Methode gibt einen boolschen Wert aus, der besagt, ob die neue Platzierung besser ist als die vorherige. Danach wiederholt sich die Do-While-Schleife.

Der Fall in der Do-While-Schleife für den Wert `result = false` ist analog zum ersten Fall, mit der Ausnahme, dass die Funktionen `calculateAreaUsed`, `determineUnused()` und `findHoles()` nicht ausgeführt werden.

Die Methode `findHoles()` findet alle Lücken in allen Listen `placed`. Sie nimmt als Argument den Index des Streifens `p`. Wenn `p = -1`, wird die Funktion auf allen Streifen ausgeführt. Es wird über jedes Rechteck im jedem Streifen iteriert und es wird geprüft, ob der Wert `x2` eines Rechtecks mit dem Wert `x1` des darauffolgenden Rechtecks übereinstimmt und ob das erste Rechteck im Streifen den Wert `x1 = 0` und das letzte Rechteck im Streifen den Wert `x2 = N` besitzen. Wenn nicht, wird eine Lücke mit den Koordinaten, die den Koordinaten der genannten Rechtecke entsprechen, und der Nummer des Streifens als ein Objekt der Klasse `Hole` in die Liste `holes` zu diesem Streifen hinzugefügt. Am Ende werden die Listen `holes` aus jedem Streifen zu einer gemeinsamen Liste `all_holes` zusammengebracht und nach den Sortierkriterien `greaterHolesSize` sortiert.

Die Methode `findNextHole()` nimmt als Argument den Iterator `itH`. In der Liste `all_holes` wird geprüft, ob sich unter dem Iterator eine Lücke befindet, zu der es ein Rechteck im Streifen, in dem sich die Lücke befindet, in der Liste `unusedRectangles` gibt. Wenn nicht, wird `itH` inkrementiert, bis so eine Lücke gefunden wird. Wenn der Iterator zum Ende der Liste `all_holes` gelangt, wird eine Lücke mit Koordinaten `-1` als Objekt der Klasse `Hole` ausgegeben, was bedeutet, dass die Do-While-Schleife abgebrochen werden soll.

Die Methode `findReplacement()` nimmt zwei Argumente: ein Objekt `hole` der Klasse `Hole`, also eine Lücke, die von der Funktion `findNextHole()` ausgegeben wurde, und den Iterator `itR`. In dieser Methode wird geprüft, ob `hole` den Wert `-1` hat. Wenn ja, wird ebenfalls der Wert `-1` ausgegeben. Danach wird geprüft, ob der Iterator `itR` das Ende der Liste `unusedRectangles` in dem in `hole` angegebenen Streifen erreichte. Wenn ja, wird der Wert `-2` ausgegeben. Ansonsten wird eine Variable `rep` erstellt, die auf das Rechteck unter dem Iterator `itR` in `unusedRectangles` gesetzt wird. Anschließend wird geprüft, ob die Koordinate `x2` der Lücke `hole` nicht kleiner ist als die Größe `size` von `rep`. Wenn doch, wird `-3` ausgegeben. Sonst wird `rep` mit den Koordinaten der Lücke `x2` ausgegeben.

