Cellulaire Automata

Jonathan Grube (2850222) en Jochem Kuijvenhoven (1432346)

Juni 2021

Opmerking vooraf

Bij het lezen en bekijken van dit verslag is het handig en verrijkend om toegang te hebben tot de code waar het verslag over gaat. Dit kan via onze publieke github met deze link: `https://github.com/JJKuijvenhoven1/Cellulaire_Automaten`

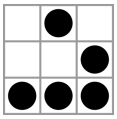
Achtergrond

De eerste cellulaire automata komen uit de jaren 40 van de vorige eeuw. Ze zijn bedacht door John von Neumann op de Los Almos National Laboratory. Hij probeerde zelf replicerende robots te maken, maar dit bleek moeilijk. Stanislaw Ulam, die daar ook werkte, kwam met de suggestie om een discreet systeem te gebruiken en zo een model van zelf replicatie te maken.
Zo'n model is opgebouwd uit een regelmatig en eindig *grid* in een eindig aantal dimensies bestaande uit een eindig aantal *cellen*. Voor elke cel zijn de *buren* relatief aan hem bepaald en is elke cel in één van een eindig aantal *toestanden*. Er worden regels gekozen die, op basis van de toestanden van de buren en van de cel zelf, de nieuwe toestand van de cel bepalen. Verder is het grid homogeen; elke cel is hetzelfde. Deze toestanden worden meestal weergegeven met de getallen $0, 1, 2, ...$ enzovoorts. Als dit allemaal gedefinieerd is dan wordt er een *starttoestand* gekozen voor elke cel. Daarna wordt in elke *generatie* of *evolutie* voor elke cel zijn nieuwe toestand bepaald volgens de regels.

Cellulaire automata bleven lang een redelijk onbekend fenomeen totdat in de jaren '70 een twee dimensionaal cellulair automaton populair werd: de zogeheten 'game of life' gemaakt door John Conway. De game of life werkt met twee toestanden: levend en dood. De regels zijn als volgt:

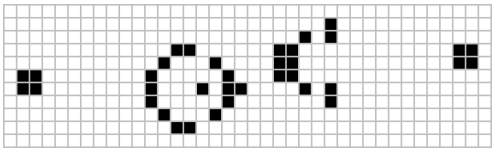
- Een levende cel met minder dan twee of meer dan drie buren gaat dood.
- Een dode cel met exact drie buren wordt levend.
- Een cel die niet in één van de bovenstaande categorieën valt, blijft hetzelfde.

De schoonheid van de game of life is dat uit deze simpele regels heel complex gedrag kan voortkomen. Een van de meest bekende bewegende vormen is de zogeheten 'glider':



figuur 1

Deze beweegt zich diagonaal over het grid. Als je vier keer dit patroon laat evolueren krijg je weer hetzelfde, maar alles is één naar rechts en één naar onder opgeschoven. Omdat de game of life erg populair is geweest hebben mensen heel veel verschillende en interessante patronen gevonden. Zo is er ook de 'glider gun'. Een opstelling die periodiek nieuwe gliders zoals hierboven maakt:



figuur 2

Dit zijn twee voorbeelden van complexe patronen die uit de simpele regels van de Game of life voortkomen. Er zijn ook nog veel meer dingen mogelijk zoals logische poorten en je kan zelfs de game of life binnen in de game of life maken! Voor meer interessante complexe patronen hebben we aan het einde van dit bestand wat linkjes toegevoegd.

De game of life is Turing complete. Dat wil zeggen dat je alles kan doen wat een Turing machine kan. In principe zou je dus alles wat je op een computer kan berekenen, ook in de game of life kunnen doen. In de praktijk worden de game of life en andere automata vaak gebruikt om verschijnselen in de echte wereld te simuleren. Een voorbeeld is een bosbrand. Dit automaton is twee dimensionaal (je kijkt als het ware van boven naar het bos) en elke cel kan drie toestanden hebben: een boom die niet verbrand is, een boom die aan het branden is en een verbrande boom. Als er op een plek al geen boom staat, kan je dat ook als een al verbrande boom zien. Je kan dan regels bedenken van hoe een brandende boom zijn nog niet verbrande buren aansteekt. Door dan je automaton te laten evolueren met verschillende startposities kan je kijken wat bijvoorbeeld de invloed is van hoe dicht bomen op elkaar staan.

