Aufgabe 4: Schrebergärten

4.1 Einleitung

Die Aufgabe besteht im Grunde daraus, eine gegebene Menge von Rechtecken in ein möglichst kleines umschließendes Rechteck unterzubringen. Dieses Problem kommt in ähnlicher Weise auch in vielen Praxisanwendungen zum Vorschein, neben naheliegenden Problemen wie dem Anordnen von Waren oder dem Kauf von Grundstücken kann es auch für Planungsprobleme benutzt werden, bei dem die Rechtecke beispielsweise Aufgaben repräsentieren, wobei die Breite den Zeitaufwand darstellte und die Höhe den Ressourcenverbrauch.

Auch ist das Problem von Relevanz fürs effiziente Laden von Webseiten: Damit nicht alle Bilder einzeln vom Server abgefragt werden müssen (mit dem einhergehenden Zeitaufwand), werden diese oftmals vor der Datenübertragung auf dem Server zu einem großen Bild zusammengefügt. Dies ist besonders bei Icons und ähnlich kleinen Bildelementen der Fall. Damit sich dies lohnt, darf einerseits das Packen der Bilder nicht zu lange dauern und andererseits darf das neu entstehende Bild nicht viel größer sein als die Summe der Einzelbilder.

In der Forschung ist das Problem auch als „Optimal Rectangle Packing“ bekannt. Es ist NP-schwer und gehört zu den Behälterproblemen[9] („Bin Packing Problems“[10]) der Packungsprobleme („Packing Problems“[11]). Für den interessierten Leser ist ein Paper von Huang und Korf von 2013[12] (mit zusätzlichem Programmcode[13]) zu empfehlen, in dem eine der besten Möglichkeiten zu finden ist, das Problem exakt zu lösen. Im Folgenden werden wir hier aber nur eine einfachere und annähernde Lösung vorstellen, die auf einer anderen Idee basiert.

4.2 Lösungsidee

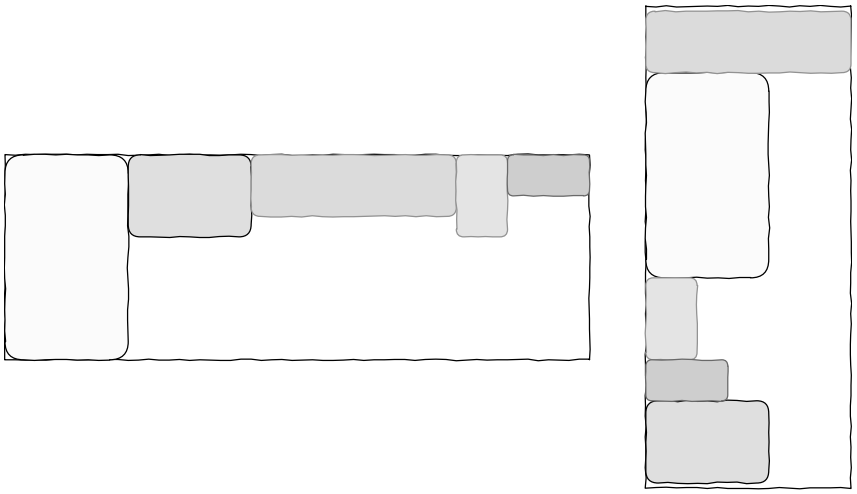

Abbildung 4.1: Einfache senkrechte und waagerechte Anordnung.