In der Methode `removeCollisions()` wird ein Paar bestehend aus einem Rechteck und den Koordinaten einer Lücke als Argument genommen. Es wird eine Kopie des ganzen `vector placed` gemacht und `placedOld` genannt. Das Rechteck, das in die Lücke eingefügt wird, nennen wir `rr` und die Lücke `hole`. Dann werden die Koordinaten von `rr` gesetzt: `rr->x2 = hole.second` und `rr->x1 = rr->x2 - getSize`. Danach werden in `oldPlaced` alle Rechtecke gesucht, die mit `rr` kollidieren, also wird geprüft, ob es Rechtecke gibt, deren Wert `x1` kleiner ist als der Wert `x2` des Rechtecks `rr`, wobei `rr` vor einem solchen Rechteck steht. Alle kollidierenden Rechtecke werden einer Liste `collisions` hinzugefügt. Danach werden die Werte `x1` und `x2` aller Rechtecke, die zu `collisions` gehören, auf `-1` gesetzt. Als Nächstes werden alle Rechtecke, deren Koordinaten `-1` betragen, aus `placed` entfernt. Danach wird die Funktion `addNew` ausgeführt, die das Rechteck `rr` und ggf. die neuen anderen Rechtecke in `placed` einfügt. Sie gibt eine Liste mit Rechtecken aus, die neu platziert werden. Sie heißt `added`. Danach wird der Gesamtflächeninhalt mittels der Funktion `calculateAreaUsed()` bestimmt. Wenn der neue Gesamtflächeninhalt größer ist als der vorherige, wird die Methode `removeCollisions()` mit dem ausgegebenen Wert `true` abgebrochen. Falls nicht, werden die Koordinaten aller neu eingefügten Rechtecke aus der Liste `added` auf `-1` gesetzt und die ursprünglichen Koordinaten der kollidierenden Rechtecke werden zurückgesetzt. `placed` wird zu `oldPlaced` gesetzt, die Funktion gibt den Wert `false` aus und bricht ab.

Die Methode `addNew()` nimmt als Argument das Rechteck zum Einfügen in eine Lücke. Es wird in dieser Funktion in `placed` an der richtigen Stelle platziert. Dann lässt man die Funktion `processStripeReturn()`, die probiert, neue Rechtecke in `placed` zu platzieren, für jeden Streifen laufen. Die Methode `placed` funktioniert ähnlich wie die Methode `processStripe`. Es wird für jeden Streifen die Liste `rectangles_stripes` iteriert und bei jedem iterierten Rechteck `r` wird geprüft, ob es mit dem verarbeiteten Streifen beginnt und ob es nicht platziert wurde. Dann wird mittels der Funktion `findNearestHole()` die nächste Lücke in diesem Streifen in `placed` gesucht. Wenn es so eine Lücke gibt, wird geprüft, ob die Lücken in allen Streifen in `placed` bestehen, zu denen das iterierte Rechteck gehört. Wenn ja, wird dieses Rechteck in alle diese Streifen in `placed` eingefügt. Diese Methode gibt eine Liste `added` aus, in der sich alle neu eingefügten Rechtecke befinden. In der Methode `addNew` wird die Liste `added` aus allen Streifen zusammengebracht und ausgegeben.



### 3 Beispiele

Das Format der Eingabe wurde geändert. Die erste Zeile beinhaltet die Zahl  $N$ . Die zweite Zeile enthält eine Folge von Öffnungszeiten des Flohmarkts. Die Uhrzeit mit einem geradem Index ist jeweils der Beginn und die Uhrzeit mit einem ungeraden Index ist jeweils das Ende.<sup>7</sup> Die dritte Zeile enthält die Anzahl der Anmeldungen  $n$  und es folgen  $n$  Zeilen mit den Anmeldungen.

Um das Zeitformat zu ändern, müssen die Zahlen im Format  $H:MM$  angegeben werden, wobei  $H$  eine beliebige natürliche Zahl (mit 0) sein kann und die Minuten  $MM$  immer als eine zweistellige Zahl dargestellt werden müssen.

Der Übersichtlichkeit halber befinden sich die genauen Standorte zu Anmeldungen zu jedem Beispiel in den angehängten csv-Dateien. In den Tabellen entsprechen die Werte  $x_1$  und  $x_2$  den Werten  $x_r$  und  $x_r + s_r$  jedes Rechtecks  $r$ . Dazu bedeuten die Wert der Koordinaten von  $-1$ , dass ein Rechteck nicht platziert ist.

