Aufgabe 2: Kool

1 Lösungsidee

Ich habe mich dafür entschieden, dieses Problem mit einem heuristischen Algorithmus zu lösen. Durch eine Bewertung der Labyrinthe kann schnell und „gezielt“ nach einer Lösung gesucht werden.

Andere Algorithmen bringen einige Nachteile mit sich:

Ein Brute-Force-Algorithmus wäre hier nicht angebracht. Es würde einfach zu lange dauern alle Würfelplatzierungen zu durchlaufen.

Schon für die Würfelmenge in der Aufgabenstellung gibt es genau

$$4 \cdot \frac{n!}{l_1! \cdot l_2! \cdot l_3! \cdot \ldots \cdot l_k!} = 4 \cdot \frac{26!}{5! \cdot 4! \cdot 3! \cdot 3! \cdot 3! \cdot 3! \cdot 3! \cdot 2!} \approx 3{,}6 \cdot 10^{19}$$

Möglichkeiten die Würfel in dem Rechteck zu platzieren (Permutation von klassenweise äquivalenten Objekten). n steht für die Anzahl der Würfel. Die Würfel werden in k Gruppen von l_1, l_2, …, l_k gleichen Würfeltypen unterteilt. Der Bruch wird mit 4 multipliziert, weil es auch vier Orientierungsmöglichkeiten der Würfel gibt.

Ein Backtracking-Algorithmus müsste im schlimmsten Fall (nur eine mögliche Lösung) auch alle Möglichkeiten durchlaufen, weshalb auch solch ein Algorithmus für dieses Problem nicht brauchbar ist.

Optimale Rechtecke

Ein Rechteck der Größe b x h hat b Plätze in der Breite und h Plätze in der Höhe. Es hat also insgesamt $b \cdot h$ Plätze, in denen ein Würfel platziert werden kann. Rechtecke, die mehr Plätze als verfügbare Würfel (aus der Würfelmenge) haben, besitzen leere Plätze.

Rechtecke gelten als optimal, wenn die Anzahl der leeren Plätze kleiner als die Breite und kleiner als die Höhe des Rechtecks ist.

Das Rechteck aus der Aufgabenstellung hat die Größe 6x5 = 30 Würfelplätze. Die Beispiel-Würfelmenge hat eine Mächtigkeit von 26. Das Rechteck besitzt somit 26 von Würfeln besetzte Plätze und 4 leere Plätze. Das Rechteck ist aber optimal, weil die Anzahl der leeren Plätze (4) kleiner als die Breite (6) und kleiner als die Höhe (5) ist.

Algorithmus

Für jedes optimale Rechteck wird nach einem gültigen Labyrinth gesucht:

Als Erstes wird das Rechteck zufällig mit Würfeln aus der Würfelmenge gefüllt. Danach werden solange zwei zufällige Plätze des Rechtecks gewählt und deren Inhalte vertauscht, bis ein gültiges Labyrinth gefunden wurde. Hat man nach maximal N Vertauschungen kein gültiges Labyrinth gefunden, so stoppt man die Suche und nimmt sich das nächste Rechteck vor.

Nach jeder Vertauschung wird das Labyrinth bewertet. Hat es sich verbessert, hat es also weniger Fehlerpunkte als vorher, dann wird mit dem „verbesserten“ Labyrinth weitergearbeitet. Hat es mehr Fehlerpunkte als vorher, so wird die Vertauschung rückgängig gemacht.

Es werden also immer nur Verbesserungen des Labyrinths weiter betrachtet.

Hat man ein gültiges Labyrinth gefunden, wird die Suche nach weiteren gültigen Labyrinthen abgebrochen.

Wurden alle optimalen Rechtecke durchlaufen und trotzdem noch kein gültiges Labyrinth gefunden, so werden alle Würfel um 90 Grad im Uhrzeigersinn gedreht und die Suche beginnt erneut.

Nachdem man die Würfel auf diese Weise dreimal gedreht hat und immer noch kein gültiges Labyrinth gefunden wurde, endet der Algorithmus. Nun kann man entweder noch länger nach Labyrinthen suchen, also die maximale Anzahl der Vertauschungen (N) erhöhen oder aber es gibt wirklich kein gültiges Labyrinth für die eingelesene Würfelmenge.

Da der Algorithmus mit dem Zufall arbeitet, ist es nicht gewährleistet, dass der Algorithmus bei jedem Programmaufruf auch ein gültiges Labyrinth findet.

Bewertung eines Labyrinths

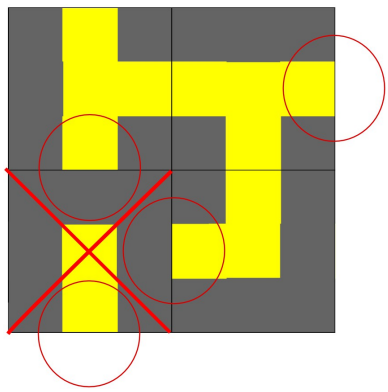
Fehlerhafte Verbindungen sind alle diejenigen Verbindungen, bei denen Sackgassen dadurch entstehen, dass ein Gang an der Wand eines anderen Würfels endet.

