Schlüssellöcher

Kevin Schier

30. November 2015

1 Lösungsidee

Um zuerst die Fragen aus Teilaufgabe 1 zu beantworten: Wenn man zwei Löcher zur Erkennung der Ausrichtung nutzt, bleiben noch 23 Löcher zur freien Verwendung. Da jedes Loch quasi 1 Bit an Daten darstellt hat man so 23 Bit Daten. Damit lassen sich $2^{23} = 8388608$ verschiedene Schlüssel bilden.

Wenn man ein gespiegeltes Muster nutzt sind zwei Spalten exakte Kopien von zwei anderen. Somit bleiben drei Spalten frei übrig. Das sind 15 Löcher, d.h. es sind $2^{15} = 32768$ unterschiedliche Schlüssel möglich.

Mein Programm versucht allerdings keinen der beiden Ansätze. Es findet immer einen Schlüssel basierend auf den bisherigen. Allerdings sorgt es dafür, das weder der neue Schlüssel, noch seine gespiegelten Versionen mit einem der bisherigen Schlüssel oder dessen Spiegelungen übereinstimmen.

Um möglichst unterschiedliche Schlüssel generieren zu können, muss man erstmal Unterschied definieren. Der Unterschied zwischen zwei Schlüsseln ist der Anteil an Löchern (Bits) der von einem in den anderen Zustand gebracht (geflippt) werden muss. Zwei identische Schlüssel haben einen Unterschied von 0. Zwei möglichst unterschiedliche Schlüssel haben den Unterschied 1.

Um den Unterschied zu mehreren Schlüsseln zu kombinieren sollte man nicht den Durchschnitt der Werte nehmen. Es bietet sich an die Werte zu multiplizieren, da das einen Unterschied von 0 zu irgendeinem Schlüssel extrem straft. Außerdem wird ein möglichst großer Unterschied zu allen Schlüsseln wesentlich besser bewertet, als großer Unterschied zu einem Schlüssel und sonst kleine Unterschiede ($0{,}5 \cdot 0{,}5 \cdot 0{,}5 = 0{,}125 > 0{,}081 = 0{,}9 \cdot 0{,}3 \cdot 0{,}3$). Da bei vielen Schlüsseln die Produkte sehr klein werden, wird intern mit dem Logarithmus der Zahlen gearbeitet. Diese müssen dann eben addiert werden.

Es ist schwierig den Schlüssel zu finden der den absolut größten Unterschied zu allen anderen hat, da es $2^{25} = 33554432$ mögliche Kandidaten gibt. Daher versucht der Algorithmus nur einen möglichst guten Schlüssel anzunähern. Er erzeugt zuerst den Schlüssel, bei dem jeder Bit den momentan selteneren Wert bei allen anderen Schlüsseln und deren Spiegelungen hat, und der Unterschiedswert zu allen anderen Schlüsseln wird berechnet. Danach wird jeder Bit einmal geflippt und überprüft, ob der Unterschiedswert gestiegen ist. Wenn er nicht gestiegen ist, wird der entsprechende Bit wieder zurückgesetzt. Dieser Vorgang wird solange wiederholt, bis ein Maximum erreicht ist. Sollte das Maximum allerdings den Unterschied 0 haben, wird der Prozess mit einem zufälligen neuen Schlüssel wiederholt, da auf keinen Fall zwei gleiche Schlüssel vorkommen dürfen.

2 Umsetzung

Die Umsetzung findet in Visual-C++ 2015 statt.

Die Klasse Key implementiert einen quadratischen Schlüssel beliebiger Größe. Außerdem enthält sie die Bestimmung des Unterschieds zu einem anderen Schlüssel in Key::differenceToKey.

Die Umsetzung des Algorithmus zur Bestimmung neuer Schlüssel abhängig von vorhandenen Schlüsseln ist in der Methode Schlüssellöcher::CalculateNewKeys.

3 Beispiele

Listing 1: Erzeugung von sieben Schlüsseln.

```
Enter number of keys
> 7
11100
10101
11011
10001
11100

00000
01010
00100
01110
00000

11111
00000
00000
00000
11011

11011
11111
11111
11111
11111

00100
11101
11110
00000
00000

01011
00000
10001
11111
00100

00000
01111
00000
11001
11111

mean difference (with mirrored keys):0.569524
```

Die ersten paar Schlüssel haben die Tendenz symmetrisch zu sein, aber sehr schnell mischen sich asymmetrische mit ein, da symmetrische Schlüssel allein nicht viel Unterschied erlauben.