#### 3.1 Beispiel 1

Der Flächeninhalt des großen Rechtecks: 10000 [m · h]

Der Gesamtflächeninhalt aller platzierten Rechtecke: 8028 [m · h]

Der Gesamtflächeninhalt aller Rechtecke: 8028 [m · h]

#### 3.2 Beispiel 2

Der Flächeninhalt des großen Rechtecks: 10000 [m · h]

Der Gesamtflächeninhalt aller platzierten Rechtecke: 9077 [m · h]

Der Gesamtflächeninhalt aller Rechtecke: 10002 [m · h]

#### 3.3 Beispiel 3

Der Flächeninhalt des großen Rechtecks: 10000 [m · h]

Der Gesamtflächeninhalt aller platzierten Rechtecke: 8778 [m · h]

Der Gesamtflächeninhalt aller Rechtecke: 10010 [m · h]

#### 3.4 Beispiel 4

Der Flächeninhalt des großen Rechtecks: 10000 [m · h]

Der Gesamtflächeninhalt aller platzierten Rechtecke: 7370 [m · h]

Der Gesamtflächeninhalt aller Rechtecke: 10534 [m · h]

#### 3.5 Beispiel 5

Der Flächeninhalt des großen Rechtecks: 10000 [m · h]

Der Gesamtflächeninhalt aller platzierten Rechtecke: 8705 [m · h]

Der Gesamtflächeninhalt aller Rechtecke: 30940 [m · h]

#### 3.6 Beispiel 6

Der Flächeninhalt des großen Rechtecks: 10000 [m · h]

Der Gesamtflächeninhalt aller platzierten Rechtecke: 10000 [m · h]

Der Gesamtflächeninhalt aller Rechtecke: 10000 [m · h]

---

<sup>7</sup>Indexierung beginnt mit 0.

### 3.7 Beispiel 7

Der Flächeninhalt des großen Rechtecks:  $10000 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller platzierten Rechtecke:  $9979 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller Rechtecke:  $10000 \text{ [m} \cdot \text{h]}$

### 3.8 Beispiel 8

Besonderheit: Die Länge  $N$  und der Zeitraum von  $B$  bis  $E$  des Flohmarkts sind verschieden von der Aufgabenstellung.

Der Flächeninhalt des großen Rechtecks:  $4776 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller platzierten Rechtecke:  $4427 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller Rechtecke:  $17228 \text{ [m} \cdot \text{h]}$

### 3.9 Beispiel 9

Besonderheit: Der Flohmarkt dauert zwei Tage: Jeweils von 8:00 bis 18:00. Der Zeitraum des Flohmarkts ist dementsprechend unterbrochen.

Der Flächeninhalt des großen Rechtecks:  $7600 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller platzierten Rechtecke:  $7591 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller Rechtecke:  $10000 \text{ [m} \cdot \text{h]}$

### 3.10 Beispiel 10

Besonderheit: Das Zeitformat ist in Minuten.

Der Flächeninhalt des großen Rechtecks:  $95274 \text{ [m} \cdot \text{min]}$

Der Gesamtflächeninhalt aller platzierten Rechtecke:  $61796 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller Rechtecke:  $68611 \text{ [m} \cdot \text{h]}$