Für jede dieser fehlerhaften Verbindungen im Labyrinth gibt es einen Fehlerpunkt.

Der einzige Eingang des Labyrinths muss sich laut Aufgabenstellung oben links (der Raumecke gegenüberliegend) im Rechteck befinden. Fehlerpunkte gibt es deshalb für jeden weiteren Eingang, der sich dort nicht befindet. Genauso ist es ein Fehler, wenn der oberste linke Würfel gar keinen Eingang oder zwei Eingänge hat.

Zusätzlich gibt es für jeden Würfel, der nicht vom Eingang des Labyrinths erreichbar ist, einen Fehlerpunkt (selbst wenn diese eventuell richtig miteinander verbunden sind).

Somit hat folgendes Labyrinth 5 Fehlerpunkte (rot markiert):



Dieses Labyrinth besitzt zwei falsche Eingänge, zwei falsche Verbindungen und einen Würfel, der nicht vom „Haupteingang“ erreichbar ist.

Ein Labyrinth ist gültig, wenn es 0 Fehlerpunkte hat.

Lösbarkeit

Kann zu einer gegebenen Würfelmenge ein gültiges Labyrinth gefunden werden, so ist die Würfelmenge lösbar.

Würfel mit einer ungeraden Anzahl an Eingängen sind T-Kreuzungen und Sackgassen. Die anderen Würfeltypen haben eine gerade Anzahl an Eingängen.

Zählt man die Eingänge aller Würfel zusammen und ist diese Anzahl gerade, so ist das Labyrinth nicht lösbar. Die notwendige Bedingung ist also, dass die Anzahl der Eingänge aller Würfel ungerade sein muss. Eine hinreichende Bedingung konnte ich nicht erkennen. Nur ein Brute-Force-Algorithmus kann zu 100% sagen, ob eine gegebene Würfelmenge lösbar ist oder nicht.

Auch wenn die Würfelmenge nur aus einem Würfeltyp besteht, ist es nicht möglich ein gültiges Labyrinth zu finden.

Nutzungsgrenzen

Die minimale Größe eines Würfels schätze ich auf einen Meter. So wäre für ein Kind genug Platz, um im Gang des Würfels herumlaufen zu können (z.B. 20cm Wandbreite und 60cm Gangbreite). Ein sehr großer Kindergarten hat eventuell einen Raum von 20x20m für ein Kool-Labyrinth zur Verfügung.

Das Programm sollte also auch bei Würfelmengen mit 400 Würfeln möglichst schnell zu einer Lösung kommen (gemessene Laufzeiten: siehe Programm-Ablaufprotokoll). Bei einem Labyrinth dieser Größe ist es jedoch wahrscheinlicher, dass sich die Kinder darin verlaufen und dann Angst bekommen.

Erweiterungen

Parallelisierung

Die Suche nach einer Lösung für ein bestimmtes optimales Rechteck habe ich parallelisiert. So verringert sich die benötigte Rechenzeit erheblich. Die optimalen Rechtecke werden parallel gelöst. Es wird für jedes optimale Rechteck ein Thread gestartet, der nach einem gültigen Labyrinth sucht. Den Standardwert für die maximale Anzahl der Threads habe ich auf die Anzahl der Prozessoren des Computers (auf dem das Programm läuft) festgelegt. Die maximale Anzahl der Threads kann jedoch auch durch den Benutzer verändert werden.

Rotation

Die Einschränkung durch die Farbmarkierung kann durch den Benutzer aufgehoben werden. Denn in der Realität wären die Würfel wahrscheinlich so konzipiert worden, dass man diese auch einzeln drehen kann, um neue Labyrinthe zu erhalten.

Ist die Rotation erlaubt, so ändert sich der Algorithmus nur geringfügig: zu einer 50-50-Wahrscheinlichkeit werden entweder zwei zufällig gewählte Würfel vertauscht oder ein zufällig gewählter Würfel gedreht.

Generator

Falls sich die Mitarbeiter des Kindergartens Würfel bestellen wollen, so können die Mitarbeiter den von mir implementierten Labyrinth-Generator benutzen. Die Größe des zu erzeugenden Labyrinths ist frei wählbar. Die generierte Würfelmenge wird in eine Textdatei gespeichert.

Bildausgabe

Sowohl die vom Labyrinth-Generator erzeugten, als auch die gefundenen gültigen Labyrinthe werden als png-Bild visualisiert und abgespeichert.

Vereinfachte Würfelmengeneingabe

Falls die Mitarbeiter des Kindergartens eine spezielle Würfelmenge testen wollen, so können sie die Würfelmenge über ein graphisches Tool eingeben und müssen so nicht das (intern von mir verwendete) Datei-Format einer Würfelmenge kennen. Stattdessen genügt ein Klick auf einen der graphisch veranschaulichten Würfeltypen und anschließend gibt man die Anzahl ein.