Vraagstelling

In deze opdracht was aan ons de taak om een eigen cellulair automaton te schrijven en te testen. Voor ons was de voornaamste uitdaging hierbij de algemeenheid. Er bestaan vele wezenlijk verschillende cellulaire automata die allemaal net iets anders werken. Hoe meer hiervan wij willen simuleren met onze code hoe ingewikkelder het wordt en hoe meer tijd we nodig hebben. Uiteindelijk hebben we besloten dat we alles met twee types regels en een eindig dimensionaal vierkant grid willen programmeren. Met vierkant bedoelen we dat het grid in alle dimensies even lang is. De twee types regels waarvoor we geprogrammeerd hebben worden onder het hoofdstuk 'Implementatie' uitgelegd. Er was ook nog de vraag hoeveel buren en randtoestanden je toe laat. Zo heb je de game of life buren, maar je zou ook andere buren kunnen toestaan. Onze code is zo gemaakt dat alle buren toegestaan zijn zolang ze als positie ten opzichte van de cel gedefinieerd zijn. Voor de randtoestanden hebben we gekozen om periodieke, *Von Neuman*, de buren buiten het grid hebben dezelfde waarde als de cel, en meerdere soorten *Dirichlet*, de buren buiten het grid hebben een vaste waarde, randvoorwaarden toe te staan. Als laatste is er ook nog de vraag hoe je dit alles begrijpelijk weer geeft aan een gebruiker. Wij besloten dit met een elementaire UI te doen.

Algoritme

Het algoritme dat we voor het maken van onze cellulaire automata gebruikt hebben is redelijk eenvoudig en volgt direct uit de beschrijving van cellulaire automata die we eerder hebben gegeven. Het idee is dat we voor een evolutie langs alle cellen lopen en dan per cel evolueren. De code die de enkele cellen evolueert moet de toestanden van de buren van de cel bepalen, checken of die buren buiten het grid vallen en zo nodig de randvoorwaarden toepassen op de buren die buiten het grid vallen. Daarna moet het algoritme uit de toestanden van de buren de nieuwe toestand van de cel bepalen. Wij doen dit met een regelmatrix die we zo dadelijk zullen uitleggen. Als laatste slaan we dit op in een nieuw en tijdelijk grid, omdat anders een deel van het grid één generatie vooruit loopt op de rest en alle nieuwe toestanden op dezelfde generatie gebaseerd moeten zijn. Als dit op alle cellen van het grid gedaan is dan is de evolutie klaar en vervangen we het oude grid met het nieuwe grid.

1. Neem een cel C.
2. Bepaal de buren van C.
3. Als een buur B buiten het grid ligt pas dan de randvoorwaarden toe.
4. Haal de nieuwe waarde van C uit de regelmatrix.
5. Sla dit op in een nieuw grid.
6. Herhaal stap 1 totdat alle cellen behandeld zijn.
7. Vervang het oude grid met het nieuwe grid.

Nu we het algoritme helder hebben, kunnen we de complexiteit gaan bekijken. Dit kunnen we vanuit een aantal perpspectieven doen, omdat er verschillende dingen zijn die groter gemaakt kunnen worden.