[9] https://de.wikipedia.org/wiki/Behälterproblem
[10] https://en.wikipedia.org/wiki/Bin_packing_problem
[11] https://en.wikipedia.org/wiki/Packing_problems
[12] Paper von E. Huang und R. Korf: *Optimal Rectangle Packing: An Absolute Placement Approach*, Journal of Artificial Intelligence Research, 46(1):47-87, 2013. https://jair.org/index.php/jair/article/view/10797/25774
[13] https://github.com/pupitetris/rectpack

Zunächst kann es sich zum besseren Verständnis lohnen, einen Blick auf die einfachen Möglichkeiten lohnen, ein umschließendes Rechteck zu finden. Die einfachste Möglichkeit ist wohl alle Rechtecke nebeneinander oder übereinander, wie beispielhaft geschehen in Abbildung 4.1, anzuordnen. Dies sorgt jedoch in den meisten Fällen für eine sehr hohe Platzverschwendung, ergibt aber bereits die ersten oberen Grenzen für unsere Lösung.

Zum Finden einer guten Lösung müssen nun effizient mehrere mögliche Lösungen durchprobiert werden. Hier wird der von Richard E. Korf entwickelte Algorithmus[14] vorgestellt, abgeändert zu einem annäherndem Verfahren[15] von Matt Perdeck. Dabei werden die möglichen einfassenden Rechtecke von breit nach schmal durchsucht.

1. Sortiere die Rechtecke absteigend nach ihrer Höhe.
2. Beginne mit einem einfassendem Rechteck, das die Höhe des höchsten Rechtecks besitzt und unendliche Breite.
3. Beginne nun die Rechtecke, angefangen mit dem höchsten, zu platzieren: Wähle dabei jeweils den Platz ganz links, der möglich ist; wenn es mehrere Möglichkeiten gibt, wähle den höchsten Platz, siehe Abbildung 4.2.
4. Passe die Breite des umgebenden Rechtecks auf das minimal mögliche an.
5. Konnten alle Rechtecke untergebracht werden?
   a) Ja: Speichere das resultierende umgebende Rechteck, wenn es das kleinste bisherige ist. Verringe die Breite des umgebenden Rechtecks um 1.
   b) Nein: Das geprüfte umgebende Rechteck war zu klein, erhöhe die Höhe um 1.
6. Wenn die Fläche des neuen umgebenden Rechtecks nach der Anpassung nun kleiner ist als die Summe der Flächen aller zu platzierenden Rechtecke, erhöhe die Höhe solange, bis dies nicht mehr der Fall ist. Andernfalls gäbe es keinerlei Möglichkeit, alle Rechtecke unterzubringen.
7. Wenn das neue Rechteck größer ist als das bisher kleinste gefundene, reduziere die Breite solange, bis dies nicht mehr der Fall ist.
8. Wenn das neue umgebende Rechteck nun schlanker ist als das breiteste Rechteck, beende die Suche; es wird kein besseres Rechteck mehr gefunden werden.
9. Gehe zurück zu Schritt 3, um die Suche mit dem neuen umgebenden Rechteck fortzusetzen.

Für eine optimale Lösung müssten in Schritt 5 alle möglichen Anordnungen erprobt werden, wenn das Rechteck nicht passt. Dies würde z.B. mit einer Backtracking- Strategie[16] erfolgen und ist recht zeitaufwändig. Sich auf die sortierte Reihung zu verlassen und mit Hilfe des beschriebenen Greedy-Verfahrens[17] zu prüfen, hat sich in der Praxis als gute und einfache Weise erprobt, solide Lösungen zu erhalten.

Prinzipiell wäre es auch möglich, verschiedene zufällige Reihenfolgen zu erproben, bevor man aufgibt oder andere feste Anordnungen zu testen. Wenn man dies umsetzen würde, wäre es möglich, einen Parameter einzuführen, der die Anzahl der zu testenden Anordnungen angibt und dadurch eine Abwägung zwischen Geschwindigkeit und Güte der Lösungen erlaubt.