Suche nach einem bestimmten Rechteck

Eine Suche nach einem Labyrinth bestimmter Größe ist auch möglich. So können die Mitarbeiter des Kindergartens gleich nach einem Labyrinth suchen, dass für den Raum dort passt. Wenn sie nämlich schon im Voraus wissen, dass ein 3x5 nicht in den Raum passen wird, so können sie gleich nach einem 4x4 Rechteck suchen lassen.

Art der Sackgassen

Falls die Mitarbeiter des Kindergartens eventuell doch Sackgassen haben möchten, die nicht mit Hilfe der Sackgassenwürfel entstanden sind, dann kann mein Programm auch diesen Wunsch erfüllen. In den meisten Fällen werden diese Art von gültigen Labyrinthen schneller gefunden, da dadurch die Zahl der möglichen gültigen Labyrinthe steigt.

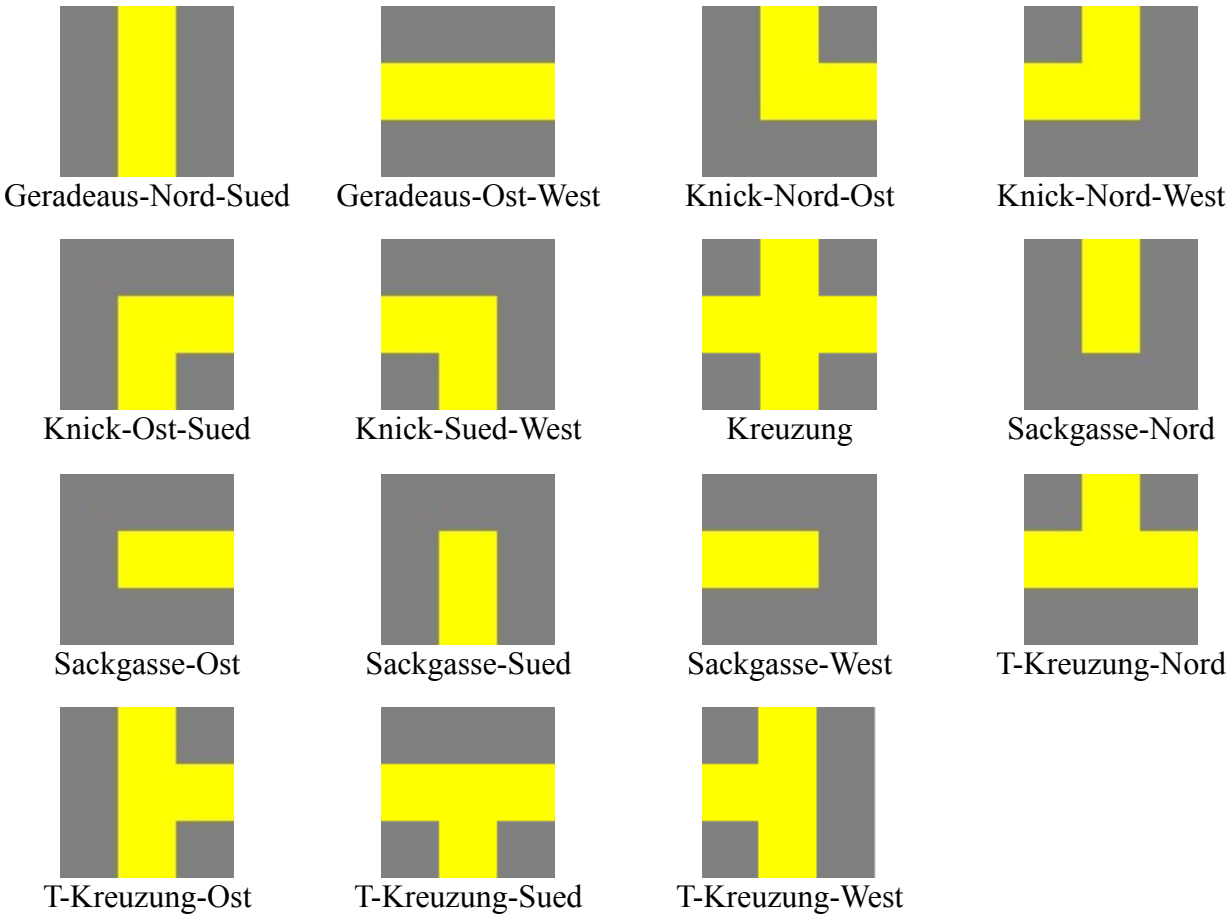
2 Programm-Dokumentation

Die Lösungsidee wurde von mir in der Programmiersprache C und unter Linux umgesetzt. Zur png-Bildausgabe wurde die GD Graphics Library[1] benutzt.