Eerst bekijken we het vergroten van het aantal cellen. Als het aantal cellen verdubbeld dan moeten we twee keer zoveel cellen doorlopen. Het bepalen van de buren van C wordt met vectoroptelling gedaan en alhoewel dit technisch gezien langer duurt met grotere getallen is deze tijd verwaarloosbaar. Verder wordt er nergens anders gebruik gemaakt van de positie van C of van het aantal cellen, dus heeft ons algoritme voor het vergroten van het aantal cellen een complexiteit van $O(N)$ met N het aantal cellen. Hierbij moeten we onthouden dat het grid vierkant is en dus het aantal cellen gelijk is aan lengte tot de macht het aantal dimensies. $O(N) = O(l^d)$. Hiernaast is het ook nog interessant om het aantal buren van een cel en het aantal dimensies van het grid te verhogen. We zien dat het aantal buren verhogen ook een lineaire complexiteit heeft. Namelijk: elke buur komt twee keer per cel aan bod. Eens wanneer hij bepaald wordt en een tweede keer wanneer nagekeken wordt of hij wel binnen het grid valt.

De complexiteit van het aantal dimensies hangt alleen af van het aantal cellen dat deze extra dimensies toevoegt, want elke cel verhoogt de complexiteit weer. Dit lijkt niet interessant, maar het is wel belangrijk om te laten zien dat het aantal vakjes en dus de complexiteit snel toeneemt als je in hogere dimensies gaat werken. Zo hebben we bijvoorbeeld de klasse `simpele_hoger_dimensionale_CA`, die een grid maakt dat in alle opgegeven dimensies lengte drie heeft en dat langzaam opvult met enen. Waar een drie dimensionale versie nog 27 cellen heeft, zit je bij tien dimensionaal op $3^{10} = 59049$ waardoor dit nauwelijks meer te berekenen valt. Andere voorwaarden die nog te vergroten vallen, zoals aantal toestanden en de randvoorwaarden, hebben geen of minder als $O(N)$ invloed op de rekentijd en worden dus niet behandeld.

Het algoritme heeft dus alleen een hoge complexiteit voor het vergroten van het aantal dimensies. Hier zijn wij tevreden mee omdat, al zouden we het graag anders zien, de impact van rekenen in hogere dimensies niet echt te omzeilen valt en we het vanwege de benodigde rekentijd belangrijk vonden dat de complexiteit van het aantal cellen vergroten niet meer dan $O(N)$ was en dat is ons gelukt.

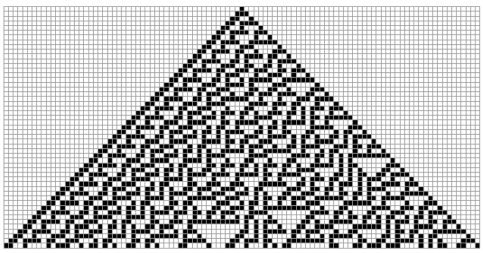
Implementatie

We gaan nu uitleggen hoe we de nieuwe toestand van een cel bepalen. Dit was een deel van de opdracht waar we duidelijk moeite mee hadden en waar we lang over nagedacht hebben, maar uiteindelijk is uit onze hersenspinsels het volgende voortgekomen: globaal gezien verdelen we de cellulaire automata in twee klassen, de *symmetrische* en de *onsymmetrische*. Bij een symmetrisch cellulair automaton kijk je alleen naar hoeveel buren van een bepaalde toestand je hebt. De game of life is een goed voorbeeld van wat wij een symmetrisch cellulair automaton noemen. De regels zijn gebaseerd op je eigen toestand en het aantal levende buren (dus die in toestand 1 zitten). Een goed

voorbeeld van een onsymmetrisch cellulair automaton is regel 30. Deze wordt ook in de opdracht besproken en is een één dimensionaal grid met twee toestanden (0 en 1), die volgens de volgende regels evolueren:

huidige toestand	nieuwe toestand middelste cel
111	0
110	0
101	0
100	1
011	1
010	1
001	1
000	0

Als we de elementen van de linker kolom als binaire getallen opvatten, zien we dat deze op volgorde van hoog naar laag staan. Door de hele rechter kolom als één binair getal op te vatten kunnen we deze regels samenvatten als het getal 30 en dus in het algemeen één dimensionale regels schrijven als een getal tussen 0 en 255. Een voorbeeld van hoe een grid met een enkele cel in toestand 1 zich evolueert onder regel 30 is dit:



figuur 3

Hier is elke rij, bovenaan beginnend, een nieuwe generatie. Het grote verschil tussen regel 30 en de game of life is dat de volgorde van de buren bij regel 30 uitmaakt. Zo is de nieuwe toestand van 110 en 011 verschillend, terwijl ze volgens symmetrische regels hetzelfde zijn. Ze zijn namelijk allebei in toestand 1 en hebben één levende buur.