[14] Paper von Richard E. Korf: *Optimal Rectangle Packing: Initial Results*, ICAPS 2003 Proceedings. https://www.aaai.org/Papers/ICAPS/2003/ICAPS03-029.pdf

[15] https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu

[16] https://de.wikipedia.org/wiki/Backtracking

[17] https://de.wikipedia.org/wiki/Greedy-Algorithmus

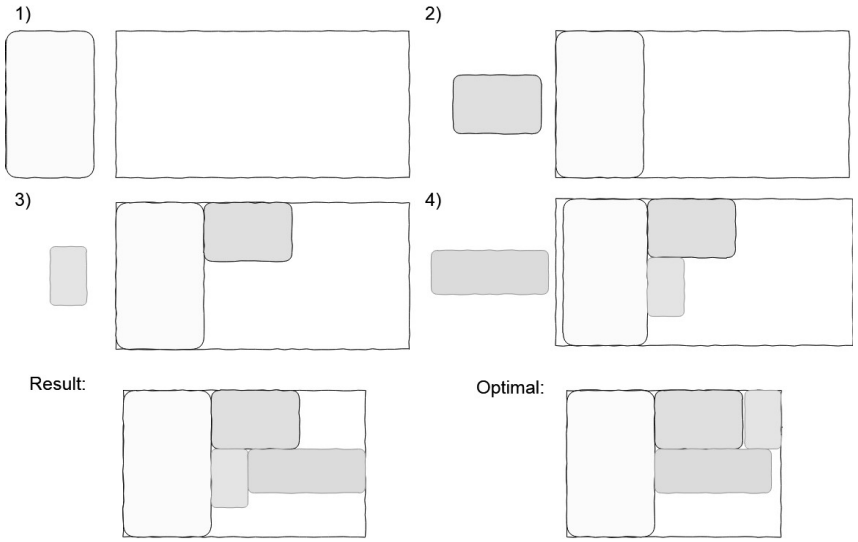

Abbildung 4.2: Beispielhafte Ausführung von Schritt 3 für vier zu platzierende Rechtecke. Jeweils das momentan einzufügende Rechteck und der Rand des umgebenden Rechtecks vor dem Einfügen sind dargestellt.

4.3 Platzierung der Rechtecke

Um wie in Schritt 3 beschrieben den Platz ganz links zu finden, gibt es mehrere Möglichkeiten. Der naive Ansatz besteht darin, in einem zweidimensionalen Array alle belegte Plätze zu markieren und bei der Platzierung dann von oben nach unten und von links nach rechts zu iterieren. Dies ist jedoch nicht wirklich effizient.

Eine Möglichkeit, dies zu verbessern, ist eine Datenstruktur, die ähnlich zu einer der oben genannten Lösungsideen Felder verwaltet, die belegt oder frei sein können. Dabei assoziiert man jede Spalte mit einer Breite und jedes Feld mit einer Höhe, und es werden nur so viele Felder erzeugt, wie benötigt werden.

Man beginnt dabei mit einem einzigen Feld, dessen Höhe und Breite dem zu testenden einfassenden Rechteck entspricht. Rechtecke werden als Paar von Zahlen, die Breite und Höhe entsprechen, dargestellt. Wird nun ein Rechteck hinzugefügt, ist dies einfach in dieses freie Feld möglich. Dabei entsteht jedoch ein Feld, das sowohl belegt, als auch unbelegt ist. Daher wird dieses Feld geteilt: horizontal auf der Höhe des Rechtecks und vertikal in der Breite. Das Ergebnis sind vier Felder.

Soll nun ein neues Rechteck hinzugefügt werden, wird durch diese Felder entsprechend von links nach rechts und von oben nach unten iteriert, um einen möglichen Platz zu finden. Wird ein solcher gefunden, kann eingefügt werden, wobei eventuell wiederum Felder geteilt werden müssen.

Am besten wird die Idee mit einem kurzen Beispiel verdeutlicht (siehe Abbildung 4.3). Man beginnt mit einem einzigen leeren Feld, dessen Abmessungen denen des zu prüfenden einfassenden Rechtecks entsprechen. Wird nun ein neues Rechteck hinzugefügt (1), ist die Position für dieses leicht zu finden, es entspricht der oberen linken Ecke des einzigen Feldes. Beim Einfügen tritt nun aber eine Situation ein, bei der ein Feld sowohl belegt als auch frei ist. Daher wird das Feld nun in vier neue Felder gespalten (2).

