verslag programmeren

Isabel Baas, Daan Braakman & Mylan Koobs

January 2023

Inhoudsopgave

1 inleiding

2 Algoritme

2.1 Data-structuur

In de code zitten meerdere klassen. De meest algemene klasse, ..., heeft een rooster met een zelf te kiezen aantal cellen in iedere dimensie. De dimensie is ook zelf te kiezen. Voor het rooster is er gebruik gemaakt van numpy arrays. Deze maken het makkelijker om te werken met meerdere dimensies.
Daarnaast heeft de klasse een regel-functie, die zelf gedefineerd kan worden, bepaald de toestanden van de volgende generaties cellen.
Ook bevat de klasse een functie voor het updaten, deze wordt later verder toegelicht.

Er zijn twee klassen afgeleid van de algemene klasse, die voor één- en voor tweedimensionale CA's. Deze bevatten alles van de algemene klasse en dat wat er nodig is om ze grafisch weer te geven. Er is een neighborhood klasse met functies waarmee de staten van omliggende cellen van een cel mee kunnen worden opgehaald. Deze kunnen gebruikt worden bij het defineren van de regel-functies.

2.2 Update

De update-functie update het gehele rooster door voor elke cel de rules functie op te roepen om te bepalen wat zijn nieuwe staat moet zijn. De complextiteit hangt af van de complextiteit van de regel-functie. Deze noemen we m. Bij n cellen wordt de rules-functie n keer aangeroepen, dus dan is de complextiteit mn.

2.3 Grafische weergave

Voor het grootste deel zijn de draw-functies voor 1D en 2D hetzelfde. De complextiteit is bijvoorbeeld helemaal gelijk. Deze is compleet afhankelijk van het aantal cellen. Zij dat aantal gelijk aan n, dan is de complexiteit dus ook n.

Voor de grafische weergave van de ééndimensionale CA was het nodig om als het waren mee te scrollen met de nieuwe generaties. Deze raakten namelijk na een aantal generaties buiten beeld. Om dit probleem op te lossen, wordt elke nieuwe generatie toegevoegd aan een lijst. Zodra de lijst meer generaties bevat dan grafisch weer te geven zijn op het scherm, word de oudste generaties uit de lijst gehaald en is deze dus ook niet meer te zien.

2.4 Neighborhood

The neighboorhoud functies zijn bedoeld om bij het maken van regels-functies makkelijk toegang te krijgen tot de toestanden van buurtcellen. Er wordt ge-

bruikt gemaakt van een Moore-Neighborhood met een zelf te kiezen afstand, die we r noemen. Ook kan dit voor elk aantal dimensies, die we d noemen (de algemene functie maakt wel gebruik van recursie en is daarom wat minder snel, dus voor 1- en 2-dimensionale CA's kan beter gebruik worden gemaakt van de daarvoor bestemde neighborhood-functies).

De functies lezen de toestanden van in totaal r^d cellen, en aangezien het lezen in constante tijd kan, is de uiteindelijke complexiteit ook r^d.

Voor de periodieke neighborhood-functie wordt nog gebruik gemaakt van modulo-operaties (modulo scherm-hoogte of -breedte), dus bij grote schermgroottes kan dit voor extra complexiteit zorgen doordat de complexiteit van modulo-operaties dan groter wordt.

2.5 Overig

Als laatste

3 Handleiding

3.1 Importeren

Stop het CA.py bestand in dezelfde map als het project, en importeer het als volgt:

```
import CA
#als de naam van het bestand veranderd is, gebruik dan de nieuwe naam ipv CA
```

Zorg er ook voor dat numpy geïnstalleerd is, want dat is wat gebruikt wordt voor het grid van de CA. Voor het gebruik maken van de ingebouwde visualisatie zal ook pygame geïnstalleerd moeten zijn.

3.2 Aanmaken van CA's

Verschillende cellulaire automaten kunnen gemaakt worden met behulp van de basisklasse `CellularAutomata`. Een CA kan als volgt gemaakt worden:

```
myCA = CA.CellularAutomata(shape, rules)
```

waar shape de vorm van de CA is, wat weergegeven word als een lijst met het aantal cellen in elke dimensie, en rules de functie is die, gegeven een cell, de index van deze cell en de rest van het rooster, de geüpdate staat van deze cell geeft.