Bij onsymmetrische cellulaire automata kunnen we de methode die hierboven voor regel 30 beschreven is, op een hele natuurlijke manier uitbreiden om de regels van een automaton met meer dimensies en toestanden te coderen. Je zet namelijk voor elke combinatie van toestanden de nieuwe toestand in een lijst zoals bij het voorbeeld van regel 30. Deze lijst zullen we voortaan de *regelcode* noemen. Bij het maken van de lijst sorteer je mogelijke toestanden zo dat ze optellende getallen vormen. Om dan de nieuwe toestand van een cel te bepalen moet je alleen nog maar uit de buren dit getal n bepalen en omdat de nieuwe toestanden op volgorde staan kan je nu de n^{de} entry van de regelcode nemen als je nieuwe toestand. Deze regelcode kan je weer net zoals hierboven als een getal samenvatten.

Voor symmetrische automata moet je jezelf eerst de volgende vraag stellen: waarom maken we dit onderscheid eigenlijk? Want technisch gezien kan je met de onsymmetrische codering ook symmetrische regels coderen. Zo zijn 100 en 001 hetzelfde in regel 30 en als we dit voor alle symmetrische combinaties van toestanden zouden doen dan hebben we een symmetrische regel. Het probleem is dat de lengte van de regelcode bij hogere aantallen buren en toestanden de spuigaten uitloopt. Als we het aantal buren B noemen en het aantal toestanden T dan is het aantal mogelijk combinaties van toestanden gelijk aan T^{B+1}. De plus één komt omdat je de cel als een buur van zichzelf ziet. Bij regel 30 is de lengte van de regelcode te overzien, namelijk: $2^{2+1} = 8$. De regelcode

heeft dus acht tekens als je hem binair schrijft, maar bij de game of life heeft de regelcode $2^{8+1} = 512$ tekens en dit wordt al snel slecht werkbaar. Het grootste probleem is dat het verhogen van het aantal toestanden heel slecht gaat. Stel namelijk dat je een simulatie van een bosbrand doet met dezelfde buren als de game of life en drie toestanden: levend, brandend en afgebrand. Dan heeft de regelcode lengte $3^{8+1} = 19.683$. Als we het symmetrisch zouden kunnen doen dan kunnen we de gevallen zoals 001 en 100 tot één geval samenvoegen. Dit maakt het geheel efficiënter en overzichtelijker en wordt zodra je met meer buren en toestanden werkt noodzakelijk.

Wij hebben een systeem bedacht om precies dit te doen. Stel je laat n verschillende toestanden toe (en geeft dezen een getal tussen 0 en $n-1$) en elke cel heeft b buren. Dan maak je een n-dimensionaal array, waarbij de eerste $n-1$ dimensies een lengte van $b+1$ hebben en de laatste een lengte van n. Het maakt eigenlijk niet uit welke dimensie een lengte van n heeft, maar dit is hoe wij het in ons programma gedaan hebben. Stel nu dat je een cel naar de volgende generatie wilt evolueren. Je bekijkt alle buren van die cel en telt van elke mogelijke toestand, behalve van toestand nul, hoeveel buren er daadwerkelijk in die toestand zijn. Het maakt hier weer niet uit welke toestand je weglaat en dat we de nul weglaten is gewoon een keuze die wij gemaakt hebben. Je kan altijd één toestand weglaten omdat het aantal buren in de laatste toestand, het totaal aantal buren min de som van de andere toestanden is. Dit scheelt mooi een dimensie in het array. Vervolgens gebruiken we deze getallen, in volgorde, als eerste $n-1$ coördinaten in onze zogenoemde *regelmatrix*. De laatste coördinaat wordt bepaald door de toestand van de cel zelf. De waarde die je dan afleest in de regelmatrix is de nieuwe toestand van je cel. Om het iets concreter te maken, laten we zien hoe de matrix van de Game of Life er uit ziet als een soort tabel:

	0	1
0	0	0
1	0	0
2	0	1
3	1	1
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0

Als eerste zien we dat de Game of Life met twee toestanden werkt, namelijk 0 en 1, dus ons array is dan twee dimensionaal en er zijn twee kolommen. Er zijn acht buren, dus we krijgen $8+1=9$ rijen. Als je nu wilt weten naar welke toestand een cel gaat, kijk je eerst hoeveel buren er niet in de nul toestand zijn (dus hoeveel er leven). Dat bepaald dan in welke rij je moet kijken. Vervolgens kijk je naar de toestand van de cel zelf en dat bepaalt de kolom: je kijkt bij 0 als de cel dood is en bij 1 als hij leeft. De waarde die je dan afleest is de nieuwe toestand van je cel. We zien dan in de tabel dat een dode cel alleen levend wordt als er precies drie levende buren zijn. Een levende cel blijft alleen leven als er twee of drie buren zijn. Dit is precies de Game of Life. De labels zitten er in de echte matrix in de code natuurlijk niet bij.

De voordelen van deze methode zijn dat het dus heel systematisch uit te breiden valt naar een willekeurig aantal buren en toestanden. Verder hangt de vorm of grote van het array totaal niet af van de dimensie van je automaton of welke cellen je als buren aanwijst. Het is natuurlijk bij lange na geen perfecte methode. Als je meer toestanden toevoegt zal een deel van het array niet gebruikt worden. Stel bijvoorbeeld dat je met drie toestanden werkt en acht buren. Dan kan het natuurlijk nooit voorkomen dat je zeven buren in toestand 1 hebt en drie buren in toestand 2.

Datastructuren

Voor het opslaan van deze regelcodes hebben we gekozen voor lijsten en lijsten in lijsten. Voor het opslaan van de relatieve coördinaten van de buren en de verschillende mogelijke toestanden hebben we ook gekozen voor lijsten. Dit is voornamelijk omdat wij al veel ervaring hebben in het gebruik van lijsten en ze makkelijk te manipuleren zijn. Voor het grid met de toestanden van de cellen hebben we gekozen voor een numpy array. De variabelen 'aantal dimensies' en 'randvoorwaarden' worden bijgehouden door integers, omdat dit een natuurlijk datatype is voor getallen in $\mathbb{N}$ en $\mathbb{Z}$.

We hebben er voor gekozen om van de numpy module te gebruiken. Deze staat er bekend om dat hij snel en eenvoudig met arrays kan werken. Deze arrays kunnen een willekeurig grote dimensie hebben en numpy heeft een heel systematische en overzichtelijke manier om elementen af te lezen en te veranderen en dat is wat wij nodig hebben voor ons grid. Verder kan numpy ook snel vector optelling doen en kunnen we dus snel de coördinaten van de buren van een cel berekenen door de relatieve posities van de buren en die van de cel naar een numpy array te converteren en daarna de relatieve positie van de buur en de positie van de cel bij elkaar op te tellen.

De keuze voor lijsten en lijsten in lijsten voor de regelcodes, buren en toestanden was grotendeels een gemakskeuze. We hebben dit besloten omdat we al veel ervaring hebben met lijsten en er dus vlot code mee kunnen schrijven. Verder hadden we een geordende object nodig waarin je met indexes waardes terug kan krijgen en waar je doorheen kan loopen. Aan deze eisen voldoen lijsten. Bij het opslaan van de relatieve posities van de buren was het gebruik van een numpy array logisch geweest aangezien we daar toch vector optelling mee gaan doen. Helaas hebben we te laat bedacht.