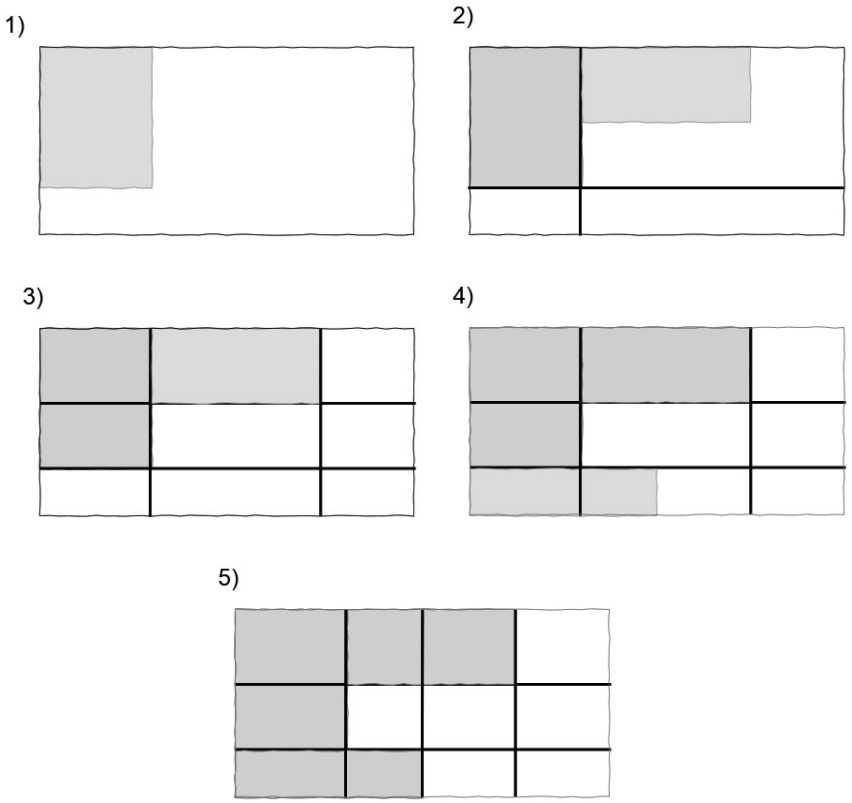

Abbildung 4.3: Beispiel für die beschrieben Datenstruktur aus Feldern (neu eingefügte Rechtecke in grün, belegte Felder in rot).

Wenn nun ein neues Rechteck eingefügt werden soll, wird zunächst durch jede Spalte, d.h. von links nach rechts und von oben nach unten iteriert. Im Beispiel des neu eingefügten Rechtecks war das Feld unter dem bereits eingefügten Rechteck nicht hoch genug, daher wird es in das obere rechte Feld eingefügt. Hier kommt es wieder vor, dass Felder geteilt werden müssen. In (4) ist das untere Feld hoch genug, so dass noch geprüft werden muss, ob zusammen mit nebeneinanderliegenden Felder genug Breite vorhanden ist.

Verbesserungen

Statt bei Schritt 5b) die Höhe nur um 1 zu erhöhen, kann man sich bei der vorgestellten Heuristik überlegen, dass eine größere Erhöhung möglich ist. Es muss nämlich entweder um die Höhe des Rechteckes erhöht werden, das nicht platziert werden konnte oder aber soweit, dass die bisher platzierten Rechtecke anders platziert werden können. Die erste Zahl ist leicht zu bekommen, für den zweiten Wert ist kurzes Nachdenken erforderlich. Aber auch diese Zahl ist kein Problem, man prüft für jedes Rechteck, das nicht in einem Feld platziert werden konnte, wie viel Höhe mehr erforderlich wäre, damit eine Platzierung gelingt und merkt sich das Minimum. Dann wird in Schritt 5a) statt um 1 das Rechteck um das Minimum der zwei vorgestellten Werte erhöht.

