Dokumentation

Lösungen von Christian Zdralek und Karl Welzel

25.09.2015

Junioraufgabe 1

Da die Aufgabe darin besteht, alle Rechtecke zu genehmigen, die sich mit keinem bereits genehmigten überscheiden, besteht hier die Hauptaufgabe darin zu überprüfen, ob sich zwei gegebene Rechtecke überschneiden.

Lösungsidee:

Anstatt zu betrachten, wann sich zwei Rechtecke überschneiden, soll betrachtet werden, wann sie dies nicht tun. Es gibt vier Fälle die unterschieden werden:

1. Das erste Rechteck liegt über dem zweiten.
   → Die y-Koordinate der unteren Kante vom ersten Rechteck ist größer als die y-Koordinate der oberen Kante vom zweitem.

2. Das erste Rechteck liegt unter dem zweiten.
   → Die y-Koordinate der oberen Kante vom ersten Rechteck ist kleiner als die y-Koordinate der unteren Kante vom zweitem.

3. Das erste Rechteck liegt links neben dem zweiten.
   → Die x-Koordinate der rechten Kante vom ersten Rechteck ist kleiner als die y-Koordinate der linken Kante vom zweitem.

4. Das erste Rechteck liegt rechts neben dem zweiten.
   → Die x-Koordinate der linken Kante vom ersten Rechteck ist größer als die y-Koordinate der rechten Kante vom zweitem.

Wenn einer dieser Fälle eintritt, dann überschneiden sich die Rechtecke nicht. Es ist klar, dass wenn das erste Rechteck nicht neben dem zweiten liegt, es zwangsläufig Überschneidungen gibt. Also überschneiden sich zwei Rechtecke genau dann, wenn keiner der vier Fälle eintritt.

Quelltext (jA1.py):

```
def overlapping(claim1, claim2):
    return not (
        claim[1] >= claim2[3] or #claim1 liegt über claim2
```

```
        claim[3] <= claim2[1] or #claim1 liegt unter claim2
        claim[2] <= claim2[0] or #claim1 liegt links neben claim2
        claim[0] >= claim2[2]) #claim1 liegt rechts neben claim2
```

Junioraufgabe 2

Lösungsidee:

Da das Ziel ist zu überprüfen, ob Kassiopeia alle weißen Felder erreichen kann, soll versucht werden ausgehend von eine Liste mit Feldern, die Kassiopeia erreichen kann zu erstellen, die am Anfang nur mit Kassiopeias Feld gefüllt ist und dann jeweils um alle angrenzenden Felder der Felder in dieser Liste erweitert wird, bis es keine weiteren gibt. Diese sind dann alle Felder die Kassiopeia erreichen kann und nun kann überprüft werden, ob dies alle weißen Felder sind.

Um diesen Vorgang etwas schneller zu machen werden die Felder in drei Stufen eingeteilt:

1. Neue Felder: alle Felder die in einem Schritt als weiße Nachbarfelder von überprüften Feldern gefunden worden
2. Überprüfte Felder: Felder, die von Kassiopeia aus erreichbar sind, deren Nachbarfelder aber noch nicht überprüft wurden
3. Vollstandig überprüfte Felder: Felder, die von Kassiopeia aus erreichbar sind und deren Nachbarfelder bereits überprüft wurden.

Das Feld von Kassiopeia wird am Anfang zu den überprüften Feldern gezählt und dann werden in jedem Schritt folgende Anweisungen ausgeführt:

1. Alle weißen Nachbarfelder von überprüften Feldern werden zu neuen Feldern.
2. Die überprüften Felder werden zu vollständig überprüften Feldern.
3. Alle neuen Felder, die nicht bereits vollständig überprüft wurden, werden zu überprüften Feldern.

Dies wird so lange durchgeführt bis es keine neuen Felder mehr gibt und damit alle weißen Felder, die Kassiopeia erreichen kann, zu den vollständig überprüften Feldern gehören.