Klassen en structuur

Het was ons doel om een zo algemeen mogelijke hoofdklasse te maken waar dan de individuele cellulaire automata van afgeleid zouden worden. Daarnaast kwamen we tot de conclusie dat het logisch was om onderscheid te maken tussen symmetrische en onsymmetrische cellulaire automata, omdat die andere regelcodes gebruiken en dus ook net wat andere code nodig hebben.

De hoofdklasse heet `cellulair_automaton` en bevat een `__init__` functie die alle input opslaat, een `evolueer` functie en alle visualisatie code. De `evolueer` functie is gesplitst in twee functies: een `randvoorwaarden_en_buurtoestanden` functie die bepaalt wat de toestanden van de buren zijn, rekening houdend met de randvoorwaarden, en een `regels_toepassen` functie. Deze `regels_toepassen` functie is leeg en bedoeld om overgeschreven te worden door de onderklassen van `cellular-automaton`. Uit deze hoofdklasse leiden we twee onderklassen af. Namelijk de symmetrische en onsymmetrische klassen. Deze klassen zijn grotendeels leeg, maar overschrijven allebei de `regels_toepassen` functie met respectievelijke code om symmetrische en onsymmetrische regelcodes te interpreteren en te gebruiken. Uit deze twee onderklassen worden dan de individuele cellulaire automata afgeleid. Deze onder onderklassen bevatten alleen een `__init__` functie waarmee de standaard waarden van die CA worden vastgelegd en die daarna daarmee de `__init__` van de klasse boven zich aanroept.

Gebruik code

UI

Tijdens het testen van de code merkten we dat het heel vervelend was om de starttoestand van een grid in code in te vullen en daarom besloten richting het einde nog om met enige haast en slordigheid een elementaire user interface te maken met de pyhon module Tkinter. Hiermee kunnen eenvoudige cellulaire automata, zoals regel 30, makkelijk geïnitialiseerd worden. We hebben al deze code in een apart bestand, GUI.py, gezet om het wat overzichtelijker te houden.

Handleiding

Bij de gebruik van onze code is het de bedoeling dat de gebruiker een nieuwe python document maakt en het bestand 'cellulaire-automata.py' importeert. Dan heeft de gebruiker toegang tot alle klassen die wij gemaakt hebben. Bij gebruik van een cellulair automaton moet die eerst geïnitialiseerd worden met zijn eigenschappen:

examples_and_tests.py

```
import Cellulaire_Automata as CA
import numpy as np
matrix = np.array([[0,1,0,0,0,0,0,0,0,0],
                   [0,0,1,0,0,0,0,0,0,0],
                   [1,1,1,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   ])
gol = CA.game_of_life(startgrid=matrix,randvoorwaarden=0)

```

figuur 4

Daarna kan gebruikt gemaakt worden van afwisselend de `evolueer` en `visualiseer` functies om het cellulair automaton te evolueren en weer te geven. Er is ook de mogelijkheid om in één keer meerdere keren te evolueren en visualiseren met de `evolueer_en_visualiseer` functie.

examples_and_tests.py

```
import Cellulaire_Automata as CA
import numpy as np
matrix = np.array([[0,1,0,0,0,0,0,0,0,0],
                   [0,0,1,0,0,0,0,0,0,0],
                   [1,1,1,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   ])
gol = CA.game_of_life(startgrid=matrix,randvoorwaarden=0)
gol.evolueer(iterations=1)
gol.visualiseer()
gol.evolueer_en_visualiseer(iterations=10,timeperframe=0.5,showevery=1)

```

figuur 5

Om te zorgen dat de visualisatie correct werkt is het nodig om hierna ook nog `visual.mainlooop` aan te roepen.