Dem Programm wird eine Datei mit der darin enthaltenen Würfelmenge als Parameter übergeben. Für die Würfelmenge aus der Aufgabenstellung sieht die Datei folgendermaßen aus:

```
3 Geradeaus-Ost-West
5 Knick-Nord-West
3 Knick-Nord-Ost
2 Knick-Ost-Sued
3 Knick-Sued-West
1 T-kreuzung-West
1 T-kreuzung-Nord
3 T-kreuzung-Ost
3 T-kreuzung-Sued
1 Kreuzung
1 Sackgasse-Sued
```

Da kein Eingabeformat in der Aufgabestellung vorgegeben wurde, habe ich mir folgende Namen für die Würfeltypen ausgedacht:



1 http://www.boutell.com/gd/

Mit der Funktion **`load_labyrinth`** wird die Würfelmenge eingelesen. Diese Funktion erstellt eine neue **`labyrinth`**-Struktur und fügt jeden Würfel einzeln zu dem Würfel-Array **`cubes`** der **`labyrinth`**-Struktur hinzu. Das Würfel-Array der **`labyrinth`**-Struktur besteht aus **`cube`**-Strukturen. Eine **`cube`**-Struktur besteht aus fünf **`char`**-Variablen (**`north`**, **`east`**, **`south`**, **`west`**, **`type`**), die für die vier möglichen Gänge und dessen Typ stehen.

Da man nach dem Einlesen der Würfelmenge die Anzahl der Würfel kennt, kann man anschließend die Größen der optimalen Rechtecke berechnen. Die Größen werden in einem Array gespeichert.

Die Suche nach einem gültigen Labyrinth übernimmt die Funktion **`labyrinth_solve`**. Dieser Funktionen wird die Würfelmenge, die Dimension des zu befüllenden Rechtecks und Parameter, die den Algorithmus beeinflussen (z.B. maximale Anzahl der Vertauschungen N), übergeben. Diese Informationen sind in einer **`labyrinth`**-Struktur gespeichert.

Die Funktion **`labyrinth_solve`** erstellt zunächst aus den übergebenen Rechteckdimensionen ein Rechteck (**`rectangle`**-Struktur). Ein Rechteck besteht aus einem zweidimensionales Array, in dem die Würfel „platziert“ werden können.

Dann beginnt der in der Lösungsidee beschriebene Algorithmus:

Das Rechteck wird durch die Funktion **`rectangle_fill`** mit einer zufälligen Anordnung der Würfel befüllt. Dann werden N-mal zwei zufällige Plätze des Rechtecks gewählt und deren Inhalt vertauscht. Nach jedem Tauschen wird das Labyrinth bewertet und geprüft, ob es sich verbessert oder verschlechtert hat. Hat es sich verschlechtert, dann wird die Vertauschung wieder rückgängig gemacht.

Das Bewerten eines Labyrinths übernimmt die Funktion **`rectangle_evaluate`**: Jeder Würfel wird geprüft, ob er fehlerhafte Verbindungen zu anderen Würfel besitzt und ob er nicht erlaubte Eingänge an den Rändern des Labyrinths bildet. Die Funktion **`recursive_check`** zählt, wie viele Würfel vom „Haupteingang“ erreichbar sind. Die Anzahl der nicht erreichbaren Würfel lässt sich dann mit *Anzahl Würfel insgesamt – Anzahl erreichbarer Würfel* errechnen. Aus diesen Informationen wird die Anzahl der Fehler berechnet. Die Funktion **`rectangle_evaluate`** bricht die Suche nach Fehlern ab, sobald das Labyrinth mehr Fehler besitzt, als es vor der Vertauschung hatte. Dadurch wird sehr viel Rechenzeit eingespart.

Sobald ein Labyrinth mit 0 Fehlern gefunden wurde, wird die aktuelle Anordnung der Würfel als png-Bild abgespeichert. Das „Zeichnen“ übernimmt die Funktion **`labyrinth2png`**.

Parallelität

Zur Parallelisierung kamen POSIX Threads (glibc) zum Einsatz. Die optimalen Rechtecke werden auf Threads verteilt, die dann die Funktion **`labyrinth_solve`** aufrufen.

Generator

Die Labyrinth-Generator-Funktion kann durch den Parameter **`-e <Breite> <Höhe>`** benutzt werden. Die Funktion **`labyrinth_generate`** erzeugt ein zufälliges Labyrinth mit Hilfe der rekursiven Funktion **`random_fill_rectangle`**.

Das so erzeugte Labyrinth wird als png-Bild und die Würfelmenge als Textdatei abgespeichert.

Vereinfachte Würfelmengeneingabe

Für die Eingabe der Würfelmenge habe ich ein kleines Tool in Java geschrieben. Es wird ein

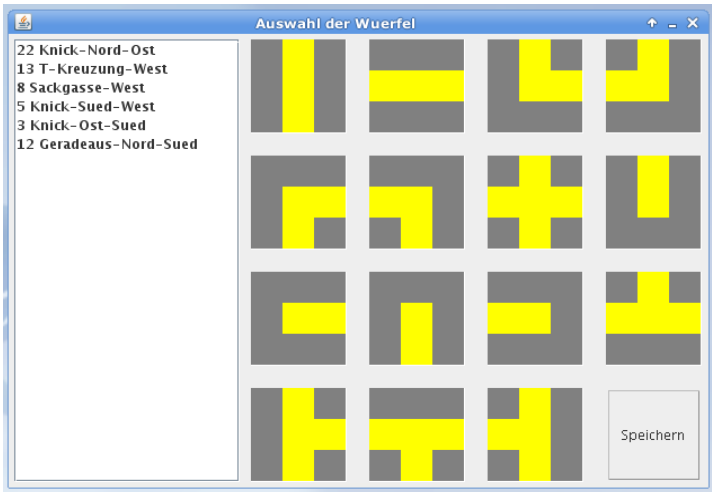
simpler **`JFrame`** erzeugt, der auf der rechten Seite die Würfeltypen und auf der linken Seite eine Liste der eingegebenen Würfel anzeigt.

