

Aufgabe 1: Flohmarkt

Teilnahme-Id: 55628

Bearbeiter dieser Aufgabe:
Michal Boron

April 2021

Inhaltsverzeichnis

1	Lösungsidee	2
1.1	Formulierung des Problems	2
1.2	Komplexität des Problems	3
1.3	Heuristik	3
1.3.1	Konversion der Eingabe	4
1.3.2	Greedy-Algorithmus	4
1.3.3	Heuristisches Verbesserungsverfahren	6
1.4	Diskussion der Ergebnisse	8
1.4.1	Grenzen der Heuristik	8
1.4.2	Qualität der Ergebnisse	8
1.5	Laufzeit	9
2	Umsetzung	9
3	Beispiele	9
3.1	Beispiel 1	9
3.2	Beispiel 2	9
3.3	Beispiel 3	9
3.4	Beispiel 4	10
3.5	Beispiel 5	10
3.6	Beispiel 6	10
3.7	Beispiel 7	10
4	Quellcode	10

Program: erweitere Zeiträume um Minuten ✓

- Definitionen, Modellierung des Problems ✓
- (Themenbezogene Arbeiten)
- Komplexität ✓
 - Notwendigkeit einer Heuristik ✓
- Konversion ✓
- heuristisches Verfahren
 - Greedy-Anlegen am Anfang ✓
 - heuristisches Verbesserungsverfahren
 - * hill climbing
- Diskussion der Ergebnisse
 - Grenzen/Mängel der Heuristik
 - * was wird nicht erkannt? (edge-cases)
 - * was lässt sich nicht eindeutig ausschließen?
 - * getroffene Annahmen
 - Qualität der Ergebnisse
 - * Qualität der Ergebnisse am Anfang (Greedy-Verfahren)
 - * Qualität bzgl. des großen Flächeninhalt, des Gesamtflächeninhalts aller Rechtecke, %
 - * was und wann kann nicht verbessert werden? (Beispiel 4: 7370)
- Laufzeit
- Umsetzung
 - Klasse Rec
 - Klasse Hole
 - Klasse Solver
 - Eingabformat!
- Beispiel s. unten

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 auch 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: Jedes Rechteck r_i in Z' besitzt 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

Wir zeigen, dass FLOHMARKT-PROBLEM NP-vollständig ist, indem wir zunächst zeigen, dass es in NP liegt und auch NP-schwer ist.

Offensichtlich kann dieses Problem von einer nichtdeterministischen Turingmaschine bezüglich der Eingabelänge in Polynomialzeit gelöst werden. Gegeben sei eine Platzierung der Rechtecke aus Z innerhalb von R . Man kann leicht einen in Polynomialzeit laufenden Algorithmus entwickeln, der anhand der Koordinaten der kleineren Rechtecke überprüft, ob keines der Rechtecke über die Grenzen von R hinausreicht und ob kein Paar von Rechtecken aus Z sich überdeckt. Somit liegt FLOHMARKT-PROBLEM in NP.

Um zu beweisen, dass das FLOHMARKT-PROBLEM NP-schwer ist, 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 kann 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 0/1-Rucksackproblem ist NP-vollständig.[2]

TODO: sprawdzić/zmieniĆ źródło

Gegeben sei eine Instanz des Rucksackproblems. Wir können 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, deren Länge der Größe des Rucksacks entspricht und deren Breite ebenfalls 1 beträgt. Somit bilden wir eine Instanz eines Flohmarkts, der eine Stunde dauert und deren 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 das FLOHMARKT-PROBLEM äquivalent zum ursprünglichen 0/1-Rucksackproblem. Wenn wir jede Instanz des FLOHMARKT-PROBLEMS in Polynomialzeit lösen können, können wir auch jedes 0/1-Rucksackproblem in Polynomialzeit lösen. Somit ist das FLOHMARKT-PROBLEM NP-schwer und, da es auch in NP liegt, ist somit auch NP-vollständig.