3.2.1 Voorbeeld 1

We maken hier een 3-dimensionale CA met in elke dimensie 10 cellen, die als regel heeft dat als een cell de waarde 1 heeft, hij 0 wordt en andersom.

We definiëren eerst de functie voor onze regel, en vullen die in wanneer we de CA maken.

```
def invert(cell, index, grid):
    if cell == 0:
        return 1
    else:
        return 0

myCA = CA.CellularAutomata([10,10,10], invert)
```

3.2.2 1- en 2-dimensionale CA's

Ook kunnen we eenvoudig 1- en 2-dimensionale CA's maken met de `Cellular1D` en `Cellular2D` klassen. Deze bevatten al ingebouwde functies voor visualisatie, wat handig kan zijn.

Een 1-dimensionale CA met 10 cellen kan als volgt worden aangemaakt:

```
#we gebruiken dezelfde functie invert als bij het eerste voorbeeld
myCA_1D = CA.Cellular1D(10, invert)
```

Merk op dat hier alleen een getal gegeven wordt voor de grootte, en geen lijst.

Een 2-dimensionale CA met 10 cellen in de breedte en 5 in de hoogte kan als volgt worden gemaakt:

```
myCA_2D = CA.Cellular2D(10, 5, invert)
```

Ook hier wordt de breedte en hoogte los gegeven, en niet in een lijst.

3.3 Het invullen van cellen

Wanneer een CA wordt aangemaakt, begint hij met 0 in elke cell. Er zijn verschillende manieren om waardes aan cellen toe te kennen.

De meest algemene, die voor elke CA werkt, is `setcells`. Deze functie neemt een lijst met index-tuples en een bepaalde waarde, en kent voor elke index-tuple de bijbehorende cell de gegeven waarde toe.

```
#op plaatsen (5,5,5), (0,0,0) & (1,2,3) wordt de waarde van de cell 1.
myCA.setcells([(5,5,5), (0,0,0), (1,2,3)], 1)
```

Ook bestaat er een functie die alle waarden terugzet naar 0, `setzeros`. Deze werkt ook voor elke CA.

```
myCA.setzeros()
```

Daarnaast bestaat er ook de functie `random`, die ook voor elke CA werkt, en die alle cellen random 0 of 1 maakt.

```
myCA.random()
```

Verder is er ook een functie speciaal voor `Cellular1D`, namelijk `start_middle`, die de cell in het midden naar 1 verandert.

```
myCA_1D.start_middle()
```

3.4 Updaten van CA

Het updaten van CA's kan eenvoudig gedaan worden met de `update` functie. Deze functie roept voor iedere cell de rules-functie op, en geeft de cell de output als waarde.

```
myCA.update()
```

Ook kan de CA automatisch een gegeven aantal keer geupdate worden door middel van de `run` functie. Deze neemt een getal voor het aantal keer dat de CA geupdate moet worden, en update hem dan zoveel keer.

```
myCA.run(500)
```

3.5 Grafische weergave

Voor de 1- en 2-dimensionale CA klassen bestaan er ingebouwde functies voor het visualizeren.

Allereerst bestaat voor beide de `draw` functie. Deze neemt als argumenten een screen (een pygame Surface), een cellsize voor de grootte van de cellen (in pixels), en een surflist, een lijst van pygame Surfaces.

Het `screen` is hetgene waar alles op getekend word, en `surflist` is een lijst met surfaces met verschillende kleuren, die gebruikt worden om alle cellen in te kleuren. Hierbij wordt de toestand van de cell gebruikt als index voor de `surflist`, dus een cell met toestand in wordt ingekleurd met de surface op plek 0 van de lijst, enzovoort. Zorg ervoor dat deze surfaces vierkant zijn met dezelfde cellgroote als de `draw` functie.

```
screen = pygame.display.set_mode((640,640))
wit = pygame.Surface(10,10)
wit.fill((255,255,255))
zwart = pygame.Surface(10,10)
zwart.fill((0,0,0))
surflist = [zwart, wit]

#Dit kan ook met 1D CA's
myCA_2D.draw(screen, 10, surflist)
```

Daarnaast bestaan er een functie die het updaten en weergeven automatisch doen, `runvisual`. Deze functie neemt als argumenten de breedte en hoogte van het scherm (in pixels), de `changetime` (tijd tussen updates in millisecondes, vooral nuttig bij CA's die anders te snel gaan), de cellsize in pixels, en een lijst met kleuren voor elke toestand. Als deze lijst kleiner is dan het aantal

toestanden, krijgen cellen met een toestand waarvoor geen kleur in de lijst zit de laatste kleur uit de lijst.

