Praktikum „SEP: Java-Programmierung“
SS 2019

Aufgabe 2: Game of Life

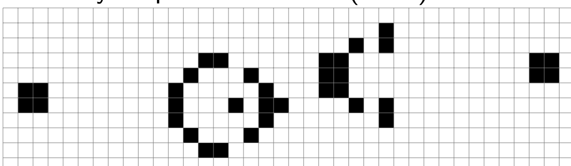
Thomas Bunk und Karlheinz Friedberger

Hinweise

- Nur 2 Wochen Zeit zur Bearbeitung!
- Finale Abgabe am Dienstag, 28.05.2019, um 10:00 Uhr

Game of Life

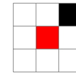
- John H. Conway's Spiel des Lebens (1970)



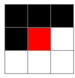
  - Mathematisches Spiel mit Zellen, die nach einigen Regeln leben, sterben oder sich vermehren können
  - Artikel in Scientific Amercian 223, Oktober 1970: https://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

Game of Life: Generationenwechsel (genetische Regeln)

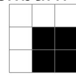
- Jede lebende Zelle mit...
  - weniger als zwei lebenden Nachbarn stirbt an Vereinsamung



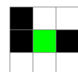
  - mehr als drei lebenden Nachbarn stirbt wegen Überbevölkerung



  - zwei oder drei lebenden Nachbarn überlebt

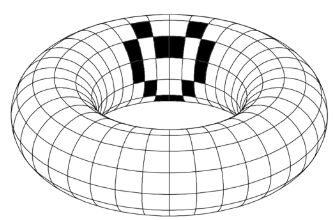


- Jede tote Zelle mit...
  - genau drei lebenden Nachbarn wird zu einer lebenden Zelle



Game of Life: Annahmen

- Vereinfachte Annahme
  - Begrenzter Speicher
  - Begrenztes Spielfeld von Anfang an
  - Zellen außerhalb des Spielfeldes werden als tot betrachtet
  - Realistischere alternative Variante: Spielfeld als Torus
    - aber nicht hier



Source: wikimedia commons

Game of Life: Start und Ende

- Start mit einer Initialpopulation
- Endsituation
  - Schwund: die ganze Population stirbt wegen Über- oder Unterbevölkerung
  - Stabilität: Population verändert sich nicht mehr beim Generationenwechsel
  - Oszillation: Population wiederholt sich nach zwei oder mehr Generationenwechsel in einer Endlosschleife
  - Unendliches Wachstum (nur bei unendlichem Spielfeld)

Game of Life: Initialisierung

- Gängige Initialpopulationen
  - Viele weitere Muster im *Life Lexicon*
  - Schwarze Zellen leben
  - Weiße Zellen sind tot

Block	
Boat	
Blinker	
Toad	

Glider	
Spaceship	
Pulsar	

Aufgabenstellung zu Aufgabe 2 (1/6)

- Kommandozeilenschnittstelle
  - Shell
  - Prompt `gol>`
- Folgende Kommandos sollen unterstützt werden
  - Game of Life

`NEW x y`

Starte neues Spiel mit neuer Spielfeldgröße

- x Anzahl der Spalten
- y Anzahl der Zeilen
- Alle Zellen tot

Aufgabenstellung zu Aufgabe 2 (2/6)

- Folgende Kommandos sollen unterstützt werden
  - Game of Life

`ALIVE i j`	Setze Zelle in *i*-ter Spalte und *j*-ter Zeile auf lebendig
`DEAD i j`	Setze Zelle in *i*-ter Spalte und *j*-ter Zeile auf tot
`GENERATE`	Berechne die nächste Generation nach den genetischen Regeln (Folie 4) ▶ In-place auf der Datenstruktur oder Out-of-place ▶ Zeige die (fortlaufende) Nummer der neuen Generation

Aufgabenstellung zu Aufgabe 2 (3/6)

- Folgende Kommandos sollen unterstützt werden
  - Game of Life

`PRINT`	Gib Spielfeld aus ▸ Lebende Zelle mit 'X' ▸ Tote Zellen mit '.' ▸ Als rechteckige Matrix
`CLEAR`	Töte alle lebenden Zellen ▸ Setze Generationenzähler zurück auf 0

Aufgabenstellung zu Aufgabe 2 (4/6)

- Folgende Kommandos sollen unterstützt werden
  - Game of Life

`RESIZE x y`

Ändere die Größe des Spielfeldes auf x Spalten und y Zeilen

- Lebende Zellen, deren Koordinaten sich auf auf dem neuen Spielfeld befinden bleiben am Leben
- Generationenzähler wird nicht zurückgesetzt

Aufgabenstellung zu Aufgabe 2 (5/6)

- Folgende Kommandos sollen unterstützt werden
  - Game of Life

`SHAPE s`	Lade vordefinierte Initialpopulation mit Namen s - Zumindest alle gängigen Initialpopulationen von vorher verfügbar (Folie 7) - Vorher wird (automatisch) ein CLEAR ausgeführt - Zentriert auf aktuellem Spielfeld - $gameX = (\frac{\#gameColumns - \#shapeColumns}{2}) + shapeX$ - $gameY = (\frac{\#gameRows - \#shapeRows}{2}) + shapeY$ - Wenn Population nicht auf das Spielfeld passt, Fehlermeldung ausgeben