## 4 Quellcode

```

1 //der Lauf des Programms
void Solver::run(){
3 //die Rechtecke werden auf die Streifen verteilt,
  // durch die sie verlaufen
5 distributeToStripes();

7 //jeder Streifen wird verarbeitet, indem der erste
  // Greedy-Algorithmus angewendet wird
9 for (int i = 0; i < M; i++)
    processStripe(i);
11

  //Indikator dafuer, ob alle Rechtecke platziert wurden
13 bool all = true;
  for (auto r: rectangles)
15     if (r->x1 == -1)
        all = false;
17

  //der Gesamtflaecheninhalt der platzierten Rechtecke wird berechnet
19 int area = calculateAreaUsed();
  if (all || getM()*getN() == area)
21     return;

23 //wenn nicht alle Rechtecke platziert wurden
  // oder das grosse Rechteck nicht vollstaendig mit Rechtecken
25 // bedeckt ist, laesst man das heuristische Verbesserungsverfahren laufen
  runOptimization();
27 }

29 //der Lauf des heuristischen Verbesserungsverfahrens
void Solver::runOptimization(){
31 //der Gesamtflaecheninhalt aller platzierten Rechtecke
  int area;

33

  //der Indikator dafuer, ob eine Platzierung angenommen wurde
35 bool result = true;

37 //die Luecke
  Hole hole;

39

  //das Rechteck mit den Koordnaten der Luecke, in die
  // es eingefuegt werden soll
41 pair<Rec*, iPair> rep;

43

  //die Iteratoren zur Liste unusedRectangles und zur
  // Liste all_holes
45 int itR = 0, itH = 0;
  do {
47     //falls die vorherige Platzierung vom Algorithmus
    // angenommen wurde
49     if (result){
        //der Gesamtflaecheninhalt aller platzierten Rechteck
        // wird bestimmt
51         area = calculateAreaUsed();

53

        //alle nicht gelegten Rechtecke werden bestimmt
        determineUnused();
55         //alle Luecken werden bestimmt
        findHoles();

57

        //eine Luecke wird gewaehlt
61         hole = findNextHole(itH = 0);
        //ein Rechteck wird anhand der Luecke gewaehlt
63         rep = findReplacement(hole, itR = 0);

65

        //wenn es Probleme mit dem Rechteck oder mit
        // der Luecke gibt
67         while (rep.second.second < 0){
            //es gibt keine Luecken mehr
69             if (rep.second.second == -1)
                return;
71

```

```

73     //itR ist ans Ende der Liste unusedRectangles gelangt
    if (rep.second.second == -2){
75         hole = findNextHole(++itH);
        rep = findReplacement(hole, itR = 0);
77     }

79     //das Rechteck kann nicht in diese Luecke
    // eingefuegt werden
    if (rep.second.second == -3)
81         rep = findReplacement(hole, ++itR);
    }

83     //das Rechteck wird in die gewaehlte Luecke platziert,
    // Kollisionen werden behoben, neue Rechtecke werden
85     // eingefuegt, eine Platzierung wird angenommen oder abgelehnt
    result = removeCollisions(area, rep);
87 }
else {
89     //ein Rechteck wird anhand der Luecke gewaehlt
    rep = findReplacement(hole, ++itR);
91     //wenn es Probleme mit dem Rechteck oder mit
    // der Luecke gibt
93     while (rep.second.second < 0){
        //es gibt keine Luecken mehr
95         if (rep.second.second == -1)
            return;
97     }

99     //itR ist ans Ende der Liste unusedRectangles gelangt
    if (rep.second.second == -2){
        hole = findNextHole(++itH);
101        rep = findReplacement(hole, itR = 0);
    }

103
105    //das Rechteck kann nicht in diese Luecke
    // eingefuegt werden
    if (rep.second.second == -3)
107        rep = findReplacement(hole, ++itR);
    }

109    //das Rechteck wird in die gewaehlte Luecke platziert,
    // Kollisionen werden behoben, neue Rechtecke werden
111    // eingefuegt, eine Platzierung wird angenommen oder abgelehnt
    result = removeCollisions(area, rep);
113 }

115 } while(rep.second.second > -1);

117 return;
119 }

121 //diese Methode verteilt jedes Rechteck auf die Streifen,
    // ueber die es verlaeuft
122 void Solver::distributeToStripes(){
123     //die Listen mit den Listen S_j
    rectangles_stripes = vector<vector<Rec*>> (M);
125
127     //jedes Rechteck wird zu jeder Liste rectangles_stripes
    // eingefuegt, zu der es gehoert
    for (auto r: rectangles) {
129         for (int i = r->getBegin(); i < r->getEnd(); i++)
            rectangles_stripes[i].pb(r);
131     }

133     //jede der Lsite wird sortiert
    for (int i = 0; i < M; i++)
135        sort(rectangles_stripes[i].begin(), rectangles_stripes[i].end(), greaterEnd);
    }

137
139 //diese Methode findet alle nicht platzierten Rechtecke in einem
    // Streifen p; p = -1 steht fuer alle Streifen
    void Solver::determineUnused(int p){
141         if (p == -1){
            for (int i = 0; i < M; i++)
143                unusedRectangles[i].clear();
            for (int i = 0; i < M; i++)