De visualisatie kan op pauze gezet worden door op de spatiebalk te drukken.

```
#dit kan wederom ook weer met 1-dimensionale CA's
myCA_2D.runvisual(640,640, 100, 10, [(0,0,0), (255,255,255)])
```

3.6 Het maken van regels

Voor het maken van regels kan gebruik gemaakt worden van ingebouwde functies die de staten van buurtcellen ophalen. Deze zitten in de `Neighborhoods`-klasse. Er zijn functies voor 1- en 2-dimensionale buurten, en functies voor buurten met willekeurige dimensie. Voor 1- en 2D wordt aangeraden om de daarvoor bestemde functies te gebruiken en niet de algemene, aangezien die meestal iets langzamer is doordat er gebruik gemaakt wordt van recursie. Er bestaan voor elk van die categorieën nog twee soorten functies, waar het verschil zit in de randvoorwaarden.

De standaard `Neighborhoods.get_neighbors` functie neemt naast parameters voor het grid, de index van de huidige cell en de reach ook nog een waarde genaamd `default` aan, die gebruikt wordt wanneer een cell buiten het grid ligt (en dus eigenlijk niet bestaat).

De `Neighborhoods.get_neighbors_periodiek` gaat verder vanaf de andere kant wanneer cellen niet bestaan.

Deze functies geven als output een lijst met de staten van de cellen. De volgorde begint bij de laagste indexen in alle dimensies, beweegt naar de hoogste index in de hoogste dimensie, en beweegt dan 1 omhoog in de één-na-hoogste dimensie en gaat weer naar de hoogste index in de hoogste dimensie, en dit gaat zo door totdat alle cellen zijn opgehaald.

Als voorbeeld maken we hier een functie voor rule 22, één van de elementaire CA's.

```
import CA

def rule22(cell, idx, grid):
    #hier halen we de staten van de buurcellen op
    states = CA.Neighborhoods.get_neighbors1D_periodiek(grid, idx, 1)

    #definieren links, rechts en het midden voor gemak
    left = states[0]
    center = states[1]
    right = states[2]

    #alle voorwaarden met de bijbehorende output
    # (gebruik makend van dat 0 False geeft bij gebruik als boolean
```

```
    # en getallen ongelijk 0 True)
    if left and center and right:
        return 0
    elif left and center and not right:
        return 0
    elif left and not center and right:
        return 0
    elif left and not center and not right:
        return 1
    elif not left and right and center:
        return 0
    elif not left and center and not right:
        return 1
    elif not left and not center and right:
        return 1
    else:
        return 0
```

Ook kunnen we hiermee vrij eenvoudig een regelfunctie maken voor Game of Life.

```
def Game_of_life_rules(cell, idx, grid):
        states = CA.Neighborhoods.get_neighbors2D_periodiek(grid, idx, 1)
        levende_buren = 0
        #telt de levende buren
        for i in states:
                if i == 1:
                    levende_buren = levende_buren + 1

        #laat er een geboren worden als er precies 3 levende buren zijn
        if cell == 0:
            if levende_buren == 3:
                return 1
            else:
                return 0


        #laat een levende cel sterven door over- of onderbevolking
        if cell == 1:
            #er is hier rekening gehouden met dat cell ook leeft
            if levende_buren > 4 or levende_buren < 3:
                return 0
            else:
                return 1
```

Ook kunnen we gebruik maken van meerdere toestanden, wat te zien is in het volgende voorbeeld waarbij zieke cellen zijn toegevoegd aan Game of Life, die

andere cellen ziek kunnen maken.

```
def Game_of_Life_Sickness(cell, idx, grid):
    states = Neighborhoods.get_neighbors2D_periodiek(grid, idx, 1)
    levende_buren = 0
    zieke_buren = 0
    #telt de levende en zieke buren
    for i in states:
        if i == 1:
            levende_buren = levende_buren + 1
        if i >= 2:
            zieke_buren = zieke_buren + 1

    #laat er een geboren worden als er precies 3 levende buren zijn
    if cell == 0:
        if levende_buren == 3:
            return 1
        else:
            return 0


    if cell == 1:
        #maakt een cel ziek (toestand 2) als er 3 of meer zieke buurcellen zijn
        if zieke_buren >= 3:
            return 2
        #laat een levende cel sterven door over- of onderbevolking
        elif levende_buren > 4 or levende_buren < 3:
            return 0
        else:
            return 1


    if cell >= 2 and cell < 5:
        #als een zieke cell meer dan 2 levende buren heeft, geneest de cell
        if levende_buren > 2:
            return 1
        #anders wordt de cell nog zieker
        else:
            return cell+1

    #als de cell erg ziek is, gaat hij dood
    if cell >= 5:
        return 0
```

