Aufgabe 4: Schrebergärten

Richard Wohlbold
Team-ID: 00012

29. Oktober 2018

Inhaltsverzeichnis

1 Lösungsidee

1.1 Allgemeines zum Problem

Das Problem, mehrere Rechtecke unterschiedlicher Größe in eines zu packen, ist schon länger bekannt, da es Anwendung im Internet findet, z.B. wenn mehrere kleine Bilddateien in eine möglichst kleine gepackt werden sollen. Es ist auch bekannt, dass das Problem NP-komplett ist, d.h. es gibt keine optimale Lösung, die in Polymomialzeit läuft, man kann eine optimale Lösung also nur durch Ausprobieren aller Möglichkeiten bestimmen [1]. Deshalb kann jede effiziente Lösung nur ein *fast* perfektes Ergebnis produzieren; mann muss also zwischen Laufzeit und Genauigkeit abwägen.

1.2 Das Verfahren

Ich benutze ein Verfahren, das von Richard E. Korf in seinem Artikel *Rectangle Packing: Initial Results* beschrieben wurde und von Matt Perdeck in *Fast Optimizing Rectangle Packing Algorithm for Building CSS Sprites* genauer beschrieben ist [1, 2].

1. Alle Rechtecke werden nach ihrer Länge absteigend sortiert.
2. Daraufhin wird das erste umgebende Rechteck erstellt. Dieses wird so gewählt, dass es als Länge die Länge des längsten Rechtecks hat und dass es als Breite die Summe aller Breiten hat.
3. Nun wird versucht, das umgebende Rechteck so effizient wie möglich zu benutzen, um alle kleinen Rechtecke unterzubringen. Dabei soll versucht werden, die genutzte Breite zu minimieren. Deshalb wird ein Rechteck nach dem anderen so weit nach links platziert wie möglich. Falls es mehrere mögliche Orte gibt, die alle gleich weit links sind, wird der oberste ausgewählt.
4. Falls alle Rechtecke untergebracht werden konnten, wird die neue Lösung mit der bestehenden Lösung verglichen und die übernommen, die die kleinere Fläche besitzt.
5. Falls alle Rechtecke untergebracht werden konnten, wird die Breite des umgebenden Rechtecks auf die der Lösung minus 1 reduziert. Ansonsten wird die Höhe um 1 erhöht.
6. Falls die Länge des umgebenden Rechtecks kleiner als die des längsten Rechtecks ist, sind wir fertig, da alle Rechtecke nun unmöglich untergebracht werden können, ansonsten fangen wir wieder bei Schritt 3 mit dem neuen umgebenden Rechteck an.

1.3 Performance

Die Performance des Verfahrens hängt zum Einen von der Anzahl der Gärten ab, jedoch auch von der Differenz zwischen Gesamtlänge und maximaler Länge von einzelnen Rechtecken. Unabhängig davon braucht das Suchverfahren länger, je mehr Positionen für jedes Rechteck infrage kommen. Deshalb ist meine Einschätzung der Performance $P \approx O(\mathit{anzahl\ der\ gärten} * \mathit{gesamtlänge}^2 * \mathit{gesamtbreite})$. In der Umsetzung kommt es bei bis zu 10 Gärten mit Längen und Breiten kaum zu Performanceschwierigkeiten, hier könnte eine bessere Methode zur Bestimmung der optimialen Position der kleinen Rechtecke helfen.

2 Umsetzung

Das Verfahren wurde in Python 3 umgesetzt.

2.1 Darstellung im Speicher

Ein noch zu platzierendes Rechteck wird als Tupel zweier Zahlen dargestellt: Eine Länge und eine Breite. Ein platziertes Rechteck ist ein Tupel aus der Länge, der Breite, einer x-Position und einer y-Position. Somit ist eine Lösung ein Tupel mit einer Liste von platzierten Gärten, der benötigten Länge und Breite. Um den Quellcode lesbarer zu gestalten, definiere ich die Funktionen `r_laenge`, `r_breite`, `r_x`, und `r_y`, die die einzelnen Teile des Rechtecks zurückgeben. Um einfach schauen zu können, ob auf einer Position (x, y) schon ein Rechteck liegt, wird eine rechteckige Matrix mit den Werten 0 für *leer* und 1 für *voll* verwendet. Diese wird bei Änderung des umgebenden Rechtecks neu initialisiert und in allen Unterfunktionen von `anordnen` verwendet.