```

```

145     determineUnused(i);

147     //der Zufallszahlengenerator
    mt19937 g(rd());

149     //jede der Liste wird sortiert
151     for (int i = 0; i < M; i++)
        sort(unusedRectangles[i].begin(), unusedRectangles[i].end(), smallerSize);
153     //shuffle(unusedRectangles[i].begin(), unusedRectangles[i].end(), smallerSize);
    }
155     else {
        //in jedem Streifen wird jedes Rechteck
        //geprueft, ob es nicht platziert wurde
157         for (auto r: rectangles_stripes[p])
159             if (r->x1 == -1)
                unusedRectangles[p].pb(r);
161     }
    }

163     //diese Methode findet alle Luecken im Streifen p;
165     // p = -1 steht fuer alle Streifen
    void Solver::findHoles(int p){
167         if (p == -1){
            //alle urspruenglichen Luecken werden entfernt
169             for (int i = 0; i < M; i++)
                holes[i].clear();
171             all_holes.clear();

173             for (int i = 0; i < M; i++)
                findHoles(i);

175             //der Zufallszahlengenerator
177             mt19937 g(rd());

179             //die Luecken werden sortiert
            sort(all_holes.begin(), all_holes.end(), greaterHolesSize);
181             //shuffle(all_holes.begin(), all_holes.end(), g);
        }
183         else {
            //es wird geprueft, ob es Luecken zwischen
            //zwei nebeneinander stehenden Rechtecken gibt

187             //1. Rechteck
            auto it = placed[p].begin();
189             for (; it != placed[p].end(); it++){
                //2. Rechteck
                auto it2 = it;
                it2++;

193                //Gibt es eine Luecke zwischen der Seite des grossen Rechtecks
                //und dem ersten Rechteck im Streifen?
                if (it == placed[p].begin() && (*it)->x1 > 0)
197                    holes[p].emplace_back(0, (*it)->x1, p);

199                //Gibt es eine Luecke zwischen der Seite des grossen Rechtecks
                //und dem letzten Rechteck im Streifen?
                if (it2 == placed[p].end()){
                    if ((*it)->x2 < N)
203                        holes[p].emplace_back((*it)->x2, N, p);
                }

205                //Gibt es eine Luecke zwischen den Rechtecken it und it2?
                else if ((*it)->x2 < (*it2)->x1)
207                    holes[p].emplace_back((*it)->x2, (*it2)->x1, p);
            }

209            //alle Luecken werden in all_holes hinzugefuegt
            for (auto h: holes[p])
211                all_holes.pb(h);
        }
    }

213 }

215 //diese Methode findet die naechste leere Luecke im Streifen p
    // fuer ein Rechteck r
217 int Solver::findNearestHole(Rec* r, int p){