Da dieses Problem NP-vollständig ist, muss man über einen optimalen Algorithmus zum FLOHMARKT-PROBLEM nachdenken. Cormen et al. beschreiben, dass es grundsätzlich drei Ansätze zum Lösen eines NP-vollständigen gibt.[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, wenn die Anzahl der kleineren Rechtecke in der Eingabe sich in Ordnung von Hunderten befindet. Die praktische Laufzeit eines exponentiellen Algorithmus ist 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 zuerst einen Greedy-Algorithmus laufen, um ein Ausgangsergebnis zu erzeugen und danach führen wir einen Berg-

steigeralgorithmus (engl. *hill climbing algorithm*) aus, der das Ausgangsergebnis heuristisch optimiert, indem er ein lokales Maximum durch mehrmalige Mutationen findet.

TODO: Teil mit Mutationen verbessern?

1.3.1 Konversion der Eingabe

Wie schon im Abschnitt 1.1 erwähnt wurde, können die Gedanken 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 oder sogar mehrere unterschiedlichen Flohmärkte behandeln. Zur Darstellung eines mehrtägigen Flohmarktes kann man die gesamte 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 41:00 (17:00 + 24 Stunden) dargestellt werden. Dann ist der Zeitraum von 17:00 bis 34:00 an keiner Stelle besetzt. Ebenfalls, wenn ein angegebener Zeitraum an einer Stelle unterbrochen ist, etwa dauert der Flohmarkt von 7:00 bis 9:00 und dann von 12:00 bis 15:00, kann der Zeitraum von 7:00 bis 15:00 angegeben werden und alle Anmeldungen, die zumindest zum Teil in der Pausenzeit liegen, können aus der Eingabe entfernt werden oder können gar nicht angegeben werden. Mehrere unterschiedlichen Flohmärkte mit derselben Länge N kann man analog kodieren. Es hängt nur von der Eingabe ab.

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 Öffnungszeit 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 (s. Abb. 1a). Die Breite eines solchen Streifen ist äquidistant für alle Streifen und, da man Stände am Flohmarkt nur zu vollständigen Stunden vermietet, beträgt die Breite eines Streifens 1 Stunde.¹ 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 S_k .

Im Programm sind diese Streifen einfach Listen mit allen kleineren Rechtecken, deren Breite m_i sich in diesem Streifen enthält. 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 wir es in jede Liste S_j für alle j hinzu, die die folgende Bedingung erfüllen: $b_i \leq j < e_i$. Das bedeutet, dass ein Rechteck von $b_i = 1$ (nach Konversion, in vollständigen Stunden) bis $e_i = 5$ in den folgenden Streifen enthalten wird: S_1, S_2, S_3, S_4 . Im Streifen S_5 wird er nicht enthalten, da die Miete mit dem Anfang der 5. Stunde endet. Wie Streifen 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 jedem Streifen S_j unabhängig voneinander nach folgenden Kriterien in dieser Reihenfolge: 1) fallend nach dem Wert e_i , 2) aufsteigend nach dem Wert b_i

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

und 3) fallend nach der Fläche jedes Rechtecks. Somit sind die ersten Rechtecke in jeder Liste S_j diejenigen, deren Wert e_i am größten ist — oft diejenigen, die am breitesten im Streifen sind. Die Reihenfolge wurde so gewählt, damit wir in dieser Reihenfolge versuchen, die Rechtecke aus den Streifen ins große Rechteck R zu platzieren. Die Idee hinter dieser Platzierung ist, dass wir zuerst die breitesten Rechtecke „links“, also an niedrigeren x -Werten, platzieren, so weit es geht. Dann füllen wir die Lücken „rechts“ (an größeren x -Werten) mit schmalleren Rechtecken. Die grobe Idee ist, dass wir das Rechteck R quasi vom Punkt $(0,0)$ bis zum Punkt (N,E) mit immer schmalleren 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 Rechtecken 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 von zwei Rechtecken in j oder den Seiten des Rechtecks R entsprechen. Man beachte insbesondere, dass es keine Lücke zwischen zwei Rechtecken gibt, die eine gemeinsame Seite haben. Die *Größe* einer Lücke beträgt: $x_2 - x_1$.

