Conway's Game of Life i Rain

Natalia Luberda

March 2023

1 Conway's Game of Life - wprowadzenie:

Conway's Game of Life jest jednym z najpopularniejszych przykładów zastosowania automatów komórkowych. Jest to dwuwymiarowy, jednorodny automat komórkowy oparty na siatce kwadratowej z sasiedztwem Moore'a. Każda komórka może być w jednym z dwóch stanów: 1 - żywa, 0 - martwa.

Martwa komórka zmienia swoje stan na żywy w nastepnej iteracji, jeśli ma dokładnie trzech żywych sasiadów, w przeciwnym razie pozostaje martwa.

Żywa komórka pozostaje żywa, jeśli ma 2 lub 3 żywych sasiadów, w przeciwnym razie umiera. Zasady te można zapisać jako 23/3.

1.1 Klasa Point

Ogólnie ta klasa odpowiada za reprezentowanie pojedynczego punktu w automacie komórkowym, śledzenie jego stanu i sasiadów oraz implementacje zasad automatu komórkowego w celu określenia nastepnego stanu punktu.

1.1.1 Atrybuty klasy:

Klasa ma nastepujace atrybuty:

- neighbors: ArrayList punktów reprezentujacych sasiednie punkty tego punktu.
- currentState: liczba całkowita reprezentujaca aktualny stan punktu.
- nextState: liczba całkowita reprezentujaca stan, w jakim ten punkt bedzie po nastepnej iteracji automatu komórkowego.
- numStates: liczba całkowita reprezentujaca liczbe możliwych stanów, jakie może mieć ten punkt.

1.1.2 Metody klasy:

Klasa ma nastepujace metody:

- clicked(): metoda zmieniajaca aktualny stan punktu na kolejny stan, którym jest nastepny stan w cyklu możliwych stanów.
- getState(): metoda zwracajaca aktualny stan punktu.
- setState(): metoda ustawiajaca aktualny stan punktu na określona wartość.
- calculateNewState(): metoda obliczajaca nastepny stan punktu na podstawie aktualnego stanu punktu i liczby aktywnych sasiadów punktu. Ta metoda implementuje zasady automatu komórkowego.
- changeState(): metoda ustawiajaca aktualny stan punktu na jego nastepny stan.
- addNeighbor(): metoda dodajaca punkt sasiedni do listy sasiadów tego punktu.
- activeNeighbors(): metoda zwracajaca liczbe aktywnych sasiadów punktu.

1.2 Wykonanie zadania

W implementacji programu należało uzupełnić brakujace fragmenty kodu w klasach Board i Point.

W klasie Board, w metodzie initialize (), należało zainicjować sasiadów dla każdej komórki w tablicy.

W klasie Point trzeba było napisać metode zwracajaca liczbe żywych sasiadów.

W metodzie calculateNewState (), należło obliczyć nowy stan danej komórki - zgodnie z jej aktualnym stanem i liczba żywych sasiadów oraz zapisać nowy stan komórki w zmiennej: nextState.

Nastenie należało zmofyfikować swój program, aby używał nastepujacych reguł: 2345/45678 - city, 45678/3 - coral.

1.2.1 Zainicjowaanie sasiadów dla każdej komórki w tablicy:

Ten kod inicjalizuje sasiedztwo dla każdej komórki w dwuwymiarowej tablicy "points".

```
for (int x = 0; x < points.length; ++x) {
    for (int y = 0; y < points[x].length; ++y) {
//TODO: initialize the neighborhood of points[x][y] cell
        for(int i = -1; i < 2; i++){ // x
            for(int j = -1; j < 2; j++){ //
if(x + i >= 0 && y + j >= 0
&& x + i < points.length && y + j < points[x].length
&& !(i == 0 && j == 0)){
                    points[x][y].addNeighbor(points[x + i][y + j]);

}
```

Pierwsza petla "for" iteruje po wierszach tablicy, a druga po kolumnach. Nastepnie dwie zagnieżdżone petle "for" przechodza przez sasiednie komórki, w których "i" reprezentuje zmiane współrzednej x, a "j" zmiane współrzednej y. Warunki if sprawdzaja, czy aktualnie przetwarzana komórka jest wewnatrz tablicy oraz czy nie jest to ta sama komórka co aktualnie przetwarzana. Jeśli warunki sa spełnione, dodaje sasiada do listy sasiadów aktualnie przetwarzanej komórki, wywołujac metode "addNeighbor".

W ten sposób inicjalizowane jest sasiedztwo każdej komórki w tablicy.