Eine weitere Überlegung wäre, einen Parameter einzuführen, bei dem man zufällige Reihenfolgen statt der sortierten Reihenfolge testet. Durch das Testen einer einzigen Reihenfolge ist das Programm sehr schnell und kommt mit einer sehr großen Anzahl an Schrebergärten zurecht, jedoch geht dies zu Lasten der Güte der Lösung. Mit Hilfe eines solchen Parameters könnte man Qualität und Geschwindigkeit manuell gegeneinander abwägen.

4.4 Alternative Lösungsideen

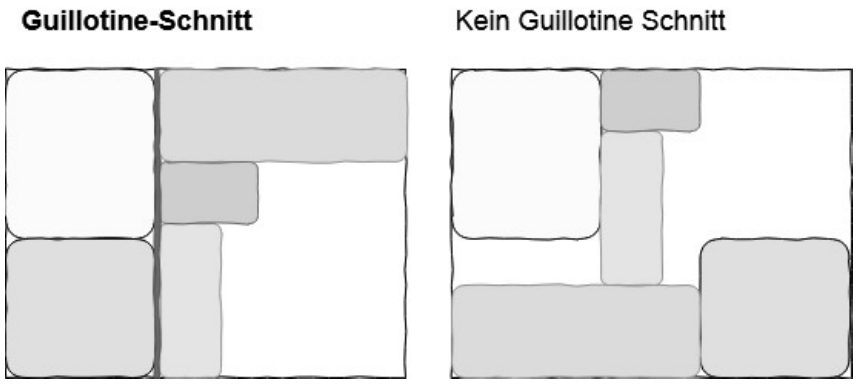

Abbildung 4.4: Beispiel für Anordnungen mit und ohne Guillotonie-Schnitt.

Die Aufgabenstellung erlaubt zur Lösung viele verschiedene Lösungswege und Heuristiken. Meist werden verschiedene Lösungen erprobt und vergleichend evaluiert. Dies kann auf unterschiedliche Arten geschehen, im akademischen Bereich ist das Problem ergiebig analysiert. Verbesserungen zum oben genannten Verfahren werden beispielsweise in einem weiteren Paper[18] von Richard E. Korf vorgestellt.

Eine Übersicht über verschiedene Verfahren bietet u.a. ein Paper[19] von Jukka Jylänki mit entsprechendem Programmcode[20] und visueller Darstellung der Algorithmen[21].

Weitere populäre Ansätze sind beispielsweise Lösungen mit sogenannten Guillotine-Schnitten zu betrachten, d.h. Lösungen, bei denen man das umfassende Rechteck sozusagen zerschneiden kann, ohne dabei die enthaltenen Rechtecke zu zerschneiden. Veranschaulicht ist diese Lösungsidee in Abb. 4.4.

Außerdem sind in der Tabelle 4.5 die optimalen Lösungen für das Packen von aufsteigenden Quadraten enthalten; an diesen kann sich orientiert werden, um die Lösungsgüte zu einzuschätzen. Dabei sind exakte Lösungen bis Größe 15 in wenigen Sekunden möglich; Heuristiken sollten deutlich mehr schaffen.

4.5 Beispiele

Im Folgenden werden die optimalen und die vom Programm errechneten Lösungen an Hand von einigen Beispielen verglichen, siehe Abbildung 4.6.

Wie bereits erwähnt, könnte die Lösungsgüte auf Kosten der Performance verbessert werden, indem man mehrere Permutationen bei der Platzierung prüft, statt nur die sortierte Reihenfolge der Rechtecke durchzurechnen.