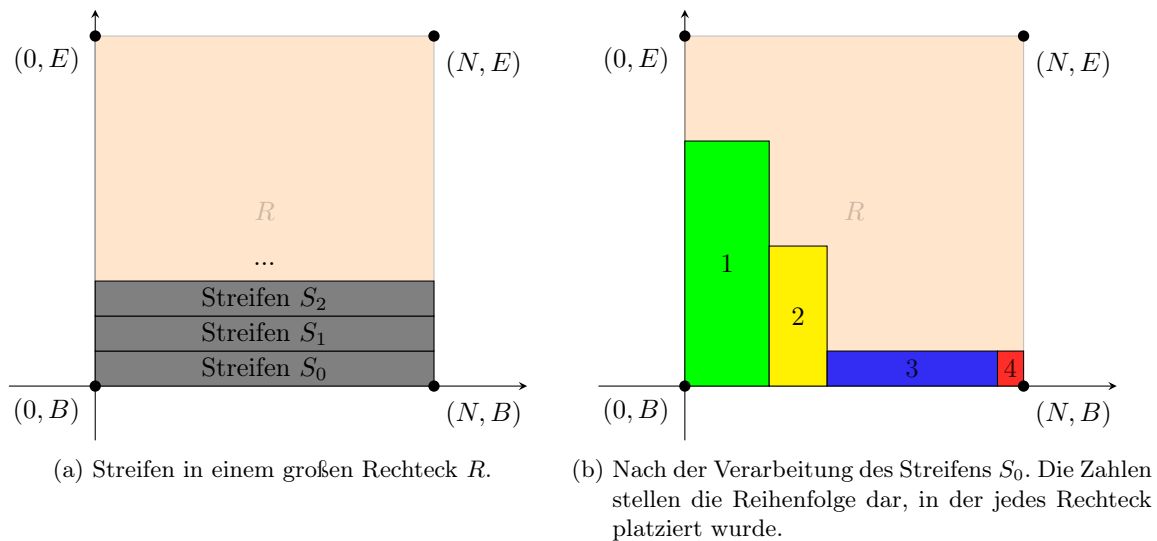


Abbildung 1: Die Abbildung des Rechtecks R auf einem Koordinatensystem. Die Seiten entlang der x -Achse haben die Länge N und die Seiten entlang der y -Achse haben die Länge M .

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 diesem Streifen, ob sein Wert b_i gleich dem Wert j ist, also ob das Rechteck (die Anmeldung) mit dem aktuellen Zeitpunkt 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 R und übergehen zum Rechteck r_{i+1} . Auf der Abbildung 1b sieht man den verarbeiteten Streifen S_0 . Insbesondere erkennt man gut die Reihenfolge der Sortierkriterien der Rechtecke im Streifen.

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

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 an 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 genug Platz über diesem Rechteck (genau: s_i) in den weiteren Streifen $S_{b_i+1}, S_{b_i+2}, \dots, S_{e_i-1}$ 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 genug große Lücke für ein Rechteck r_i gibt. Wenn es eine solche Lücke nicht gibt, bedeutet, 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.

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 im Streifen, 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 nochmal nutzen, da wir voraussetzen, dass die Streifen in der aufsteigenden Reihenfolge ein nach dem anderen verarbeitet werden. Dann kann es sein, dass es sich eine Lücke zwischen den Punkten (x_j, j) und $(x_j + \ell, j)$ der Länge ℓ an einer Stelle in einem Streifen j befindet und dass ein Rechteck r_i mit $s_i < \ell$ 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 des Verbesserungsverfahrens besteht darin, man führt zu einer Veränderung (Mutation) in C , d.h., ein nicht platziertes Rechteck r wird in R platziert und es müssen ggf. Rechtecke aus R entfernt werden, die mit r kollidieren. So kommt man auf einen neuen Zustand, eine neue Platzierung der Rechtecke C' . 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 noch möglich ein Zustand C zu weiter verändern. Wir können diesen Ansatz als einen Bersteigeralgorithmus klassifizieren.