Klickt man rechts auf einen Würfeltyp, dann öffnet sich ein Dialogfenster, in dem man die Anzahl der Würfel eingibt. In der Liste auf der linken Seite aktualisiert sich dann die Würfelmenge.

Hat man alle Würfel eingegeben, dann kann man die Würfelmenge in einer Datei abspeichern, indem man auf den Button „Speichern“ klickt.

Falls man sich bei der Eingabe vertan hat, dann kann man durch einen Rechtsklick auf einen der Würfel die Würfelmenge löschen.

Das Programm kann mit **`java Main`** gestartet werden und hat folgendes Layout:



3 Programm-Ablaufprotokoll

Durch den Parameter **--help** erhält man eine Auflistung aller möglichen Parameter für dieses Programm:

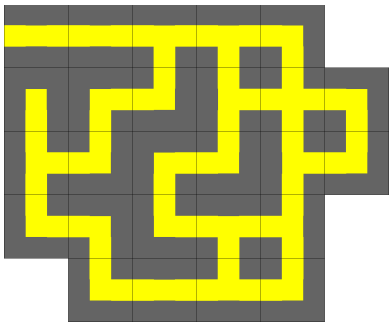
```
$ ./kool --help

kool [OPTIONEN] <Dateiname>

moegliche Optionen:
  -r                    Rotation erlauben

  -g                    erlaube auch Sackgassen, die
                        nicht aus Sackgassenwuerfeln bestehen

  -a                    durchlaufe alle optimalen Rechtecke

  -e <Breite> <Hoehe>   erzeuge zufaelliges Labyrinth

  -s <Wert>             Anzahl Schritte festlegen
                        (Standardwert: 3000000)

  -c <Anzahl>           Anzahl der Threads festlegen

Bestimmtes Labyrinth suchen:
  -b <Breite>           Breite festlegen
  -h <Hoehe>            Hoehe festlegen
```

Die Würfelmenge aus der Aufgabenstellung habe ich in der Datei **beispiel.lab** abgespeichert. Zum Ermitteln der Kool-Labyrinthe startet man das Programm mit dem Dateinamen als Parameter:

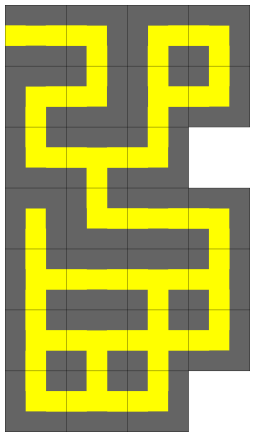
```
$ time ./kool beispiel.lab
suche    2 x 13   Labyrinth
suche    4 x 7    Labyrinth
suche    5 x 6    Labyrinth
suche    9 x 3    Labyrinth
suche    3 x 9    Labyrinth
suche    1 x 26   Labyrinth
suche    6 x 5    Labyrinth
suche   13 x 2    Labyrinth
gefundenes Labyrinth wurde in 4x7.png gespeichert.
gefundenes Labyrinth wurde in 9x3.png gespeichert.

real   0m14.896s
user   0m44.423s
sys    0m23.852s
```

Hier wurden zwei (weil 8-Prozessoren) gültige Kool-Labyrinthe in ca. 15 Sekunden gefunden und als Bild abgespeichert.

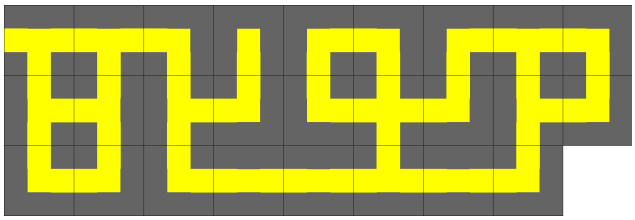
So sehen einige von den gefundenen Labyrinthen aus:



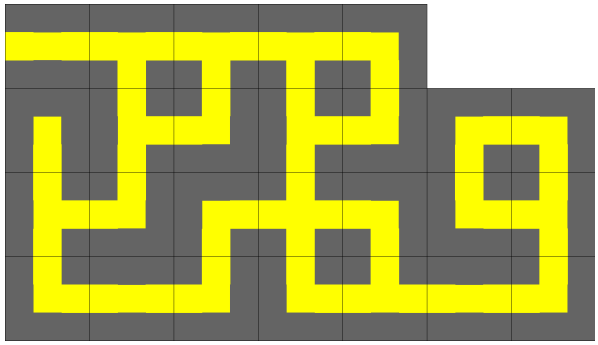
6x5



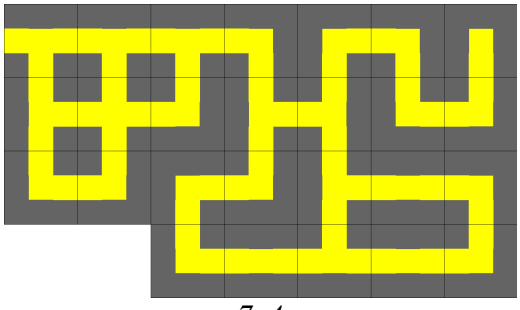
4x7



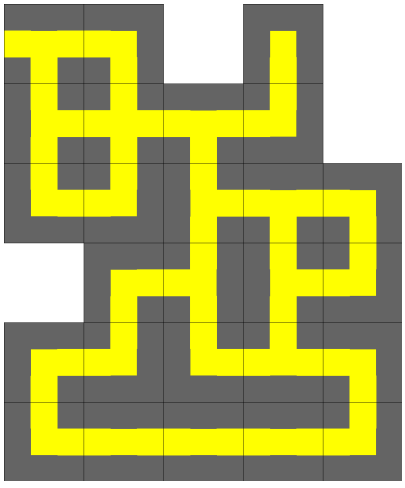
9x3



7x5



7x4

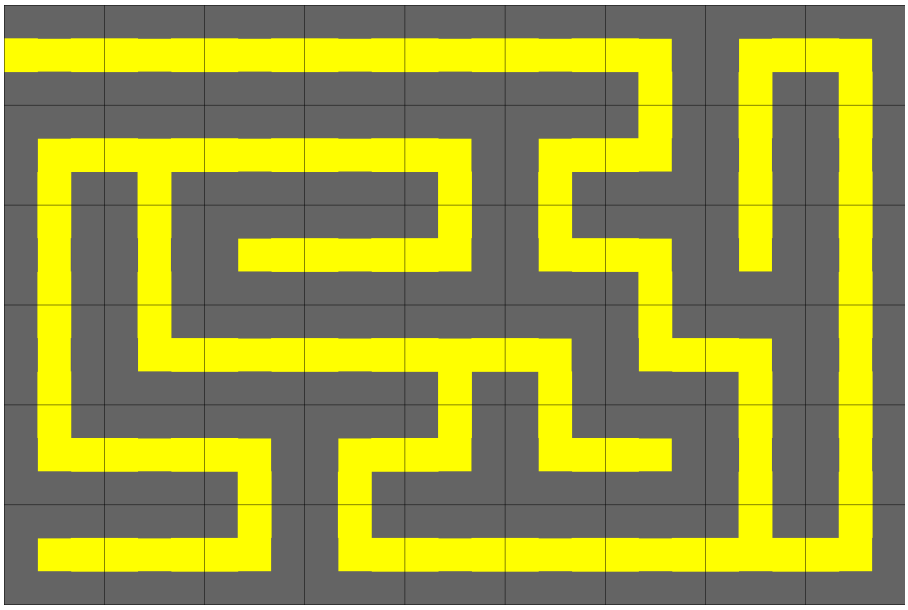


5x6

Möchten die Mitarbeiter des Kindergartens ein Kool-Labyrinth kaufen, was auf eine Fläche von 9x6 Würfeln passt, so können sie ein zufälliges Labyrinth mit folgendem Programmaufruf erzeugen:

```
$ ./kool -e 9 6
Erzeugtes Labyrinth wurde in zufall.lab und zufall.png gespeichert.
```

Das erzeugte Labyrinth sieht dann beispielsweise so aus:



Die Würfelmenge wurde in der Datei **`zufall.lab`** abgespeichert:

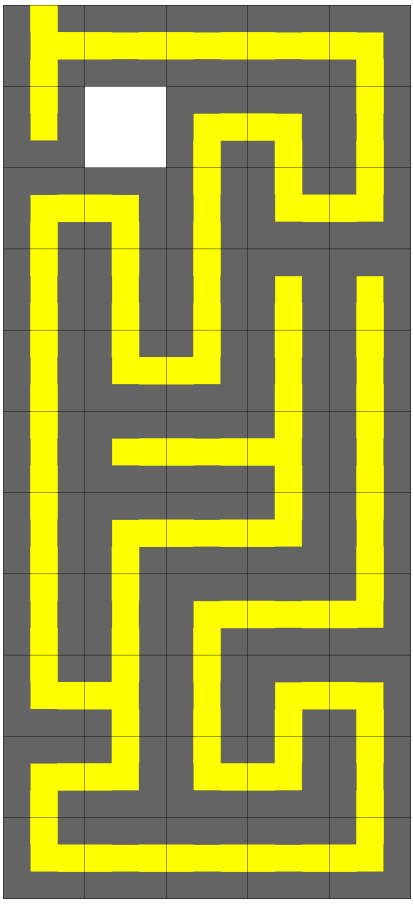
```
16 geradeaus-ost-west
7 knick-sued-west
4 knick-ost-sued
2 t-kreuzung-sued
5 knick-nord-west
9 geradeaus-nord-sued
2 sackgasse-ost
6 knick-nord-ost
1 sackgasse-nord
1 sackgasse-west
1 t-kreuzung-nord
```