Um diesen Vorgang besser zu visualisieren, sind hier einzelne Schritte für das Beispiel aus der Aufgabe dargestellt. Hier steht „Z“ für vollständig überprüfte und „A“ für überprüfte Felder. (Neue Felder sind nicht dargestellt, da sie nur als Zwischenstufe innerhalb eines Schrittes dienen und hier die Felder jeweils nach einem Schritt dargestellt werden.)

```
#########
#  #    #
#  # #  #
#  A #  #
#    #  #
#########
Schritt: 0
```

```
#########
#  #    #
#  # #  #
# AZA#  #
#  A #  #
#########
Schritt: 1
#########
#  #    #
# A#A#  #
#AZZZ#  #
# AZA#  #
#########
Schritt: 2
#########
# A#A   #
#AZ#Z#  #
#ZZZZ#  #
#AZZZ#  #
#########
Schritt: 3
...
#########
#ZZ#ZZZZ#
#ZZ#Z#ZZ#
#ZZZZ#ZZ#
#ZZZZ#ZZ#
#########
Schritt: 10
```

Beispiele:

kassiopeia1.txt Kassiopeia sind einige Felder versperrt.

kassiopeia2.txt Kassiopeia kann alle Felder erreichen.

kassiopeia3.txt Kassiopeia kann alle Felder erreichen.

kassiopeia4.txt Kassiopeia kann alle Felder erreichen.

kassiopeia5.txt Kassiopeia kann alle Felder erreichen.

kassiopeia6.txt Kassiopeia kann alle Felder erreichen.

kassiopeia7.txt Kassiopeia kann alle Felder erreichen.

Quelltext (jA2.py):

```
step = 0
first = True
while new_fields or first:
    first = False
    print_visualizing_fields(visualizing_fields, step)
    new_fields = set()
    for cell in checked_fields:
        new_fields |= find_neighbours(cell, fields)
    fully_checked_fields |= checked_fields
    checked_fields = new_fields-fully_checked_fields
    for field in fully_checked_fields:
        visualizing_fields[field[0]][field[1]] = "Z"
    for field in checked_fields:
        visualizing_fields[field[0]][field[1]] = "A"
    step += 1
```

Aufgabe 1

Lösungsidee:

Wir haben für die Lösung dieser Aufgabe einen rekursiven Lösungsansatz verwendet. Bei diesem ruft sich eine Funktion immer wieder selbst auf und verändert dabei nur die Parameter leicht. (https://de.wikipedia.org/wiki/F

Um einen Weg für Kassiopeia zu finden, bei dem sie jedes weiße Feld einmal betritt und die Felder F bereits betreten hat

- wird überprüft, ob F bereits alle weißen Felder beinhaltet, dann ist die Lösung gefunden und wird zurückgegeben.
- wird versucht für jedes Nachbarfeld des zuletzt betretenen Feldes, das nicht in F vorkommt (sonst würde Kassiopeia ein Feld doppelt betreten können) einen solchen Weg zu finden
- Wenn ein Weg gefunden wurde, dann ist die Lösung gefunden und wird zurückgegeben
- Wenn keine Weg gefunden wurde, dann wir dies als Ergebnis zurückgegeben

Beispiele:

kassiopeia1.txt Es gibt keinen Weg für Kassopeia jedes weiße Feld genau einmal zu betreten.

kassiopeia2.txt Es gibt keinen Weg für Kassopeia jedes weiße Feld genau einmal zu betreten.

kassiopeia3.txt WWSSOOOONOSONNWWW

kassiopeia4.txt Es gibt keinen Weg für Kassopeia jedes weiße Feld genau einmal zu betreten.

kassiopeia5.txt WWWWWWWWWWWW

kassiopeia6.txt Es gibt keinen Weg für Kassopeia jedes weiße Feld genau einmal zu betreten.

kassiopeia7.txt Es gibt keinen Weg für Kassopeia jedes weiße Feld genau einmal zu betreten.

Quelltext:

```
def find_path(previous_path):
    if all([field == ((i, j) in previous_path)
            for i, row in enumerate(fields)
            for j, field in enumerate(row)]):
        return previous_path
    for field in find_neighbours(previous_path[-1], fields):
        if field in previous_path:
            continue
        path = find_path(previous_path+[field])
        if path:
            return path
    return False
```

Aufgabe 4

Teilaufgabe 1.1:

Es gibt insgesamt für jedes Loch zwei Möglichkeiten: gestanzt oder nicht. Da es 23 Löcher gibt, die nicht schon vorgegeben wurden, gibt es für diese Einschränkung $2^{23} = 8.388.608$ verschieden Schlüsselkarten.

Teilaufgabe 1.2:

Hier werden die rechten zwei Spalten durch die linken bestimmt, deshalb gibt es noch 15 Löcher, die variiert werden können. Also gibt es für diese Methode $2^{15} = 32.768$ verschiedene Schlüsselkarten.