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 mehr Lücken 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 j befinden, die nicht platziert wurden. Die Liste U_j muss man auf eine Weise sortieren. Jedes Rechteck r_i in jeder solchen Liste ordnen 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 beschrieben.

Danach muss man die Lücken in einer Platzierung in jedem Streifen 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 der Wert $x_i + s_i$ gleich dem Wert x_{i+1} ist. Wenn nicht, gibt es eine Lücke zwischen den Rechtecken r_i und r_{i+1} . Dazu muss man auch das erste Rechteck untersuchen, ob die Koordinate x_0 des Rechtecks r_0 dem Wert 0 entspricht bzw. ob die Koordinate $x_l + s_l$ des letzten Rechtecks r_l in diesem Streifen mit dem Wert N übereinstimmt. Wenn nicht, entstehen auch Lücken zwischen den Wänden des großen Rechtecks R .

Danach werden alle Listen H_j zu einer Liste H zusammengebracht. Diese Liste muss auch auf eine Weise geordnet werden. 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.

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

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 gesteuert 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 mindestens ein Rechteck gibt, 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

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:  $j \leftarrow itH.Strreifen \quad \triangleright j$  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:    $\Sigma_C \leftarrow |C|$   $\triangleright$  der Gesamtflächeninhalt aller platzierten Rechtecke in  $C$ 
10:   $C' \leftarrow \text{next}(C, itH, itR)$ 
11:   $itR \leftarrow itR + 1$ 
12:   $\Sigma_{C'} \leftarrow |C'|$ 
13:  if  $\Sigma_{C'} > \Sigma_C$  then
14:     $H \leftarrow \text{getAllHoles}(C')$ 
15:     $U \leftarrow \text{getRecs}(C')$ 
16:     $itH \leftarrow H.begin$ 
17:     $k \leftarrow itH.Strreifen$ 
18:     $itR \leftarrow U_j.begin$ 
19:     $C \leftarrow C'$ 
20:  end if
21: end while

```

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 ans 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 ein unter dem Iterator itR stehendes 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 so gelegt, dass $x_2 = x_r + s_r$. Auf diese Weise beträgt der Wert $x_r := x_2 - s_r$.² Selbstverständlich kann an dieser Stelle zu Kollisionen kommen — Rechtecke können sich überdecken. Vor dem Platzieren prüft man nicht, ob es durchgehend eine Lücke zwischen 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 allen 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 13. Wenn der Gesamtflächeninhalt der Platzierung C' größer, also besser, 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. $U_k.begin$ gesetzt.

²Die Situation, in der $x_r < 0$ gilt, wird schon in der zweiten Funktion dadurch ausgeschlossen, dass der Iterator itR inkrementiert wird und das nächste Rechteck gewählt wird.

1.4 Diskussion der Ergebnisse

1.4.1 Grenzen der Heuristik

Im Abschnitt 1.2 wurde bewiesen, dass das FLOHMARKT-PROBLEM NP-vollständig ist. Es gibt $|Z|!$ mögliche Anordnungen der Rechtecke. Deshalb für $|Z|$ in der Ordnung von ca. 700 wird eine Brute-Force-Lösung in einer nicht akzeptablen Zeit gelöst. Das ist auch der Grund, warum wir eine Heuristik verwenden. Allerdings, da eine Heuristik ein Approximationsalgorithmus ist und nur nahezu optimale Ergebnisse liefert, muss es Kompromisse geben. Dieser Kompromiss betrifft vor allem die Laufzeit und dafür, dass der Algorithmus in Polynomialzeit läuft, trifft das Programm an vielen Stellen vereinfachte Entscheidungen, die nicht zum optimalen Ergebnis führen. In diesem Abschnitt diskutieren wir nur über die Grenzen der Heuristiken im Programm 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 kein optimales Ergebnis zum Beispiel 2, weil dieses Ergebnis im Laufe des Verbesserungsverfahrens verbessert wird.