3.7 Overzicht van functies

`CA.CellularAutomata(shape: list, rules)`: Maakt een cellulaire automata aan met de shape voor de vorm van het grid, en een functie rules met als input een cell, een index en het grid en als output de geupdate staat van de cell.

`CA.Cellular1D(size: int, rules)`: Maak een 1-dimensionale CA met grootte size en een functie rules met als input een cell, een index en het grid en als output de geupdate staat van de cell.

`CA.Cellular2D(width: int, height: int, rules)`: Maak een 2-dimensionale CA met width cellen in de breedte en heigth cellen in de hoogte en een functie rules met als input een cell, een index en het grid en als output de geupdate staat van de cell.

`CA.CellularAutomata.update()`: Update het hele grid door voor elke cel de rules functie op te roepen om te bepalen wat zijn nieuwe staat moet zijn.

`CA.CellularAutomata.run(updates: int)`: Update het grid een gegeven aantal keer.

`CA.CellularAutomata.setcells(coordinates: list, value: int)`: Verandert de waarden van cellen met de gegeven coördinaten naar de gegeven waarde.

`CA.CellularAutomata.setzeros()`: Verandert alle toestanden in het grid naar 0.

`CA.CellularAutomata.random(max: int)`: Verandert alle toestanden in het grid naar random getallen tussen 0 en max.

`CA.Cellular1D.start_middle()`: Maakt de toestand van de middelste cel gelijk aan 1. Vooral bedoelt voor de elementaire CA's.

`CA.Cellular1D.draw(screen: pygame.Surface, cellsize: int, surflist: list)`: Tekent het grid op een screen, scrollt automatisch mee als onderkant van scherm wordt bereikt. Gebruikt de surflist met surfaces om de cellen in te tekenen op basis van de toestand.

`CA.Cellular1D.runvisual(width: int, height: int, changetime: int, cellsize: int, colorlist: list)`: Start pygame visualisatie met bepaalde width en height in pixels, changetime geeft tijd in ms tussen updates en colorlist zorgt voor de kleuren per toestand (als lijst met RGB-waardes).

`CA.Cellular2D.draw(screen: pygame.Surface, cellsize: int, surflist: list)`: Tekent het grid op een screen. Gebruikt de surflist met surfaces om de cellen in te tekenen op basis van de toestand.

`CA.Cellular2D.runvisual(width: int, height: int, changetime: int, cellsize: int, colorlist: list)`: Start pygame visualisatie met bepaalde width en height in pixels, changetime geeft tijd in ms tussen updates en colorlist zorgt voor de kleuren per toestand (als lijst met RGB-waardes).

`CA.GameOfLife(width: int, height: int)`: Maakt een 2D CA aan met Game of Life regels.

`CA.GameOfLife.glider(offset_width: int, offset_height: int, direction: int)`: Zet een zogenaamde glider in het grid, met de offset ten opzichte van linksboven en de richting een getal tussen 0 en 3, met 0 voor naar linksboven, 1 voor naar rechtsboven, 2 voor linksonder en 3 voor rechtsonder.

`CA.Neighborhoods.get_neighbors_1D(grid: np.ndarray, idx: list, reach: int, default: int)`: geeft een list met de toestanden van buurtcellen in een 1D Moore-Neighborhood met reach voor de grootte van de buurt. Gebruikt default als cellen niet bestaan.

`CA.Neighborhoods.get_neighbors_1D_periodiek(grid: np.ndarray, idx: list, reach: int)`: geeft een list met de toestanden van buurtcellen in een 1D Moore-Neighborhood met reach voor de grootte van de buurt. Gaat verder vanaf de andere kant als cellen niet bestaan.

`CA.Neighborhoods.get_neighbors_2D(grid: np.ndarray, idx: list, reach: int, default: int)`: geeft een list met de toestanden van buurtcellen in een 2D Moore-Neighborhood met reach voor de grootte van de buurt. Gebruikt default als cellen niet bestaan.