Diese Textdatei kann man dem Programm wieder übergeben, um so andere „Labyrinth-Varianten“ aus der Würfelmenge zu erhalten:

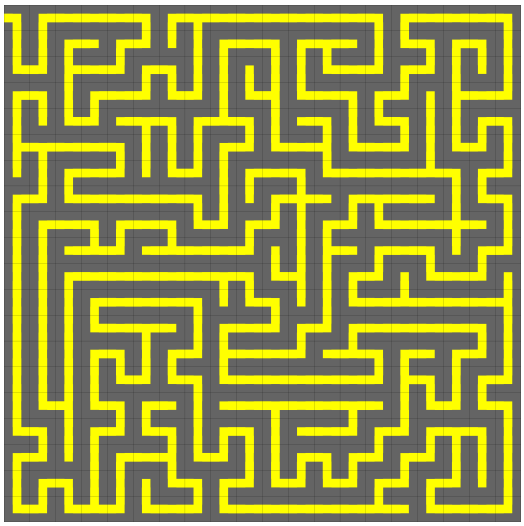
```
$ ./kool zufall.lab
suche   54 x 1    Labyrinth
suche   27 x 2    Labyrinth
suche   11 x 5    Labyrinth
suche   14 x 4    Labyrinth
suche    9 x 6    Labyrinth
suche    8 x 7    Labyrinth
suche   18 x 3    Labyrinth
suche    1 x 54   Labyrinth
suche    2 x 27   Labyrinth
suche    3 x 18   Labyrinth
suche    4 x 14   Labyrinth
```

```
suche    5 x 11   Labyrinth
suche    6 x 9    Labyrinth
suche    7 x 8    Labyrinth

alle Wuerfel wurden um 90 Grad gedreht

suche   54 x 1    Labyrinth
suche   18 x 3    Labyrinth
suche   14 x 4    Labyrinth
suche   27 x 2    Labyrinth
suche   11 x 5    Labyrinth
suche    9 x 6    Labyrinth
suche    8 x 7    Labyrinth
suche    1 x 54   Labyrinth
suche    2 x 27   Labyrinth
suche    3 x 18   Labyrinth
suche    4 x 14   Labyrinth
suche    6 x 9    Labyrinth
suche    5 x 11   Labyrinth
gefundenes Labyrinth wurde in 5x11.png gespeichert.
```

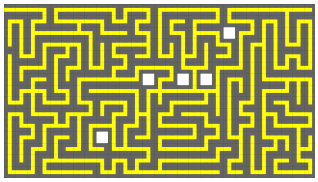
In diesem Fall wurde erst ein Labyrinth gefunden, nachdem alle Würfel um 90 Grad gedreht wurden. Nach einer Minute wurde folgendes gültiges Labyrinth gefunden:



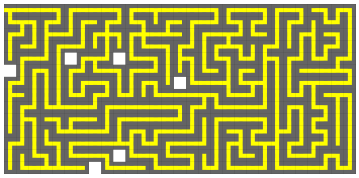
Härtetest für den Algorithmus sind große Würfelmengen mit z.B. 400 Würfeln (z.B. mit **`./kool -e 20 20`** erzeugt):



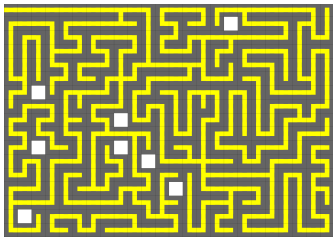
Innerhalb von einer Minute wurden drei Kool-Labyrinthe mit der selben Würfelmenge gefunden:



27x15



29x14



24x17

Noch größere Labyrinthe können dank Parallelisierung schnell gelöst werden: Für eine Würfelmenge mit 900 Würfeln brauchte das Programm 8 Minuten um ein gültiges Labyrinth zu finden. Bei einer so großen Würfelmenge muss jedoch die maximale Anzahl der Vertauschungen (*N*) erhöht werden, damit überhaupt Labyrinthe gefunden werden. In diesem Fall musste man die Anzahl *N* auf 10 Millionen Schritte setzen:

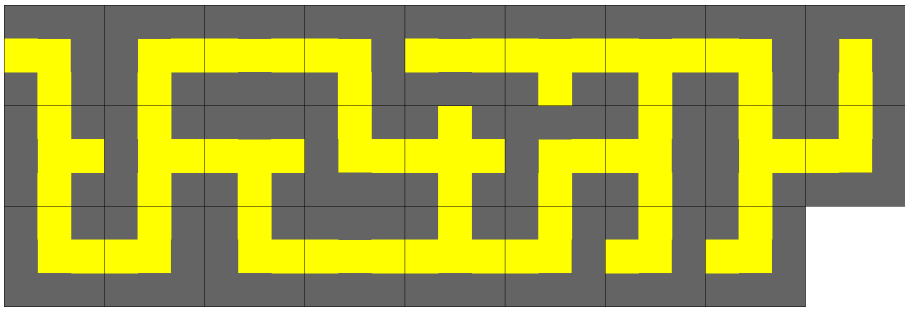
```
$ ./kool 900wuerfel.lab -s 10000000
...
```