Im Verbesserungsverfahren wurden mehrere Kompromisse zugunsten der Laufzeit gemacht. Vor allem liegt die Schwierigkeit zugrunde dem Verfahren — warum ein Bergsteigeralgorithmus und nicht z.B. ein Verfahren mit simuliertem Abglühen oder ein ganz anderer heuristischer Ansatz? Außerdem liegen die Schwierigkeiten des Verbesserungsverfahrens auch an der 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 das Rechteck deutlich kleiner ist als die Lücke, oder man könnte die Rechtecke in darüber und darunter stehenden Streifen in Betracht ziehen. Wie beim Greedy-Algorithmus am Anfang, hängt die Reihenfolge der Rechtecke beim Ausfüllen der Lücken von der Reihenfolge in Listen S_j ab.

1.4.2 Qualität der Ergebnisse

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

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

- Der Verhältnis vom Gesamtflächeninhalt der ins große Rechteck gelegten Rechtecke zu 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 Veranstaltern 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 Personen 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 wenigen Personen würde ein Ergebnis interessieren, das vielleicht um ein paar Prozent besser ist, aber dessen Bestimmung mehrere Stunden (oder Tage!) dauert. Deshalb wurde auch die Brute-Force-Lösung ausgeschlossen.

TODO: Abglühen

Ergebnisse einzeln beschreiben

Ergebnisse nach dem Greedy-Algorithmus

welches Ergebnis konnte (nicht) verbessert werden?

Kriterien I	Kriterien II	Bsp. 1	Bsp. 2	Bsp. 3	Bsp. 4	Bsp. 5	Bsp. 6	Bsp. 7
greaterHolesSize	smallerSize	8028	9077	8778	7370	8705	10000	9979
greaterHolesSize	greaterSize	8028	9077	8778	7370	8705	10000	9973
greaterHolesSize	smallerArea	8028	9077	8778	7370	8705	10000	9979
greaterHoleaSize	greaterSize	8028	9077	8778	7370	8705	10000	9973
greaterHoleaSize	greaterEnd	8028	9077	8778	7370	8705	10000	9973
smallerHolesSize	smallerSize	8028	9077	8778	7370	8599	10000	9980
smallerHolesSize	greaterSize	8028	9077	8778	7370	8599	10000	9973
smallerHolesSize	smallerArea	8028	9077	8778	7370	8599	10000	9980
smallerHolesSize	greaterArea	8028	9077	8778	7370	8599	10000	9973
smallerHolesSize	greaterEnd	8028	9077	8778	7370	8599	10000	9991

1.5 Laufzeit

Literatur

- [1] T.H. Cormen u.a. *Introduction To Algorithms. Third edition.* Introduction to Algorithms. MIT Press, 2009. ISBN: 9780262533058.
- [2] Zhang Li'ang und Geng Suyun. „The Complexity of the 0/1 Multiknapsack Problem“. In: *Journal of Computer Science and Technology* 1.1 (1986), 46–50.

2 Umsetzung

TODO: Eingabeformat von Minuten erwähnen

3 Beispiele

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 .

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: $10000 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller platzierten Rechtecke: $7370 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller Rechtecke: $10534 \text{ [m} \cdot \text{h]}$

3.5 Beispiel 5

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

Der Gesamtflächeninhalt aller platzierten Rechtecke: $8705 \text{ [m} \cdot \text{h]}$

Der Gesamtflächeninhalt aller Rechtecke: $30940 \text{ [m} \cdot \text{h]}$

3.6 Beispiel 6

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

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

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

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: $10000 \text{ [m} \cdot \text{h]}$

Weitere Beispiele:

- s. Abschnitt Konversion
- andere Zeiten ✓ (Bsp 8)
- andere Länge ✓ (Bsp 8)
- mehrtägiger Flohmarkt
- mehrere Flohmärkte ~~+ mehrere Flohmärkte mit unterschiedlichen Längen~~
- unterbrochener Zeitraum ✓ (Bsp 10; Pause 12-13)
- unterbrochene Länge
- Minuten
- edge-cases, bei denen der Algorithmus nicht funktioniert

4 Quellcode