Teilaufgabe 2:

Lösungsidee:

Da es die Aufgabe ist, möglichst unterschiedliche Schlüsselkarten zu erzeugen, haben wir zunächst versucht festzulegen, was „unterschiedlich“ hier bedeutet und uns darauf geeinigt, dass unterschiedlich bedeutet, dass wenn man zwei Schlüsselkarten (in der gleichen Ausrichtung, es wurde nicht auf das Problem aus Teilaufgabe 1 eingegangen) vergleicht sich möglichst viele der einzelnen Stanz-Positionen darin unterscheiden, ob sie gestanzt wurden oder nicht. Wenn also bei der ersten Schlüsselkarte an jeder vorgesehenen Position ein Loch gestanzt wurde und bei der zweiten auch außer in der ersten

Reihe, dann unterscheiden sich 5 Positionen und die beiden Schlüsselkarten haben einen sogenannten „Abstand“ (Maßeinheit, die auch im Programm verwendet wird) von 5.

Da wir die Schlüsselkarten einzeln hintereinander erzeugen wollten, mussten wir nun ein Maß für den Abstand zwischen den schon vorhandenen Schlüssel und dem neuen. Dafür haben wird den „Mindestabstand“ eingeführt. Er gibt den kleinsten Abstand zwischen der neuen Schlüsselkarte und einer der schon vorhandenen Schlüsselkarte an.

Nun soll also eine Schlüsselkarte gefunden werden, die zu den schon vorhandenen einen möglichst großen Mindestabstand haben soll. Dafür muss es zunächst Schlüsselkarten geben, die als Ausgangspunkt genutzt werden. Dafür bieten sich die beiden Extreme „alle Positionen gestanzt“ und „keine Position gestanzt“ an, weil sie den größtmöglichen Abstand von 25 zueinander haben. Danach kann es natürlich keine weiteren Schlüsselkarten mit dem Mindestabstand von 25 geben, sondern nur noch maximal welche mit dem Mindestabstand von 12 (wenn man genau 12 oder 13 Positionen ausstanzt).

Nachdem die Vorbereitungen getan sind, haben wir uns folgendes Verfahren ausgedacht: Wenn eine neue Schlüsselkarte erzeugt werden soll, wird als Ausgangspunkt eine Schüsselkarte verwendet, die bei jeder Position mit dem Großteil der schon vorhandenen Schlüsselkarten übereinstimmt (wenn der Großteil an dieser Position ein Loch hat, hat sie es auch und umgekehrt). Dann werden die einzelnen Positionen mehrmals in zufälliger Reihenfolge durchlaufen und für jede Position die ähnlichste Schlüsselkarte (also die mit dem geringsten Abstand) gefunden und an dieser Position der Zustand der neuen Schlüsselkarte als das Gegenteil des Zustandes der ähnlichsten Schlüsselkarte gesetzt. Damit wird der Mindestabstand der neuen Schlüsselkarte erhöht, da sich die neue Schlüsselkarte im Vergleich zur ähnlichsten an dieser Position unterscheidet. Es kann passieren, dass die beiden Schlüsselkarte sich an dieser Position bereits unterschieden haben, dann ändert sich zwar der Mindestabstand nicht, aber es ändert sich die ähnlichste Schlüsselkarte auch nicht bis eine Änderung der neuen stattgefunden hat.

Bei diesem System wird der Mindestabstand einer Schlüsselkarte also immer weiter erhöht, aber es stellt sich die Frage ob der größtmögliche Mindestabstand für diese Schlüsselkarte bereits erreicht wurde. Dafür versucht das Programm die Schlüsselkarte so lange zu verbessern bis sie einen Mindestabstand hat, der genau so groß ist wie bei dem Schlüssel davor (anfangs natürlich 12 und nicht 25). Es bricht jedoch nach 100 Versuchen (hat sich in Tests als sinnvolle Größe herausgestellt) dies ab und findet einen Schlüssel, der einen um 1 verringerten Mindestabstand hat.

Es hat sich herausgestellt, dass wenn eine neue Schlüsselkarte einen Mindestabstand von z zu allen anderen hat, auch seine inverse Gegen-Schlüsselkarte einen solchen Mindestabstand von z hat. Wir haben also unseren Vorgang damit verbessert, dass jede zweite Schlüsselkarte als inverse der vorherigen generiert wird und da dies sehr schnell geht konnte damit die Geschwindigkeit fast halbiert werden.