examples_and_tests.py

```
import Cellulaire_Automata as CA
import numpy as np
matrix = np.array([[0,1,0,0,0,0,0,0,0,0],
                   [0,0,1,0,0,0,0,0,0,0],
                   [1,1,1,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   [0,0,0,0,0,0,0,0,0,0],
                   ])
gol = CA.game_of_life(startgrid=matrix,randvoorwaarden=0)
gol.evolueer(iterations=1)
gol.visualiseer()
gol.evolueer_en_visualiseer(iterations=10,timeperframe=0.5,showevery=1)
gol.visual.mainloop()

```

figuur 6

Als de gebruiker het te ingewikkeld vind om eigen code te schrijven, dan is het ook nog mogelijk om het bestand GUI.py te runnen of te importeren. Dit activeert dan de beperkte visuele omgeving. Hierin kunnen alleen de game of life en onsymmetrische, één dimensionale cellulaire automata gebruikt worden. Je vult de instellingen in die je wilt hebben en klikt op submit. Hierna opent een window met rechts het startgrid waarop je kan klikken om de startpositie aan te passen. Bij custom regel zit de instelling voor het regel getal ook in deze tweede window.

De verschillende klassen en cellulaire automata zijn als volgt:

- symmetrische cellulaire automata
  - `game_of_life(startgrid,randvoorwaarden)`

    Dit is de in de achtergrond uitgelegde cellulaire automata die in de jaran '70 populair werd. De inputs zijn het startgrid in de vorm van een numpy array en een integer om de randvoorwaarden aan te geven.
  - `simpele_hoger_dimensionale_CA(dimensions)`

    Deze cellulaire automaton hebben we gemaakt als voorbeeld van hoe onze code zou omgaan met cellulaire automaten van hogere dimensies. De enige input is het aantal dimensies.
  - `symmetrische_CA(dimensions,startgrid,randvoorwaarden,burenlijst, toestanden,regelcode)`

    Dit is de algemene symmetrische cellulaire automata, hierin kan alles aangepast worden, maar dit is alleen aan te raden aan gebruikers die weten wat ze willen.
- onsymmetrische cellulaire automata
  - `regel30(startgrid,randvoorwaarden)`

    Dit is de reeds bekende een dimensionale CA met regelgetal 30. Het startgrid in de vorm van een een dimensionaal numpy array moet worden meegegeven net zoals het randvoorwaarden getal.
  - `customregel(startgrid,randvoorwaarden,regelcode)`

    Deze functie werkt net als de regel 30, maar je kan het getal, dus de vervanging van 30, invoeren in regelcode en dan wordt dat geconverteerd naar een bijbehorende regelcode.
  - `onsymmetrische_CA(dimensions,startgrid,randvoorwaarden,burenlijst, toestanden,regelcode)`

Dit is de algemene onsymmetrische cellulair automata waarin alles aangepast kan worden. Deze dient alleen aangeroepen te worden als de andere onsymmetrische klasses niet voldoen.

- `cellulair_automaton(dimensions,startgrid,randvoorwaarden,burenlijst, toestanden,regelcode)`

  Dit is de hoofdklasse waarvan alles is afgeleid. Deze dient nooit aangeroepen te worden.

Dimensions behoort een integer groter als nul te zijn. Startgrid behoort een vierkant numpy array te zijn. randvoorwaarden behoort een integer met waarde -2, -1, 0, 1, 2... enzovoort te zijn. Burenlijst behoort een lijst met daarin lijsten met daarin de relatieve coördinaten van de buren te zijn. Toestanden behoort een lijst van de vorm [0, 1, 2...] te zijn waarbij je zoveel getallen neerzet als je toestanden wil. De regelcode is ergens anders uitgelegd.

Nog een laatste opmerking: Mocht je een backend error krijgen in spyder, dan is het van belang dat je in tools>preferences>console>graphics>backend inline selecteert en daarna een nieuwe console opent. Dit zorgt ervoor dat Tkinter de backend kan overwriten.