`CA.Neighborhoods.get_neighbors_2D_periodiek(grid: np.ndarray, idx: list, reach: int)`: geeft een list met de toestanden van buurtcellen in een 2D Moore-Neighborhood met reach voor de grootte van de buurt. Gaat verder vanaf de andere kant als cellen niet bestaan.

`CA.Neighborhoods.get_neighbors(grid: np.ndarray, idx: list, reach: int, default: int)`: geeft een list met de toestanden van buurtcellen in een willeukeurig-dimensionale Moore-Neighborhood met reach voor de grootte van de buurt. Gebruikt default als cellen niet bestaan.

`CA.Neighborhoods.get_neighbors_periodiek(grid: np.ndarray, idx: list, reach: int, default: int)`: geeft een list met de toestanden van buurtcellen in een willeukeurig-dimensionale Moore-Neighborhood met reach voor de grootte van de buurt. Gaat verder vanaf de andere kant als cellen niet bestaan.

4 Iets van een discussie

Iets wat op ons opviel tijdens het afronden van de eendimensionale klasse is dat het "tekenen" van iedere nieuwe rij in de grafische weergaven steeds langzamer verliep naarmate het programma langer had gerund. Na het controleren bleek python ook een onproportioneel groot deel van de processor te gebruiken. Dit is een probleem waar we tegenaan liepen. Het probleem viel ons op toen we de code hadden toegevoegd die er voor zorgt dat wanneer de grote van de CA groter was dan het scherm, de grafische weergave wel in het midden van het scherm begon en dat de CA als het ware buiten het scherm verder loopt. Echter

leek dit niet de oorzaak te zijn.
Een vergelijkbaar probleem kwam omhoog nadat de twee dimensionale klasse af was. Een CA met de regels van Game of life en een rooster grote van bijvoorbeeld 50 bij 50 gaat zo snel als is ingesteld in de *changetime* optie. Maar hoe groter het rooster wordt ingesteld, hoe langzamer de CA gaat. Er is geen directe oorzaak gevonden, niet iets kleins wat opgeglost zou kunnen worden zonder dat een groot deel van de code herschreven zou moeten worden. Aangezien we dit probleem pas laat tegenkwamen is dit ook niet opgelost.

De volgende punten zijn niet zo zeer tekortkomingen, maar meer toevoegingen die we graag hadden gedaan.
De eerste is de mogelijkheid om als het ware over het rooster heen te kunnen bewegen waneer het rooster groter is dan het scherm. Aangezien dit niet een vereiste was voor de opdracht en het waarschijnlijk erg veel werk zou vergen, hebben we besloten dit achterwegen te laten.
Daarnaast wilde we het graag mogelijk maken om cellen tot leven te brengen door er op te klikken, naast dat je geselecteerde cellen een toestand kan geven met de *setcells* optie. Zo zou het veel makkelijker zijn om bijvoorbeeld een glider gun te maken in Game of life. Het begin van het maken van deze optie is de mogelijk om de CA op pauze te zetten.

4.1 Taak verdeling & tijdsmanagement

De taakverdeling was iets dat heel natuurlijk verliep. Buiten de werkcolleges werd er appart gewerkt aan onderdelen van de code en het verslag en in de werkcolleges werkte we veel samen. Vaak dachten we samen aan oplossingen voor een onder deel wat op dat moment relevant was en typte een persoon het uit in code. De onderdelen van het verslag zijn iets bewuster verdeeld, maar ook dit ging heel simpel. Uit eindelijk is al het werk dat nodig was voor het project niet compleet evenredig verdeeld, maar niet in mate dat het voor iemand een probleem was. Het schrijven van code gaat nou eenmaal makkelijker voor mensen met meer ervaring. Als we eerder waren begonnen was er wellicht ruimte geweest om nog meer van elkaar te leren.
Tijd was wel iets wat lichte stres heeft opgeleverd. Dit kwam omdat we te laat begonnen omdat we dachten dat het project pas aan het einde van de tentamen week ingeleverd moest worden. Ondanks dat hebben we er niet echt last van gehad.

Verder was een groot deel van onze tijd tijdens werkcolleges gebruikt voor discussies over "Steen", en het label wat hij heeft opgebouwd. Wij hebben dus een erg grote tijdscomplexiteit.