1.2.2 Wyznaczanie żywych sasiadów każdej komórki:

```
public int activeNeighbors() {
    int activeNeighbor = 0;
    for (Point neighbor : neighbors) {
        if (neighbor.currentState != 0) {
            activeNeighbor++;
        }
    }
    return activeNeighbor;
}
```

Metoda activeNeighbors() iteruje przez liste sasiadów danego punktu i zlicza ile z nich ma stan żywy (różny od 0). Nastepnie zwraca liczbe zliczonych sasiadów.

1.2.3 Nowy stan danej komórki - zasada 23/3:

Ten kod implementuje reguły gry w życie dla podstawowej wersji Cellular Automata z regułami 23/3, które określaja, jak komórki beda zmieniać swoje stany w kolejnych iteracjach gry.

```
int active = this.activeNeighbors();
// FIRST TASK: 23/3
if(this.currentState == 0){
    if(active == 3){
        this.nextState = 1;
    }
}
else if(active == 2 || active == 3){
    this.nextState = 1;
}
else{
    this.nextState = 0;
}
```

Jeśli bieżacy stan komórki to 0, a liczba aktywnych sasiadów to dokładnie 3, to w nastepnej iteracji komórka zmieni swój stan na 1, ponieważ została "zrodzona" przez trzy aktywne komórki-sasiadów.

Jeśli bieżacy stan komórki to 1, a liczba aktywnych sasiadów wynosi 2 lub 3, to w nastepnej iteracji komórka pozostanie aktywna, ponieważ ma wystarczajaca ilość aktywnych komórek-sasiadów do przetrwania.

Jeśli bieżacy stan komórki to 1, a liczba aktywnych sasiadów jest mniejsza niż 2 lub wieksza niż 3, to w nastepnej iteracji komórka umrze i zmieni swój stan na 0.

1.2.4 Nowy stan danej komórki - zasada 2345/45678:

Ten fragment kodu odnosi sie do drugiego zadania, w którym należy zaimplementować inne reguły przejścia dla Gry w życie, mianowicie 2345/45678 - miasta.

```
int active = this.activeNeighbors();
// TASK 2: 2345/45678 - cities
if (this.currentState == 0) {
    if (active == 6 || active == 4 || active == 5 ||
    active == 7 || active == 8) {
        this.nextState = 1;
    }
} else if (active == 2 || active == 3 || active == 4 || active == 5) {
    this.nextState = 1;
} else {
    this.nextState = 0;
}
```

W tym przypadku, dla każdej komórki planszy należy sprawdzić ilość żywych sasiadów (z wykorzystaniem metody activeNeighbors()), a nastepnie na podstawie jej aktualnego stanu oraz ilości żywych sasiadów określić jej stan w kolejnej iteracji i zapisać go w zmiennej nextState.

W przypadku tej reguły, jeśli aktualna komórka jest martwa i ma dokładnie 4, 5, 6, 7 lub 8 żywych sasiadów, to w kolejnej iteracji staje sie żywa. Jeśli natomiast jest żywa, to pozostaje taka, jeśli ma 2, 3, 4 lub 5 żywych sasiadów, a w przeciwnym razie staje sie martwa.

1.2.5 Nowy stan danej komórki - zasada 45678/3:

Ten kod implementuje reguły gry w życie dla automatu komórkowego z regułami 45678/3, zwanej także "koralowym" automatem.

```
int active = this.activeNeighbors();
//TASK 3: 45678/3 - coral
                if(this.currentState == 0){
                        if(active == 3){
                                this.nextState = 1;
                        }
                }else if(active == 5 || active == 4||
  active == 6 || active == 7 || active ==8  ){
```

```
            this.nextState = 1;
        }else{
            this.nextState = 0;
        }
```

W tym przypadku, komórka jest uznawana za żywa, jeśli ma trzech żywych sasiadów. Jeśli komórka jest martwa, ale ma pieciu, czterech, sześciu, siedmiu lub ośmiu żywych sasiadów, staje sie żywa w nastepnej iteracji. W przeciwnym razie pozostaje martwa.

1.3 Wnioski:

W naszym programie zaimplementowaliśmy symulacje Conway's Game of Life z trzema różnymi zestawami reguł. Przy użyciu pierwszego zestawu reguł (23/3) w każdej iteracji symulacji z tym zestawem reguł tworzy wzorce, które przemieszczaja sie i zmieniaja kształt w czasie. Widzimy, jak żywe komórki łacza sie w wieksze formacje i tworza stałe wzorce.

