Universität Bremen

Praktische Informatik 3 WS 18/19

Christoph Lüth
Thomas Barkowsky
Andreas Kästner
Gerrit Marquardt
Tobias Haslop
Matz Habermann
Berthold Hoffmann

10. Übungsblatt

Ausgabe: 15.01.19
Abgabe: 29.01.19 12:00

Wir befinden uns in Kreta vor gut 3000 Jahren. Minos, König von Kreta, hat ein Problem: nach einer Indiskretion seiner Gattin mit einem Stier muss er ein menschenfressendes Ungeheuer, den Minotaurus (halb Mensch, halb Stier, sehr stark und mit großem Appetit), sicher unterbringen. Er ließ dazu das erste bekannte Labyrinth erbauen — ein Irrgarten, aus dem sich nur schwer der Ausweg finden läßt.

Wir sehen also: Labyrinthe sind von großem Nutzen, ob zur sicheren Unterbringung von Ungeheuern, in der Spieleindustrie, oder einfach nur weil sie nett anzusehen sind. In diesem Übungsblatt wollen wir deshalb Labyrinthe erstellen.

Konzeptionell ist ein Labyrinth der aufspannende Baum eines zu Grunde liegenden Graphen, der alle denkbaren Übergänge darstellt; aufspannend heißt, dass es von einer Wurzel (dem Eingang) einen Pfad zu jedem Knoten des Graphen gibt. Der Einfachheit halber stellen wir den aufspannenden Baum auch als Graphen dar, das vereinfacht die Visualisierung. Zur Verbesserung der Vermarktbarkeit integrieren wir die Grapherzeugung in ein Webinterface; so ist auch eine portable Anzeige per SVG leicht implementiert.

Unser Program besteht aus vier Teilen: ein einfachen Modul für Graphen, ein Modul zur Erzeugung und zum Anzeigen von Labyrinthen, und der Webschnittstelle.

10.1 *Noblesse oblige: Graphen* — *5 Punkte*

Es gibt für Haskell diverse Graph-Büchereien, aber wir entwickeln aus didaktischen Gründen unsere eigene. Das Module Graph implementiert *ungerichtete* Graphen als Datentyp Graph v generisch über den Kanten v, die angeordnet sein müssen (Ord v). Die Operationen sind wie folgt:

- Mit addEdge und remEdge können Kanten hinzugefügt und gelöscht werden. Ist bei addEdge Quelle oder Ziel der Kante nicht im Graphen enthalten, wird der Graph unverändert zurückgegeben.
- Mit addVertex und remVertex werden Knoten hinzugefügt oder (mitsamt der eingehenden Kanten) gelöscht.
- Mit hasEdge kann gefragt werden, ob es zwischen zwei Knoten eine Kante gibt, und neighbours gibt die Menge aller von einem Knoten erreichbaren Knoten zurück.
- empty ist der leere Graph, und discrete diskretisiert einen Graphen, indem alle Kanten entfernt werden.
- fromList erzeugt einen Graphen aus einer Liste von Kanten (implizit werden die darin enthaltenen Knoten hinzugefügt), und toList konvertiert einen Graphen in eine derartige Liste.

Man beachte, dass der Graph ungerichtet ist, also muss es für jede Kante (v,w) auch immer eine Kante (w,v) geben. In Haskell ist die Signatur der Schnittstelle ist wie folgt:

```
data Graph v

addVertex   :: Ord v⇒ Graph v→ v→ Graph v
remVertex   :: Ord v⇒ Graph v→ v→ Graph v
addEdge     :: Ord v⇒ Graph v→ v→ v→ Graph v
remEdge     :: Ord v⇒ Graph v→ v→ v→ Graph v
neighbours  :: Ord v⇒ Graph v→ v→ Set v
hasEdge     :: Ord v⇒ Graph v→ v→ v→ Bool
empty       :: Graph v
discrete    :: Ord v⇒ Graph v→ Graph v
```

```
fromList  :: Ord v⇒ [(v, v)]→ Graph v
toList    :: Ord v⇒ Graph v→ [(v, v)]
```

Hierbei ist Set der Datentyp aus dem Modul Data.Set, mit dem Mengen modelliert werden können.

10.2 *Labyrintherzeugung* — *6 Punkte*

Jetzt wollen wir Labyrinthe erzeugen. Gegeben sein ein ungerichteter Graph U, der alle denkbaren Übergänge modelliert, und ein Startknoten s von U. Gesucht ist ein Untergraph von U mit denselben Knoten, der einen aufspannenden Baum von U darstellt, d.h. es gibt zu jedem Knoten v einen Pfad von s nach v.

Es gibt eine ganze Handvoll Algorithmen, um Labyrinthe (aufspannende Bäume) zu erzeugen.[1] Startpunkt ist immer ein Graph U und ein Startknoten s in U. Einer der einfachsten Algorithmen, rekursives Backtracking, funktioniert wie folgt:

(1) Zu jedem Zeitpunkt haben wir einen bis hierher konstruierten Graphen G, eine Menge von schon besuchten Knoten, eine Liste von noch nicht vollständig abgearbeiteten Knoten P und einen aktuellen Knoten v.

(2) Wir betrachten jetzt die Menge I der noch nicht besuchten Nachbarn von v.

(3) Ist diese Menge I nicht leer, wählen wir zufällig einen der noch nicht besuchten Nachbarn, w, von v aus, fügen zu dem Graphen G eine Kanten von v nach w hinzu, fügen w zu der Menge der schon besuchten Knoten hinzu, fügen v zu der Liste P hinzu, und rufen uns rekursiv mit dem neuen Knoten w auf.