Aufgabenstellung zu Aufgabe 2 (6/6)

- Folgende Kommandos sollen unterstützt werden
  - Allgemeine Kommandos

`HELP`	Gibt einen sinnvollen Hilfetext aus
`QUIT`	Beendet das Programm

  - Erster Buchstabe des jeweiligen Kommandos genügt
  - Groß- und Kleinschreibung der Befehle soll egal sein

- Fehlermeldungen beginnen mit “`Error! `”
- Hauptdatei heißt `Shell.java`
- Datei `Tests.txt` auf Praktomat mit Quellcode hochladen

Implementierung: Update bei Generationswechsel

- Entweder IN-PLACE oder OUT-OF-PLACE.
- OUT-OF-PLACE ist einfach und schnell umzusetzen
- IN-PLACE ist effizienter

Implementierung OUT-OF-PLACE

- 2D-Matrix `Cell[][]` wird umkopiert
  in eine zweite 2D-Matrix `Cell[][]`
  1. Alle Zellen des Grids durchlaufen
  2. Anzahl Nachbarn für jede Zelle der neuen Generation berechnen
- Referenz auf 2D-Matrix updaten

Implementierung IN-PLACE (1/3)

- Alle Geburten und Tode passieren simultan
  1. Anzahl Nachbarn für jedes in Frage kommende Mitglied der neuen Generation berechnen und speichern
     - Nachbarzähler **aller** lebenden Zellen zunächst auf 0 setzen
     - Dann alle momentan lebenden Zellen durchlaufen und jeweils 1 auf den Nachbarzähler ihrer bis zu 8 Nachbarn addieren
     - Jede Zelle, deren Zähler aktualisiert wurde, **kann** zur nächsten Generation gehören
  2. Alle Zellen der alten Population durchlaufen und die toten Zellen entfernen
  3. Alle Zellen der potentiellen neuen Generation (aus 1.) durchlaufen und die Neugeborenen auf lebendig setzen

Implementierung IN-PLACE (2/3)

- Neugeborene spielen keine Rolle beim Sterben oder Weiterleben
  - Dürfen also bei den Berechnungen nicht mitgezählt werden
  - Müssen daher separat gespeichert werden, nicht in aktueller Population
- An dieser Stelle können leicht Fehler gemacht werden!
- Alle Zellen des Spielfelds zu durchlaufen ist nicht notwendig für Generationenwechsel

Implementierung IN-PLACE (3/3)

- Hilfreich zum Speichern der Population Container aus `java.util`
  - `Set<Cell> population = new LinkedHashSet<Cell>();`
    - Verhindert von mehrfachen Hinzufügen einer Zelle
    - `population.contains(cell)` in $\mathcal{O}(1)$
    - Iteration über alle Elemente aus `population` in $\mathcal{O}(n)$
- Größenänderung des Gitters soll Population nicht löschen
  - Zellen bleiben da wo sie sind
  - Beim Verkleinern werden nicht mehr existente Zellen gelöscht
- Für genetische Regeln Konstanten verwenden

Implementierung

- Schnittstelle des *Models* zur `Shell`

```
public interface Grid {
    boolean isAlive(int col, int row); // get status of cell
    void setAlive(int col, int row, boolean alive); // set status of cell
    void resize(int cols, int rows); // resize grid
    int getColumns(); // x-dimension
    int getRows(); // y-dimension
    Collection<Cell> getPopulation(); // all living cells
    void clear(); // kill all cells
    void next(); // compute next generation
    int getGenerations(); // get number of generations
    String toString(); // get string representation
}
```

- Verwendung

```
public class Game implements Grid {
    // implement all methods listed in Grid
}
```

- Eigene Exceptions hier nicht notwendig
- Strikte Trennung der `Shell` von der Spiellogik
  - Zugriff nur über das Interface Grid
  - `Grid game = new Game(...);`

Hinweise

- Alle Zellen (lebendig und tot) in 2D-Matrix `Cell[][]`
  - Cell speichert Koordinate (x, y) und ggf. Nachbarzähler
- Aktuell lebendige Zellen ggf. nochmal separat als Menge `LinkedHashSet<Cell>`
- Wichtig: Bei `HashSet<K>` bzw. `HashMap<K, V>` muss `K` eine sinnvolle Implementierung von `hashCode()` bereitstellen
  - Außerdem natürlich auch `equals(Object o)`

Erinnerung

- Abgabetermin bereits in 2 Wochen!
- Achtet auf eine saubere Implementierung. Die Folgeaufgabe wird auf dieser aufbauen!
- Bei Fragen oder Anmerkungen, Mail an:
  - bunk[at]sosy.ifi.lmu.de
  - karlheinz.friedberger[at]ifi.lmu.de

  (Alternativ könnt ihr jederzeit gerne bei uns persönlich im Büro vorbeischauen. Die genaue Adresse findet ihr unter: https://www.sosy-lab.org/people.php)