Drugi zestaw reguł (2345/45678) tworzy efektywnie miasta zbudowane z klastrów ożywionych komórek, w których każdy klaster ma swoja unikalna strukture. W tej symulacji wzorce sa mniej stabilne i przemieszczaja sie bardziej chaotycznie niż w przypadku pierwszego zestawu reguł.

Trzeci zestaw reguł (45678/3) tworzy wzory przypominajace rafy koralowe, składajace sie z klastrów ożywionych komórek o nieregularnym kształcie. Ten zestaw reguł wytwarza bardziej statyczne i złożone wzory niż dwa poprzednie zestawy.

Wnioskiem z tych symulacji jest to, że zestawy reguł definiujace symulacje maja kluczowe znaczenie dla wyniku symulacji. Każdy zestaw reguł wytwarza zupełnie inne wzorce i efekty wizualne, które sa bardzo wrażliwe na reguły, które nimi rzadza. Ten kod pokazuje, jak zmiana zestawu reguł może mieć znaczacy wpływ na końcowy wynik i wyglad symulacji.

2 Rain - wprowadzenie:

Zadanie polega na zaimplementowaniu symulacji deszczu w automacie komórkowym. Należy ustawić sasiedztwo komórek tak, aby jedynym sasiadem danej komórki był element znajdujacy sie bezpośrednio poniżej niej. Korzystamy z tych samych klas co w zadaniu pierwszym, lekko je modywikujac.

2.1 Wykonanie zadania:

W klasie Point należy napisać metode drop(), która z pewnym małym prawdopodobieństwem (np. 5 procent) zmieni stan komórki na 6. W klasie Board, w metodzie iteration() należy użyć metody drop() dla komórek w górnym rzedzie. Jeśli stan komórki jest wyższy niż 0, należy ustawić nextState na currentState - 1. Jeśli stan komórki wynosi 0, a jej sasiad jest wiekszy niż 0, należy ustawić nextState na 6. W klasie Board, w metodzie drawNetting() należy zmienić

kolory komórek, tak aby kolor niebieski stawał sie wyblakły wraz ze zmniejszaniem sie liczby stanu. Można użyć metody: g.setColor(new Color(0.0f, 0.0f, 1.0f, 0.65f)); Parametry konstruktora klasy Color to: czerwony, zielony, niebieski, alfa.

2.1.1 Dodawanie sasiadów komórki:

Ten fragment kodu służy do inicjalizacji sasiedztwa komórek w siatce punktów. Najpierw tworzony jest zmienny "i" równy -1, a nastepnie iteruje sie przez wszystkie komórki siatki przy pomocy petli for. W każdym przebiegu petli dodawany jest sasiad dla komórki w punkcie (x,y), którego indeks y przesuniety jest o wartość zmiennej "i". Należy jednak zauważyć, że sprawdzana jest poprawność indeksów w celu unikniecia wyjścia poza zakres tablicy.

```
for (int x = 0; x < points.length; ++x) {
        for (int y = 0; y < points[x].length; ++y) {
//TODO: initialize the neighborhood of points[x][y] cell
                int i = -1; // x// y
        if(y + i >= 0 && y + i < points.length){
        points[x][y].addNeighbor(points[x][y+i]);
        }
  }
}
```

Ostatecznie, każda komórka posiada jeden sasiad - komórke znajdujaca sie bezpośrednio powyżej niej. Ten rodzaj sasiedztwa można wykorzystać np. do symulacji opadów atmosferycznych, gdzie opady w jednej komórce wpływaja jedynie na komórke znajdujaca sie bezpośrednio poniżej niej.

2.1.2 Metoda, która z pewnym małym prawdopodobieństwem (np. 5 procent) zmieni stan komórki na 6:

Metoda drop() w klasie Point służy do zmiany stanu komórki na 6 z pewnym małym prawdopodobieństwem (np. 5 procent)

```
public void drop(){
        if((int)(Math.random()*100) <= 6 ){
                this.setState(6);
        }
}
```

Wykorzystywana jest do symulacji opadów deszczu. Metoda losuje liczbe z przedziału [0, 100) i jeśli wylosowana liczba jest mniejsza lub równa 6, to zmienia stan komórki na 6.