(4) Ist die Menge I leer, versuchen wir Backtracking (hier erzeugen wir potenziell eine Abzweigung): ist die Liste P noch nicht leer, wird der Kopf von P der nächste aktuelle Knoten und wir rufen uns rekursiv auf, ansonsten (P ist leer) sind wir am Ende und geben den momentanen Graphen zurück.

Implementieren Sie die Erzeugung eines Labyrinthes ein Modul Maze mit der Funktion

```
create :: Ord v ⇒ Graph v→ v→ IO (Graph v)
```

Eine nützliche Hilfsfunktion, die Sie zuerst implementieren, ist

```
pick :: Set a→ IO a
```

welche aus einer nicht-leeren Menge zufällig ein Element auswählt.

10.3 *Labyrinthe darstellen* — *6 Punkte*

Jetzt wollen wir unser Labyrinth natürlich auch darstellen. Wir fangen mit „klassischen“ Labyrinthen an, die eine gitterartige Struktur haben — jeder Knoten hat höchstens vier Nachbarn, und die Knoten lassen sich als Gitter anordnen. Hier hat der zu Grunde liegende Graph als Knoten Paare (n, m) von natürlichen Zahlen, und Kanten von (n, m) nach $(n + 1, m)$ und $(n, m + 1)$ (und in der anderen Richtung, weil der Graph nicht gerichtet ist).

Ein Gitter-Labyrinth wird durch einen Datentyp Maze implementiert, und besteht aus einem Graphen Graph (Int, Int), einem Startknoten, sowie der Größe. Wir definieren folgende Funktionen:

- render :: Maze→ Int→ Svg gibt eine graphische Darstellung des Labyrinthes als SVG-Grafik, repräsentiert durch den Typen Svg aus Text.Blaze.Svg11, zurück. Der zweite Parameter ist die Größe (in Pixeln) einer einzelnen Zelle. Abbildung 1 zeigt eine beispielartige Repräsentation; der Startpunkt ist durch einen dicken grünen Punkt dargestellt.
- new :: Int→ Int→ IO Maze erzeugt ein neues Labyrinth der angegebenen Größe (Breite und Höhe).

Für new ist eine Funktion initialGraph nützlich, die ein volles Gitter der angebenen Größe erzeugt. Dieser Graph ist der zu Grunde liegende Graph U für die Erzeugung des Labyrinths (siehe Maze.create).

[1] Wikipedia ist eine guter Startpunkt, und diese Webseite hat eine Handvoll Algorithmen mit schönen Animationen dazu: http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap.html

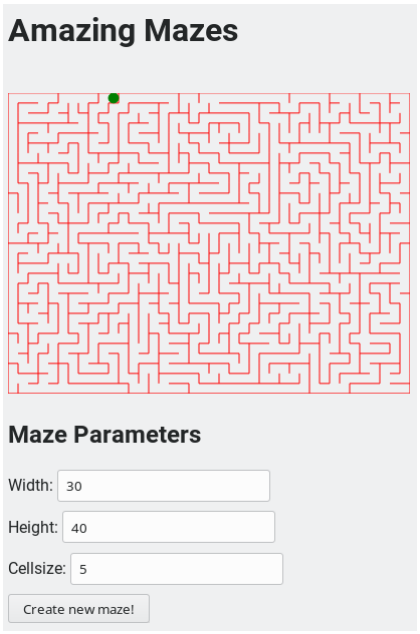

Abbildung 1: Ein Beispiel für ein Labyrinth (im Webinterface).

10.4 *Der Benutzer kommt ins Spiel* *3 Punkte*

Für die einfache Benutzung stellen wir eine Webschnittstelle bereit. Ein Gerüst für diese ist im Modul `Server` implementiert, Sie müssen dort noch ihre Funktionen aus `GridMaze` anbinden.

Die Webschnittstelle benutzt die aus der Vorlesung bekannte Einbettung von HTML/SVG in Haskell, sowie den Haskell-Webserver `warp`. Um den Server zu übersetzen, benötigen Sie folgende Module, die Sie mit `cabal install` installieren:

```
cabal install blaze-html blaze-svg warp
```

10.5 *Mehr Labyrinthe* *Bis zu 10 Bonuspunkte*

Als Zusatzaufgabe können Sie eine (oder mehrere) der folgenden Erweiterungen implementieren:

(a) *Andere Generierungsalgorithmen:* Rekursives Backtracking ist nur eine von vielen Möglichkeiten, Labyrinthe zu erzeugen. Implementieren Sie einen anderen (auf der oben referenzierten Webseite aufgeführten) Algorithmus, und vergleichen Sie Performance und Ästhetik der erzeugten Labyrinthe.

(b) *Anderes Layout:* Labyrinthe müssen nicht gitterartig sein! Erzeugen Sie hexagonale, dreieckige, runde, oder irreguläre Labyrinthe. Generell gehen alle Labyrinte von einem voll verbundenen Labyrinth (dem `initialGraph`) aus, und erzeugen daraus ein Labyrinth. Das Modul `Maze` kann dabei unverändert bleiben. Wichtig ist: was sind die Knoten, und wie wird das Layout erzeugt.

Erweitern Sie die Webschnittstelle um eine Selektionsbox (`select`), welche Benutzer auswählen lassen, welche Art von Labyrinth sie sehen möchten.

(c) *Mehr Funktionalität:* Erweitern Sie die Webschnittstelle so, dass man bei einem angezeugten Labyrinth in einen Knoten des Labyrinths klicken kann, und die Schnittstelle dann den Weg vom Startpunkt dorthin anzeigt.