```

```

    auto it = placed[p].begin();
219
    //falls es in einem Streifen p noch keine Rechtecke gibt,
221    // kann r am Anfang platziert werden
    if (placed[p].empty())
223        return 0;

225    //falls es eine genug grosse Luecke am Anfang des Streifens gibt,
    // kann r dort platziert werden
227    if ((*it)->x1 >= r->getSize())
        return 0;

229
    //es wird ueber die Rechtecke im Streifen p iteriert
231    for (; it != placed[p].end(); it++){
        auto it2 = it;
233        it2++;

235        //wenn it das letzte Rechteck im Streifen ist
        if (it2 == placed[p].end()) {
237            //es wird geprueft, ob es eine genug Luecke zur Wand
            // des grossen Rechtecks gibt
239            if (N - (*it)->x2 >= r->getSize())
                return (*it)->x2;
241            //falls es unmoeglich ist, das Rechteck zu platzieren,
            // wird -1 ausgegeben
243            else
                return -1;
245        }

247        //falls es eine genug grosse Luecke zwischen zwei Rechtecken
        // im Streifen gibt
249        if ((*it2)->x1 - (*it)->x2 >= r->getSize()){
            return (*it)->x2;
251        }
    }

253
    //falls es unmoeglich ist, das Rechteck zu platzieren,
255    // wird -1 ausgegeben
    return -1;
257 }

259 //diese Methode verarbeitet den Streifen p beim Lauf
    // des Greedy-Algorithmus am Anfang
261 void Solver::processStripe(int p){
    //es wird durch die Rechtecke im Streifen p iteriert
263    for(auto r: rectangles_stripes[p]){
        //falls das Rechteck bereits platziert wurde oder
265        // nicht mit dem Streifen beginnt,
        // wird zum naechsten Rechteck uebergegangen
267        if (r->x1 > -1 || r->getBegin() < p)
            continue;

269
        //falls es noch keine Rechtecke im Streifen gibt,
        // wird das Rechteck ans Anfang platziert
271        if (placed[p].empty())
            r->x1 = 0;
273        else {
275            //es wird die naechste Luecke im Streifen p fuer
            // dieses Rechteck gesucht
277            int curr = findNearestHole(r, p);

279            //falls so eine Luecke gefunden wurde, wird
            // die x1 Koordinate des Rechtecks r gesetzt
281            if (curr > -1)
                r->x1 = curr;
283            //falls so eine Luecke nicht gefunden wurde,
            // wird zum naechsten Rechteck uebergegangen
285            else
                continue;
287        }

289        //die Koordinate x2 zum Rechteck r wird gesetzt
        r->x2 = r->x1 + r->getSize();

```

```

291     //in alle Streifen, zu denen r gehoert, wird
293     // r an der richtigen Stelle eingefuegt
    for (int i = r->getBegin(); i < r->getEnd(); i++)
295         placed[i].insert(r);
    }
297 }

299 //diese Methode verarbeitet den Streifen p beim Lauf
    // des Verbesserungsalgorithmus
301 vector<Rec*> Solver::processStripeReturn(int p){
    //ein vector mit allen neu eingefuegten Rechtecken
303     vector<Rec*> added;

305     //es wird durch die Rechtecke im Streifen p iteriert
    for(auto r: rectangles_strips[p]){
307         //falls das Rechteck bereits platziert wurde oder
        // nicht mit dem Streifen beginnt,
309         // wird zum naechsten Rechteck uebergangen
        if (r->x1 > -1 || r->getBegin() < p)
311             continue;

313         //die naechste Luecke wird im Streifen p fuer
        // dieses Rechteck gesucht
315         int curr = findNearestHole(r, p);

317         //das iterierte Rechteck wird kopiert
        Rec rr(r->getSize(), r->getBegin(), r->getEnd());
319         rr.x1 = curr;
        rr.x2 = curr + rr.getSize();
321         Rec *rr_p = &rr;

323         //falls so eine Luecke gefunden wurde, wird geprueft, ob es genug
        // Platz in anderen Streifen gibt
325         if (curr > -1){
            for (int i = r->getBegin()+1; i < r->getEnd(); i++){
327                 //die potenzielle Stelle fuer das Rechteck rr wird im Streifen
                // gefunden
329                 auto it = upper_bound(placed[i].begin(), placed[i].end(), rr_p, smallerx2);

331                 //die Koordinate x1 des Rechtecks unter dem Iterator it
                int curr_x1;
333                 //die Koordinate x2 des davor stehenden Rechtecks
                int prev_x2;

335
337                 //falls it sich am Ende der Liste befindet,
                // wird curr_x1 zu N gesetzt
                if (it != placed[i].end())
339                     curr_x1 = (*it)->x1;
                else
341                     curr_x1 = N;

343                 //falls it sich am Anfang der Liste befindet,
                // wird prev_x2 zu 0 gesetzt
                if (it != placed[i].begin())
345                     prev_x2 = (*(--it))->x2;
                else
347                     prev_x2 = 0;

349
351                 //es wird geprueft, ob das Rechteck platziert
                //werden kann
                if (curr_x1 - prev_x2 < rr.getSize() ||
353                     curr_x1 < curr + rr.getSize()
                    || prev_x2 > rr.x1) {
355                     goto next;
                }
            }
357        }
        //die Koordinate x1 wird dem Rechteck r gesetzt
359        r->x1 = curr;
    }
    else {
361        next:
363        continue;
    }