2.1.3 Zmiany stanów komórki:

Metoda "calculateNewState()" oblicza nowy stan komórki w zależności od aktualnego stanu oraz liczby sasiednich komórek.

```
public void calculateNewState() {
        int active = this.activeNeighbors();
        if(this.getState()>0){
                this.nextState = this.getState() -1;
        }else if(this.getState()== 0 && this.activeNeighbors()>0){
                this.nextState = 6;
        }


}
```

Najpierw jest wywoływana metoda "activeNeighbors()" zwracajaca liczbe aktywnych sasiadów komórki. Nastepnie, jeśli stan komórki jest wiekszy od 0, nowy stan to stan aktualny pomniejszony o 1. Jeśli stan komórki wynosi 0 i istnieje przynajmniej jeden aktywny sasiad, nowy stan to 6.

2.1.4 Zmiana koloru komórki, aby niebieski stawał sie wyblakły wraz ze zmniejszaniem sie liczby stanu :

Metoda drawNetting() rysuje siatke tła oraz ustawia kolor komórek na podstawie ich stanu (state).

```
// draws the background netting
private void drawNetting(Graphics g, int gridSpace) {
        Insets insets = getInsets();
        int firstX = insets.left;
        int firstY = insets.top;
        int lastX = this.getWidth() - insets.right;
        int lastY = this.getHeight() - insets.bottom;

        int x = firstX;
        while (x < lastX) {
                g.drawLine(x, firstY, x, lastY);
                x += gridSpace;
        }

        int y = firstY;
        while (y < lastY) {
                g.drawLine(firstX, y, lastX, y);
                y += gridSpace;
        }

        for (x = 0; x < points.length; ++x) {
                for (y = 0; y < points[x].length; ++y) {
                        if (points[x][y].getState() != 0) {
                                switch (points[x][y].getState()) {
                                case 1:
```

```
g.setColor(new Color(0.0f, 0.0f, 1.0f, 0.65f));
                                                    break;
                                                case 2:
g.setColor(new Color(0.0f, 0.0f, 0.8f, 0.65f));
                                                    break;
                                                case 3:
g.setColor(new Color(0.0f, 0.0f, 0.6f, 0.65f));
                                                    break;
                                                case 4:
g.setColor(new Color(0.0f, 0.0f, 0.4f, 0.65f));
                                                    break;
                                                case 5:
g.setColor(new Color(0.0f, 0.0f, 0.2f, 0.65f));
                                                    break;
                                                case 6:
g.setColor(new Color(0.0f, 0.0f, 0.1f, 0.65f));
                                                    break;
                                                }
g.fillRect((x * size) + 1, (y * size) + 1, (size - 1), (size - 1));
                                            }
                                        }
                                    }

                }
```

Metoda rysuje siatke w tle planszy, a nastepnie iteruje przez wszystkie komórki na planszy (przechowywane w tablicy dwuwymiarowej points). Dla każdej komórki, która ma stan inny niż 0 , ustawiany jest odpowiedni kolor na podstawie jej stanu. Kolory te sa zdefiniowane jako kolor niebieski z różnym stopniem przezroczystości (alfa), co powoduje, że kolory te staja sie bardziej "rozmyte" wraz ze zmniejszaniem sie wartości stanu. Nastepnie komórka jest wypełniana na planszy odpowiednim kolorem w zależności od stanu.

2.2 Wnioski i efekt wizualny:

Na podstawie kodu można zaimplementować symulacje opadów deszczu. Każdej komórce planszy przypisuje sie stan odpowiadajacy poziomowi wody, a sasiedzi to tylko komórki znajdujace sie bezpośrednio pod dana komórka.

W metodzie iteration() klasy Board określa sie, jakie beda kolejne stany komórek na podstawie ich stanów i liczby aktywnych sasiadów. Jeśli stan komórki jest wiekszy od 0, zmniejsza sie o 1, a jeśli wynosi 0, a jej sasiad ma stan wiekszy od 0, ustawia sie stan na 6.

W metodzie drawNetting() klasy Board rysuje sie tło planszy i każda komórka wypełniana jest kolorem, który odpowiada jej stanowi. Im wyższy stan, tym ciemniejszy kolor.

Efektem wizualnym jest symulacja opadów deszczu na planszy o kształcie

siatki. Każda kropla deszczu jest reprezentowana przez komórke o stanie równym 6. Komórki sasiadujace z ta kropla, jeśli ich stan jest wyższy od 0, zmniejszaja swój stan o 1, co odzwierciedla wypływanie wody z danej komórki. Wizualnie na planszy pojawiaja sie wypełnione kolorem jasnoniebieskim komórki, które z czasem zmieniaja swój kolor na ciemniejszy, aż do stanu 0, który odpowiada brakowi wody w danej komórce.