[18] Paper von Richard E. Korf: *Optimal Rectangle Packing: New Results*, ICAPS 2004 Proceedings. https://www.aaai.org/Papers/ICAPS/2004/ICAPS04-019.pdf

[19] Paper von Jukka Jylänki: *A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing*, 2010. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.695.2918

[20] https://github.com/juj/RectangleBinPack

[21] http://www.binpacking.4fan.cz/

Size N	Optimal Solution	Waste Percent
1	1×1	0%
2	2×3	16.67%
3	3×5	6.67%
4	5×7	14.29%
5	5×12	8.33%
6	9×11	8.08%
7	7×22	9.09%
7	11×14	9.09%
8	14×15	2.86%
9	15×20	5.00%
10	15×27	4.94%
11	19×27	1.36%
12	23×29	2.55%
13	22×38	2.03%
14	23×45	1.93%
15	23×55	1.98%
16	27×56	1.06%
16	28×54	1.06%
17	39×46	0.50%
18	31×69	1.40%
19	47×53	0.84%
20	34×85	0.69%
21	38×85	0.99%
22	39×98	0.71%
23	64×68	0.64%
24	56×88	0.57%
25	43×129	0.40%

Abbildung 4.5: Tabelle für die optimale Lösungen der Anordnung von aufsteigend größeren Quadraten, d.h. 1x1,2x2,... *nxn*, entnommen aus Table 1 des Papers von Richard E. Korf: *Optimal Rectangle Packing: New Results*, ICAPS 2004 Proceedings. https://www.aaai.org/Papers/ICAPS/2004/ICAPS04-019.pdf