Da die Schlüsselkarten auch ausgegeben werden sollen, haben wir uns hierfür eine einfach Textdarstellung ausgedacht: Ausgestanzte Positionen werden als „x“ dargestellt, nicht ausgestanzte als „.“

Beispiele:

- $N = 7$:
  Der Mindestabstand der letzten erzeugten Schlüsselkarte liegt bei 12

```
Schluesselkarte 1:
xxxxx
xxxxx
xxxxx
```

```
xxxxx
xxxxx
Mindestabstand: 25
Schluesselkarte 2:
.....
.....
.....
.....
.....
Mindestabstand: 25
Schluesselkarte 3:
x.x.x
x..xx
x..x.
.x..x
xx.x.
Mindestabstand: 12
Schluesselkarte 4:
.x.x.
.xx..
.xx.x
x.xx.
..x.x
Mindestabstand: 12
Schluesselkarte 5:
..x.x
xx...
xxxxx
..xxx
x....
Mindestabstand: 12
Schluesselkarte 6:
xx.x.
..xxx
.....
xx...
.xxxx
Mindestabstand: 12
Schluesselkarte 7:
.....
xxxxx
....x
xxxx.
x..x.
Mindestabstand: 12
```

- $N = 20$:

Der Mindestabstand der letzten erzeugten Schlüsselkarte liegt bei 11
Die erzeugten Schlüssel sind nicht die gleichen wie bei $N = 7$, weil bei der Erzeugung auch der Zufall eine Rolle spielt.

```
Schluesselkarte 1:
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
Mindestabstand: 25
Schluesselkarte 2:
.....
.....
.....
.....
.....
Mindestabstand: 25
Schluesselkarte 3:
x..xx
x.x.x
x.x..
..x..
x.xxx
Mindestabstand: 12
Schluesselkarte 4:
.xx..
.x.x.
.x.xx
xx.xx
.x...
Mindestabstand: 12
Schluesselkarte 5:
xx...
xx.xx
xx.x.
....x
..xxx
Mindestabstand: 12
Schluesselkarte 6:
..xxx
..x..
..x.x
xxxx.
xx...
Mindestabstand: 12
Schluesselkarte 7:
```

```
x.x.x
.....
xxxxx
x...x
x..xx
Mindestabstand: 12
Schluesselkarte 8:
.x.x.
xxxxx
.....
.xxx.
.xx..
Mindestabstand: 12
Schluesselkarte 9:
xxxx.
....x
.x..x
xxx..
..xxx
Mindestabstand: 12
Schluesselkarte 10:
....x
xxxx.
x.xx.
...xx
xx...
Mindestabstand: 12
Schluesselkarte 11:
xx..x
.x.xx
x...x
x.x..
xx..x
Mindestabstand: 12
Schluesselkarte 12:
..xx.
x.x..
.xxx.
.x.xx
..xx.
Mindestabstand: 12
Schluesselkarte 13:
..xx.
.x.x.
x.xx.
.xx..
.x.xx
```

```
Mindestabstand: 12
Schluesselkarte 14:
xx..x
x.x.x
.x..x
x..xx
x.x..
Mindestabstand: 12
Schluesselkarte 15:
.x.xx
...xx
xxx..
.x..x
xxx.x
Mindestabstand: 11
Schluesselkarte 16:
x.x..
xxx..
...xx
x.xx.
...x.
Mindestabstand: 11
Schluesselkarte 17:
x....
..xxx
x.xxx
xx...
..x..
Mindestabstand: 11
Schluesselkarte 18:
.xxxx
xx...
.x...
..xxx
xx.xx
Mindestabstand: 11
Schluesselkarte 19:
.xxxx
x..x.
..xxx
x....
..x.x
Mindestabstand: 11
Schluesselkarte 20:
x....
.xx.x
xx...
```

```
.xxxx
xx.x.
Mindestabstand: 11
```

- $N = 1000$:
  Der Mindestabstand der letzten erzeugten Schlüsselkarte liegt bei 6.

Quelltext (A4.py):

```
def next_key(keys, max_tries=100):
    if len(keys) == 2:
        target_difference = 12
    else:
        target_difference = min_key_difference(keys[-1], keys[:-1])

    r = [sum(key[i] for key in keys) > len(keys)/2. for i in range(25)]
    tries = 0
    while tries < max_tries and min_key_difference(r, keys) < target_difference \
            or min_key_difference(r, keys) < max(target_difference-1, 1):
        for i in shuffled(range(25)):
            sim_key = min([(key_difference(key, r), key) for key in keys])[1]
            r[i] = not sim_key[i]
        tries += 1
    return r
```