```

```

    }
365
    //das Rechteck wird in alle Streifen eingefuegt,
367    // zu denen es gehoert
    r->x2 = r->x1 + r->getSize();
369    for (int i = r->getBegin(); i < r->getEnd(); i++)
        placed[i].insert(r);
371    added.pb(r);
    }
373    return added;
}

375
//diese Methode findet die naechste groesste Luecke im ganzen
377 // grossen Rechteck; itH ist ein Iterator fuer die Liste all_holes
Hole Solver::findNextHole(int itH){
379     for (; itH < int(all_holes.size()); itH++)
        //es wird geprueft, ob die Luecke nicht in einem Streifen
        // liegt, in dem alle Rechtecke platziert sind
381         if (!unusedRectangles[all_holes[itH].stripe].empty()){
383             return all_holes[itH];
        }
385

387     //falls es keine Luecken mehr gibt
    Hole h(-1, -1);
389     return h;
}

391
//diese Methode findet ein noch nicht platziertes Rechteck
393 // fuer einer Luecke hole; itR ist ein Iterator fuer unusedRectangles
pair<Rec*, iPair> Solver::findReplacement(Hole hole, int itR){
395     //der Streifen, zu dem die Luecke gehoert
    int stripe = hole.stripe;
397

    Rec *rep;
399    Rec a(-1,-1,-1);
    Rec *a_p = &a;
401

    //falls es keine Luecken mehr gibt
403    if (stripe == -1)
        return {a_p, {-1, -1}};
405

    //falls es keine Rechtecke mehr gibt,
407    //weil der Iterator da Ende der Liste erreicht
    if (unusedRectangles[stripe].empty() ||
409        itR > int(unusedRectangles[stripe].size() - 1)){
        return {a_p, {-2, -2}};
411    }

413    //das Rechteck wird aus der Lsite unusedRectangles abgelesen
    rep = unusedRectangles[stripe][itR];
415

    //falls das Rechteck ueber die Grenzen des grossen Rechtecks
417    // hinausreichen wuerde
    if (rep->getSize() > hole.x2)
419        return {a_p, {-3, -3}};

421    //zurueckgegeben wird das Rechteck und die Koordinaten der Luecke
    return make_pair(rep, make_pair(hole.x1, hole.x2));
423 }

425 //die Methode versucht im Verbesserungsverfahren, neue Rechtecke
// ins grosse Rechteck zu legen
427 vector<Rec*> Solver::addNew(Rec* rep){
    //ein vector mit allen neu eingefuegten Rechtecke
429    vector<Rec*> added;

431    //das Rechteck wird in alle Streifen eingefuegt
    for (int i = rep->getBegin(); i < rep->getEnd(); i++)
433        placed[i].insert(rep);
    added.pb(rep);
435

    //alle Streifen werden verarbeitet, um potenziell neue