2.2 Teile des Programms

2.2.1 genuegend_platz

In der Funktion `genuegend_platz` wird geprüft, ob an der Position (x, y) genügend Platz für das gegebene Rechteck ist. Da die Rechtecke nach absteigender Länge platziert werden, reicht es, die äußeren Felder in der Matrix auf andere Rechtecke zu überprüfen. Dies ist der Fall, da es unmöglich ist, ein Rechteck mit größerer Länge in einem mit kleinerer Länge einzuschließen. Wichtig ist, zu überprüfen, ob auch noch genügend Platz im umgebenden Rechteck nach rechts und nach unten vorhanden ist.

2.2.2 eins_platzieren

In der Funktion `eins_platzieren` ist ein Rechteck und die Matrix der bereits platzierten Rechtecke gegeben. Die Funktion ruft `genuegend_platz` von oben nach unten und von links nach rechts auf. So wird immer die Position gefunden, bei der genügend Platz vorhanden ist, die so weit links liegt wie möglich. Falls es zwei dieser Positionen gibt, wählt dieses Vorgehen die aus, die weiter oben ist. Falls eine Position gefunden werden konnte, wählt es außerdem die, die sich weiter oben befindet. Wenn eine Position gefunden wurde, wird nun die Matrix so verändert, dass die neu belegten Felder *voll* sind. Daraufhin wird ein platziertes Rechteck zurückgegeben. Falls Keine Position gefunden werden konnte, wird `None` zurückgegeben.

2.2.3 anordnen

Der Funktion `anordnen` ist eine absteigend nach Länge sortierte Liste von zu platzierenden Rechtecken und eine Länge und Breite des umgebenden Rechtecks gegeben. Sie erstellt eine leere Matrix und ruft für alle gegebenen Rechtecke `eins_platzieren` auf. Als letztes wird berechnet, wie viel der gegebenen Breite tatsächlich benötigt wurde.

2.2.4 Auf Modulebene

Hinweis: Da aus der Aufgabe kein Eingabeformat hervorgeht, werden die Rechtecke im Programm verändert. Da Python nicht kompiliert werden muss, ist dies für den Benutzer zumutbar.

Auf Modulebene werden als Erstes die Rechtecke absteigend nach Höhe sortiert. Danach werden Grundmaße, wie die Maximallänge, Maximalbreite, Gesamtbreite und Gesamtflaeche bestimmt. Nach dem oben beschriebenen Verfahren werden die momentane Länge und Breite verändert die Lösung mit minimaler Fläche wird gespeichert. Zuletzt wird der Unterschied zwischen optimaler Fläche und Lösungsfläche bestimmt und mit den Positionen, Maßen und Flächen der einzelnen Gärten ausgegeben.

3 Optimierung

Um das Verfahren für große Rechtecke zu optimieren, führe ich einen *Faktor* ein. Dieser ist der größte gemeinsame Teiler aller Maße der Rechtecke, der über den euklidischen Algorithmus ermittelt wird. Aufgrund dieser Eigenschaft muss die Breite nicht um 1 verkleinert werden, sondern direkt um $faktor$, da man alle Maße durch $faktor$ teilen kann, dass Verfahren schneller durchführen kann und dann wieder alle Maße mit $faktor$ multiplizieren kann. Dies hilft bei Beispiel 4.5 weiter, jedoch nicht bei Beispiel 4.6.

Eine andere Möglichkeit, die Umsetzung zu optimieren wäre, statt einer Python-`list` etwas zu nehmen, was für jedes Bit einen Zustand speichern kann. Dadurch würden Speichernutzung und Zugriffsgeschwindigkeiten gesenkt werden (durch höhere Geschwindigkeit einer spezialisierten Lösungen und den Cache im Prozessor).

4 Beispiele

4.1 Triviales Beispiel: Kein Rechteck

Ein triviales Beispiel ohne ein Rechteck. Hier sollte das Ergebnis $0m^2$ groß sein. Die Änderung im Programm lautet hier:

```
1 rechtecke = []
```

Die Ausgabe:

```
$ python3 schrebergaerten.py
0 m x 0 m => 0 m^2, 0 m^2 verschwendet
```

4.2 Triviales Beispiel: Ein Rechteck

Ein triviales Beispiel mit nur einem $50m * 25m$-Rechteck. Hier sollte das Ergebnis $1250m^2$ groß sein. Die Änderung im Programm lautet hier:

```
rechtecke = [
    (50, 25)
]
```

Die Ausgabe:

```
$ python3 schrebergaerten.py
25 m x 50 m => 1250 m^2, 0 m^2 verschwendet
Garten 50 m x 25 m im Punkt (0, 0)
```