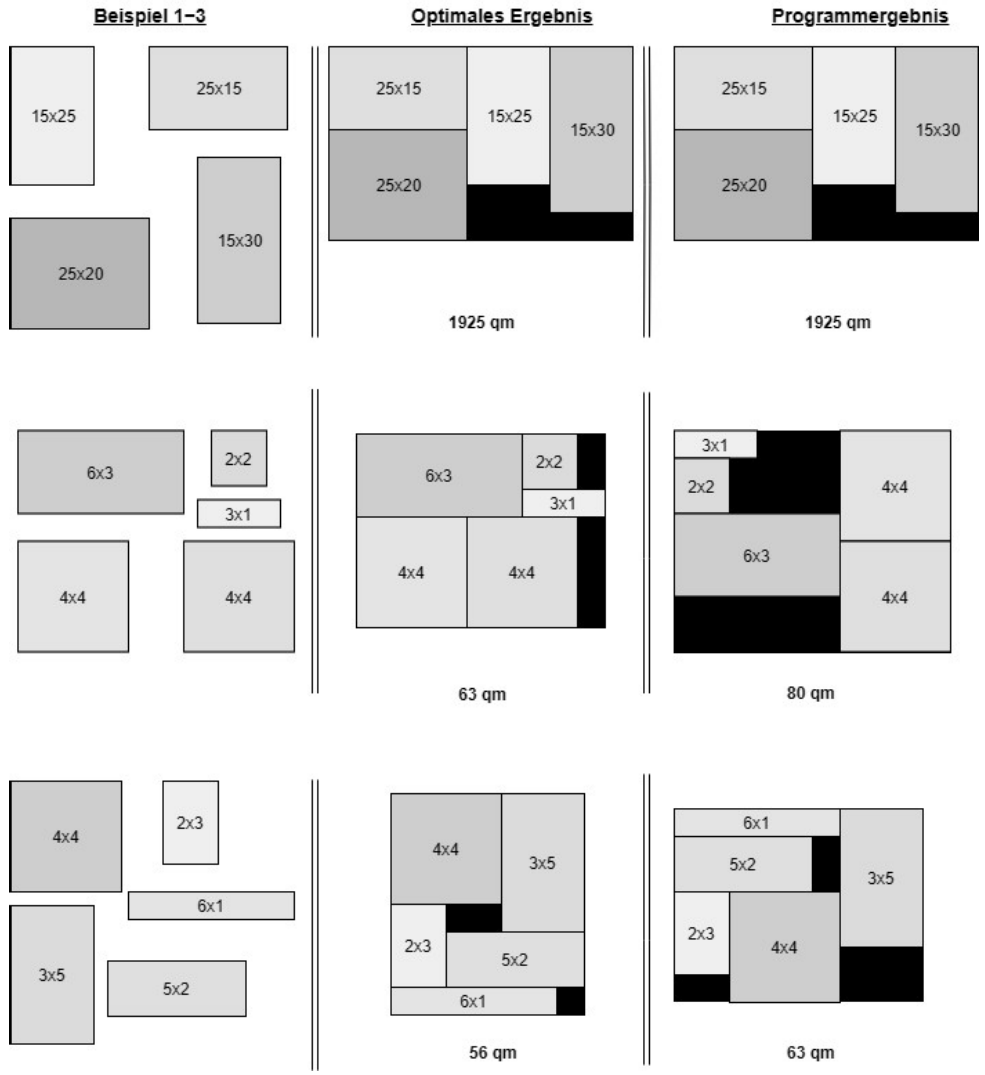

Abbildung 4.6: Beispiele samt optimaler und vom Programm errechneter Lösung.

4.6 Bewertungskriterien

Die Bewertungskriterien (Fettdruck) vom Bewertungsbogen werden hier näher erläutert (Punktabzug in []).

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Die vorgegebenen Daten sollten vom Programm intern geeignet gespeichert werden. Insbesondere sind Arrays dafür geeignet. Statt aus einer Datei oder mittels einer Eingabezeile können die Rechteckdaten auch z. B als Konstanten im Programmcode enthalten sein.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren sollte korrekt sein und allgemein genug sein, d.h. nicht auf spezielle Beispiele zugeschnitten. Auch sollte es nicht auf bis zu vier Rechtecke oder auf eine andere Zahl von Rechtecken festgelegt sein. Die Geometrie der Rechtecke sollte korrekt umgesetzt sein (z.B. Beachten von Schnittpunkten und Überlappungen). Eine Drehung der Rechtecke war durch die Aufgabenstellung eigentlich ausgeschlossen, aber da sie die Aufgabe schwieriger macht, gibt es dafür keinen Punktabzug.

- (4) [-1] **Strategie zur Optimierung mangelhaft / fehlt**:
  Grundsätzlich gibt es bei diesem Problem etliche Lösungsmöglichkeiten, weshalb beim Lösungsansatz viel Spielraum gelassen werden sollte. Auf jeden Fall sollte der Flächeninhalt des die einzelnen Schrebergärten umgebenden Rechtecks die maßgebliche Größe für die Optimierung sein. Auch Greedy-Verfahren zur Optimierung sind in Ordnung. Wichtig ist jedoch, dass klar dokumentiert ist, ob der gewählte Lösungsansatz optimale oder approximative Lösungen liefert. Gedanken und Angaben zur konkret möglichen Flächenersparnis wären zwar schön (z.B. theoretische Überlegungen zum Optimum versus Approximation oder ein Vergleich der Ergebnisse unterschiedlicher Rechenverfahren anhand von Beispielen), sind aber von der Aufgabenstellung nicht gefordert und können daher im Falle einer fehlerhaften Beschreibung nicht zu Punktabzug führen.
- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig**:
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug. Die Laufzeiteffizienz wird nicht bewertet.
- (6) [-1] **Programmausgabe ungeeignet**:
  Die Programmausgabe sollte mittels einer visuellen Darstellung erfolgen und nachvollziehbar sein. Grafische Darstellungen sind schön, aber auch Visualisierungen mit Textzeichen sind akzeptabel. Wichtig ist, dass die eingegebenen Rechtecke erkennbar sind.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige (mind. 2/3)**:
  Es wird die Dokumentation und visuelle Darstellung der Ergebnisse von mind. 2 der 3 vorgegebenen Beispiele erwartet.