Die Rotation kann mit dem Parameter **-r** erlaubt werden:

```
$ ./kool beispiel.lab -r
suche   26 x 1    Labyrinth
suche   13 x 2    Labyrinth
suche    9 x 3    Labyrinth
suche    7 x 4    Labyrinth
suche    6 x 5    Labyrinth
suche    2 x 13   Labyrinth
suche    3 x 9    Labyrinth
suche    1 x 26   Labyrinth
gefundenes Labyrinth wurde in 7x4.png gespeichert.
gefundenes Labyrinth wurde in 6x5.png gespeichert.
gefundenes Labyrinth wurde in 3x9.png gespeichert.
gefundenes Labyrinth wurde in 9x3.png gespeichert.
```

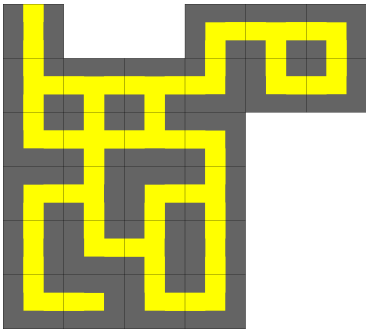
Dadurch können natürlich viel mehr Labyrinthe gefunden werden, weil es einfach eine große Einschränkung ist, wenn die Farbmarkierungen alle in eine Richtung zeigen müssen.

Möchte man auch Sackgassen zulassen, die nicht in Sackgassenwürfeln enden, so wird der Parameter **-g** angegeben. Auch dadurch wird die Suche nach Labyrinthen erheblich vereinfacht, nur sehen die Labyrinthe dann ein wenig gewöhnungsbedürftig aus:



Soll nach einem Labyrinth mit einer bestimmten Größe gesucht werden, kann man dies mit den Parametern **-b <Breite>** und **-h <Höhe>** machen:

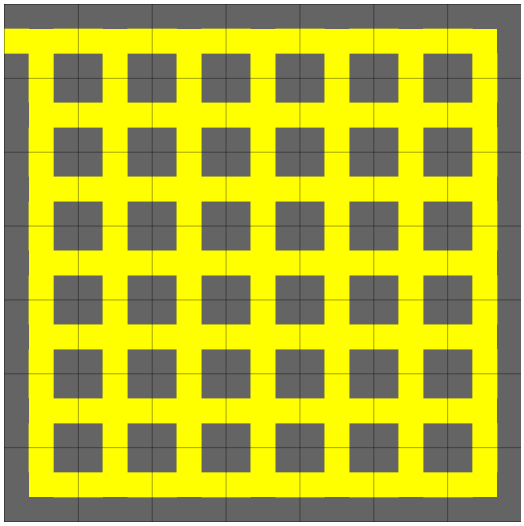
```
$ ./kool -b 6 -h 6 beispiel.lab
suche    6 x 6    Labyrinth

alle Wuerfel wurden um 90 Grad gedreht

suche    6 x 6    Labyrinth
gefundenes Labyrinth wurde in 6x6.png gespeichert.
```



Sonderfälle

Triviale Fälle wie z.B. solch ein Labyrinth (**`simpel.lab`**):

löst mein Programm genauso gut, wie Würfelmengen, wo es nur eine mögliche Lösung gibt (**`eineloesung.lab`**):



Wenn die Würfelmenge die notwendige Bedingung nicht erfüllt, dann gibt das Programm folgende Fehlermeldung aus:

```
$ ./kool nurkreuzungen.lab
Wuerfel koennen nicht zu einem gueltigen Labyrinth zusammengefuegt werden
```

Ist es nicht möglich aus einer Würfelmenge ein gültiges Labyrinth zu finden, dann gibt das Programm Folgendes aus:

```
$ ./kool keineloesung.lab
suche    6 x 1    Labyrinth
suche    2 x 3    Labyrinth
suche    1 x 6    Labyrinth
suche    3 x 2    Labyrinth

alle Wuerfel wurden um 90 Grad gedreht

suche    6 x 1    Labyrinth
suche    1 x 6    Labyrinth
suche    3 x 2    Labyrinth
suche    2 x 3    Labyrinth

alle Wuerfel wurden um 90 Grad gedreht

suche    6 x 1    Labyrinth
suche    3 x 2    Labyrinth
suche    1 x 6    Labyrinth
suche    2 x 3    Labyrinth

alle Wuerfel wurden um 90 Grad gedreht

suche    6 x 1    Labyrinth
suche    1 x 6    Labyrinth
suche    3 x 2    Labyrinth
suche    2 x 3    Labyrinth
Es konnte kein gueltiges Labyrinth gefunden werden.
```

In solchen Fällen ist die Würfelmenge mit großer Wahrscheinlichkeit wirklich nicht lösbar. Möchte man es trotzdem weiter versuchen, dann könnte man entweder *N* erhöhen (Parameter **`-s`**) oder ein paar Einschränkungen mit den Parametern **`-r`** und/oder **`-g`** aufheben.