Listing 2: Erzeugung von 20 Schlüsseln.

```
Enter number of keys
> 20
01110
01101
11111
00010
00101

10001
00000
00000
10101
01010
```

```
00000
11011
00000
01010
10001

11111
11111
11111
11111
11111

01111
00100
10000
10000
00100

00000
11010
11111
00100
01010

10011
00000
01110
11011
10001

10100
10101
10000
01110
11110

11110
11011
01001
10101
10000

01010
01110
00100
11011
01110

10001
10100
11011
00000
11101

11110
01000
11100
01100
01001

00101
11101
11100
11001
00010

01000
00010
11001
10111
11101

10011
10011
01100
00110
00100

00110
11100
01010
10000
11011

11101
01010
10001
11010
11000

11010
```

```
10101
00101
10100
10111

00100
10000
11010
11111
00100

11001
11110
11100
00010
11010

mean difference (with mirrored keys):0.52
```

Listing 3: Erzeugung von 1000 Schlüsseln.

```
Enter number of keys
> 1000

[...]

mean difference (with mirrored keys):0.500302
```

Wie man sieht sinkt der durchschnittliche Unterschied nur sehr langsam bei großen Mengen an Schlüsseln. Und obwohl dieser Algorithmus gar nicht versucht perfekt unterschiedliche Schlüssel zu finden, ist der Unterschied generell sehr hoch.

4 Quelltext

Der Algorithmus.

```
double calculateKeyDifferenceFactor(const Schlüssellöcher::Key& key, const std::vector<Schlüssellöcher::Key>& otherKeys)
{
  double differenceFactor = 1.0;
  auto flipped = key.flipped();
  for (auto& otherKey : otherKeys)
  {
    double diff = key.differenceToKey(otherKey) * flipped.differenceToKey(otherKey);
    differenceFactor += (diff == 0 ? -std::numeric_limits<double>::infinity() : std::log(diff));
  }
  return differenceFactor;
}

std::vector<Schlüssellöcher::Key> Schlüssellöcher::Schlüssellöcher::CalculateNewKeys(uint32_t size, const std::vector<Key
    >& otherKeys, int numberOfKeys)
{
  int sizesq = size*size;
  std::vector<Key> result(numberOfKeys, Key(size));

  //add the keys and their mirrored versions to the internal vector
  std::vector<Key> keysWithMirrors(otherKeys.size() * 2);
  keysWithMirrors.reserve(otherKeys.size() * 2 + numberOfKeys * 2);
  auto& it = keysWithMirrors.begin();
  for (auto& key : otherKeys)
  {
    *it++ = key;
    *it++ = key.flipped();
  }

  //if there aren't any keys yet, create a random one
  int startPoint = 0;
  if (otherKeys.size() == 0)
  {
    result[0] = CreateRandomKey(size);
    keysWithMirrors.push_back(result[0]);
    keysWithMirrors.push_back(result[0].flipped());
    startPoint++;
```

```
    }
    for (int id = startPoint; id < numberOfKeys; id++)
    {

      //initial guess:
      //set bits to the least frequent ones
      for (int i = 0; i < sizesq; ++i)
      {
        int setCount = std::count_if(keysWithMirrors.begin(), keysWithMirrors.end(), [i](const Key& it)->bool {return it.
            bits[i] == true; });
        result[id].bits[i] = setCount < (keysWithMirrors.size() / 2);
      }

      //refine the guess by flipping bits and looking if the difference has increased
      double baseMeanDifference;
      bool anythingChanged = true;
      while (anythingChanged)
      {
        anythingChanged = false;
        baseMeanDifference = calculateKeyDifferenceFactor(result[id], keysWithMirrors);
        for (int i = 0; i < sizesq; ++i)
        {
          result[id].bits[i].flip();
          double newMeanDifference = calculateKeyDifferenceFactor(result[id], keysWithMirrors);
          if (newMeanDifference > baseMeanDifference)
          {
            baseMeanDifference = newMeanDifference;
            anythingChanged = true;
          }
          else
            result[id].bits[i].flip();
        }
        //randomize the key if it is still the same as some other one
        if (baseMeanDifference == -std::numeric_limits<double>::infinity())
        {
          result[id] = CreateRandomKey(size);
          anythingChanged = true;
        }
      }
      keysWithMirrors.push_back(result[id]);
      keysWithMirrors.push_back(result[id].flipped());
    }
    return result;
  }
```