4.3 Triviales Beispiel: Vier Quadrate

Ein anderes Beispiel ist eines mit vier $25m * 25m$-Quadraten. Hier sollte das Ergebnis $2500m^2$ groß sein. Die Änderung im Programm lautet hier:

```
rechtecke = [
    (25, 25),
    (25, 25),
    (25, 25),
    (25, 25)
]
```

Die Ausgabe:

```
$ python3 schrebergaerten.py
100 m x 25 m => 2500 m^2, 0 m^2 verschwendet
Garten 25 m x 25 m im Punkt (0, 0)
Garten 25 m x 25 m im Punkt (25, 0)
Garten 25 m x 25 m im Punkt (50, 0)
Garten 25 m x 25 m im Punkt (75, 0)
```

4.4 Das Beispiel aus der Aufgabe

Es gibt vier Rechtecke:

- $25m * 15m$
- $15m * 30m$
- $15m * 25m$
- $25m * 20m$

Die Änderung im Programm lautet:

```
rechtecke = [
    (25, 15),
    (15, 30),
    (15, 25),
    (25, 20),
]
```

Die Ausgabe:

```
$ python3 schrebergaerten.py
65 m x 30 m => 1950 m^2, 250 m^2 verschwendet
Garten 25 m x 15 m im Punkt (0, 0)
Garten 25 m x 20 m im Punkt (15, 0)
Garten 15 m x 30 m im Punkt (35, 0)
Garten 15 m x 25 m im Punkt (35, 15)
```

Somit ist das Programm nur ein wenig schlechter als das Beispiel ($1950m^2$ vs $1650m^2$).

4.5 Ein größeres Beispiel, lösbar mithilfe des Faktors

```
rechtecke = [
    (700, 300),
    (500, 900),
    (350, 200),
    (600, 800),
    (50, 100),
    (500, 600)
]
```

Ausgabe (mit unmerklicher Verzögerung):

```
$ python3 schrebergaerten.py
1500 m x 1200 m => 1800000 m^2, 285000 m^2 verschwendet
Garten 700 m x 300 m im Punkt (0, 0)
Garten 600 m x 800 m im Punkt (300, 0)
Garten 500 m x 900 m im Punkt (0, 700)
Garten 500 m x 600 m im Punkt (900, 600)
Garten 350 m x 200 m im Punkt (1100, 0)
Garten 50 m x 100 m im Punkt (300, 600)
```

4.6 Ein größeres Beispiel, nicht lösbar mithilfe des Faktors

```
rechtecke = [
    (700, 299),
    (500, 900),
    (350, 200),
    (600, 800),
    (50, 100),
    (500, 600)
]
```

Ausgabe beendet nicht.

5 Quellcode (ausschnittsweise)

5.1 genuegend_platz

Die Funktion `genuegend_platz`, die prüft, ob in einem Punkt (x, y) genügend Platz für ein Rechteck mit der Länge `laenge` und der Breite `breite` ist. Dies wird mit der Matrix `matrix` gemacht:

```
def genuegend_platz(x, y, breite, laenge, matrix):
    """
    Schaut, ob an der Position (x;y) noch genuegend Platz fuer ein Rechteck der Groesse (
    breite;laenge) ist
    """

    # Ist das Umgebungsrechteck gross genug?
    if x + breite > len(matrix[0]) or y + laenge > len(matrix):
        return False
```

```
    current_x = x
    while current_x < x + breite:
        if matrix[y][current_x] != LEER or matrix[y + laenge - 1][current_x] != LEER:
            return False
        current_x += 1

    current_y = y
    while current_y < y + laenge:
        if matrix[current_y][x] != LEER or matrix[current_y][x + breite - 1] != LEER:
            return False
        current_y += 1

    return True
```

code/genuegend_platz.py

5.2 eins_platzieren.py

Die Funktion `eins_platzieren`, die die ideale Position für ein Rechteck der Länge `laenge` und der Breite `breite` findet. Dazu wird die Matrix `matrix` gebraucht, um zu überprüfen, welche Felder bereits belegt sind.

```
def eins_platzieren(breite, laenge, matrix):

    platziert = False

    # Einen Platz für das Rechteck finden
    for platziertes_x in range(len(matrix[0])):
        for platziertes_y in range(len(matrix)):
            if genuegend_platz(platziertes_x, platziertes_y, breite, laenge, matrix):
                platziert = True
                break
        if platziert:
            break

    if platziert:
        # Die Matrix wiederherstellen
        for x in range(platziertes_x, platziertes_x + breite):
            for y in range(platziertes_y, platziertes_y + laenge):
                matrix[y][x] = VOLL
    else:
        return None

    return (laenge, breite, platziertes_x, platziertes_y)
```