```



```

437 // Rechtecke einzufuegen
    for (int i = 0; i < M; i++){
439         auto v = processStripeReturn(i);
            added.insert(added.end(), v.begin(), v.end());
441     }

443     return added;
    }

445 //diese Methode entfernt alle Rechtecke, die mit einem im Verbesserungsverfahren
447 // gelegten Rechteck kollidieren und aktualisiert die Platzierung,
// falls es sich ein besserer Gesamtflaecheninhalt ergibt
449 bool Solver::removeCollisions(int area, pair<Rec*, iPair> rep){
    //alle Streifen mit allen platzierten Rechtecken werden kopiert
451     auto placedOld = placed;

453     //alle kollidierenden Rechtecke
    vector<pair<set<Rec*, setRecSort>::iterator, Rec>> collisions;
455
    //alle Rechtecke, die man entfernen muss
457     vector<pair<int, set<Rec*, setRecSort>::iterator>> to_remove;
    auto [rr, hole] = rep;

459     //die Koordinaten des einzufuegenden Rechtecks werden
    // festgelegt
    rr->x2 = hole.second;
461     rr->x1 = rr->x2 - rr->getSize();

463
    Rec rep_c = *rr;
    rep_c.x2 = rep_c.x1;
465     Rec *rr_p = &rep_c;

467
    //alle Kollisionen werden gesucht
    for (int i = rr->getBegin(); i < rr->getEnd(); i++){
469         //die Stelle in jedem Streifen fuer die Platzierung wird gesucht
        auto it = upper_bound(placedOld[i].begin(), placedOld[i].end(), rr_p, smallerx2);
471
        //alle Kollisionen werden in collisions gespeichert
473         for (; it != placedOld[i].end() && ((*it)->x1 <= rr->x2); it++){
            Rec &r_copy = *(*it);
475             collisions.pb({it, r_copy});
477         }
479     }

481     //die Koordinaten aller kollidierenden Rechtecke werden auf -1 gesetzt
    for (auto [it, r_copy]: collisions){
483         (*it)->x1 = -1;
        (*it)->x2 = -1;
485     }

487     //alle Rechtecke mit Koordinaten -1 werden aus der Platzierung entfernt
    for (int i = 0; i < M; i++){
489         for (auto it = placed[i].begin(); it != placed[i].end(); it++){
            if ((*it)->x1 == -1)
491             to_remove.pb({i, it});
        }
493     for (auto [stripe, it]: to_remove)
        placed[stripe].erase(it);
495
    //neue Rechtecke werden in die Platzierung eingefuegt
497     vector<Rec*> added = addNew(rr);

499     //der Gesamtflaecheninhalt aller neu platzierten Rechtecke
    // wird berechnet
    int new_area = calculateAreaUsed();
501     double diff = new_area - area;

503
    //die neue Platzierung wird angenommen
505     if (diff > 0) {
        return true;
507     }
    //die neue Platzierung wird abgelehnt
509     else {

```

```
511     //alle neu eingefuegten Rechtecke werden entfernt
512     for (auto r: added){
513         r->x1 = -1;
514         r->x2 = -1;
515     }
516
517     //die Koordinaten aller urspruenglich kollidierenden
518     // Rechtecke werden zurueckgesetzt
519     for (auto [it, r_copy]: collisions){
520         (*it)->x1 = r_copy.x1;
521         (*it)->x2 = r_copy.x2;
522     }
523
524     //die Platzierung wird zurueckgesetzt
525     placed = placedOld;
526     return false;
527 }
528
529 //diese Methode gibt den Gesamtflaecheninhalt aller Rechtecke,
530 // die platziert wurden
531 int Solver::calculateAreaUsed(){
532     int sum = 0;
533     for (auto r: rectangles)
534         if (r->x1 > -1)
535             sum += r->getArea();
536     return sum;
537 }
```