code/eins_platzieren.py

5.3 anordnen

Die Funktion `anordnen`, die versucht, eine Liste von Rechtecken `rechtecke` in einem Rechteck der Länge `laenge` und der Breite `breite` unterzubringen:

```
def anordnen(rechtecke, breite, laenge):
    matrix = [ [LEER for _ in range(breite)] for _ in range(laenge) ]
    loesung = []
    for r in rechtecke:
        platziert = eins_platzieren(r_breite(r), r_laenge(r), matrix)
        if platziert is not None:
            loesung.append(platziert)
        else:
            return None

    rechteck_ganz_weit_rechts = max(loesung, key=lambda r: r_x(r) + r_breite(r))
    maximal_benoetigte_breite = r_x(rechteck_ganz_weit_rechts) + r_breite(
    rechteck_ganz_weit_rechts)

    return (loesung, laenge, maximal_benoetigte_breite)
```

code/anordnen.py

5.4 Modulebene

Auf Modulebene wird versucht, die Rechtecke um ihren größten gemeinsamen Teiler herunterzuskalieren. Daraufhin wird **anordnen** so aufgerufen, dass ein optimales Ergebnis entsteht. Hier müssen auch die Rechtecke eingegeben werden:

```
if __name__ == "__main__":
    # Länge mal Breite
    rechtecke = [
        # Rechtecke hier einfügen, z.B.
        #(100, 100),
        #(100, 100),
        #(100, 100),
        #(100, 100),
     ]

    # Keine Rechtecke vorhanden
    if len(rechtecke) == 0:
        print("0 m x 0 m => 0 m^2, 0 m^2 verschwendet")
        exit()

    gesamtflaeche = 0
    for r in rechtecke:
        gesamtflaeche += r_laenge(r) * r_breite(r)

    faktor = r_laenge(rechtecke[0])
    for r in rechtecke:
        faktor = gcd(faktor, r_laenge(r))
        faktor = gcd(faktor, r_breite(r))


    rechtecke = list(map(lambda r: (r[0] // faktor, r[1] // faktor), rechtecke))
    sortierte_rechtecke = sorted(rechtecke, key=lambda x: r_laenge(x), reverse=True)

    # Breite aller Rechtecke zusammen
    gesamtbreite = 0
    for r in rechtecke:
        gesamtbreite += r_breite(r)
    # Breite, die das breiteste Rechteck hat
    maximalbreite = r_breite(max(rechtecke, key=lambda r: r_breite(r)))


    maximallaenge = r_laenge(sortierte_rechtecke[0])



    bestes_ergebnis = []

    momentane_laenge = maximallaenge
    momentane_breite = gesamtbreite

    while momentane_breite >= maximalbreite:
        ergebnis = anordnen(sortierte_rechtecke, momentane_breite, momentane_laenge)
        if ergebnis is not None:
            # Alle Anordnungen, die die gleicht Breite haben werden, überspringen
            momentane_breite = ergebnis[2] - 1

            # Ergebnis speichern
            if len(bestes_ergebnis) == 0 or bestes_ergebnis[1] * bestes_ergebnis[2] >
    ergebnis[1] * ergebnis[2]:
                bestes_ergebnis = ergebnis
        else:
            momentane_laenge += 1


    # Alles wieder um Faktor vergrößern
    bestes_ergebnis = (
        list(map(lambda r: tuple(map(lambda g: g*faktor, r)), bestes_ergebnis[0])),
        faktor * bestes_ergebnis[1],
        faktor * bestes_ergebnis[2]
    )


    loesungsflaeche = bestes_ergebnis[1] * bestes_ergebnis[2]
```

```
68
    print("%d m x %d m => %d m^2, %d m^2 verschwendet" % (bestes_ergebnis[2],
    bestes_ergebnis[1], loesungsflaeche, (loesungsflaeche - gesamtflaeche)))
70  for r in bestes_ergebnis[0]:
        print("Garten %d m x %d m im Punkt (%d, %d)" % (r_laenge(r), r_breite(r), r_x(r),
     r_y(r)))
```

code/modulebene.py

Literatur

[1] Richard E. Korf. „Rectangle Packing: Initial Results“. In: *ICAPS-03 Proceedings*. 2003.

[2] Matt Perdeck. *Fast Optimizing Rectangle Packing Algorithm for Building CSS Sprites*. URL: https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu%22.