Voorbeelden

Voor gebruikers die geen zin of kennis hebben om uitvoerig gebruik te maken van onze klassen, hebben wij in een apart bestand genaamd `example_en_tests` een aantal voorbeelden gezet. Deze hebben voor onszelf tevens gediend als unit tests om er zeker van te zijn dat de verschillende klassen naar behoren werken. Een van deze voorbeelden is een invulling van de algemene symmetrische klasse waarmee het rekenwerk achter een hexagonaal grid gedaan kan worden. Voor de visualitsatie van dit hexagonale grid was helaas geen tijd meer en dus laten we dit als oefening aan de lezer.

Conclusie

Cellulaire automata zijn een interessant voorbeeld van complex gedrag dat ontstaat uit eenvoudige regels en geheel in dit thema is dit project ook iets complex dat begon als iets simpels. Het evolueren van een cellulair automaton laat zich reduceren tot een 7-staps algoritme, maar bij het uitvoeren van dat algoritme komt veel code kijken. Ons doel was om de code zo algemeen mogelijk te schrijven. Eén van de problemen die we tegenkwamen was de lengte van onsymmetrische regelcodes. Dit hebben we opgelost met een zelfontworpen regelmatrix. Verder besloten we tegen het einde aan dat het nodig was om een UI te maken voor het invoeren van matrices.

Discussie en mogelijke uitbreidingen

Al met al zijn we tevreden over hoe het project is gegaan. Wij vinden dat de indeling in klassen en de structuur erg sterk geworden is, omdat je makkelijk andere soorten automata kan afleiden van de klassen die we hebben. Ook hebben we het proces van het evolueren van een cellulair automaton goed beschreven in een algoritme. Verder zijn de wiskundigen in ons erg tevreden met de systematische en uitbreidbare manier waarop onze code werkt. Waar we niet zo tevreden mee zijn, is de UI. We hebben er veel van geleerd, maar we zijn er te laat mee begonnen om het gewenste resultaat te bereiken. We wilden namelijk ook het opzetten van een algemeen, symmetrisch twee dimensionaal automaton in de UI mogelijk maken. We dachten dan dat we de regelmatrix ook vrij makkelijk door de user zouden kunnen laten invoeren, op een soortgelijke manier als hoe we nu het startgrid van de game of life bepalen. Ook het toelaten van meerdere toestanden hadden we graag gezien en was waarschijnlijk ook mogelijk als we meer tijd hadden. Achteraf gezien is het maken van een grid met buttons, zoals we het nu doen voor de game of life, misschien toch niet zo ideaal voor grotere grids. Een beter plan zou zijn om, als je ergens op een soort canvas klikt met

de muis, de muiscoördinaten op te vragen en aan hand daarvan een paar pixels te kleuren. Verder zou het handig zijn, en het gebruiksgemak vergroten, als we ook driedimensionale matrices visueel weer kunnen geven, in plaats van met een print statement. Als laatste zou het interessant zijn om te onderzoeken wat voor optimalisaties er mogelijk zijn om de code sneller te maken. Wij hebben er zelf geen algemene bedacht, maar in specifieke gevallen, zoals game of life, is zeker veel mogelijk. Zo kan je in het stargrid blokken van cellen die allemaal in toestand nul zitten, identificeren. Aangezien je weet dat een blok van nullen gewoon hetzelfde blijft, hoef je dus voor een deel van je grid een tijd lang niet extra uit te rekenen wat er met de cellen gebeurt. Als je kleine patronen in een groot grid bekijkt, kan dit veel rekentijd besparen.

Taakverdeling

We hebben het overgrote deel van de code en het verslag samen gemaakt. Vaak hebben we in Teams een meeting gestart en streamde iemand zijn scherm. Zo hebben we de wat ingewikkeldere code, zoals voor evolueren van een enkele cel, aangepakt. Jonathan heeft meer aan de UI gewerkt en de regelmatrix voor het coderen van symmetrische automata. Jochem heeft meer aan het systeem van klassen en de visualisatie gewerkt. In het verslag heeft Jonathan wat meer aan de implementatie, de opmaak en netheid en de conclusie gedaan. Jochem heeft meer aan de achtergrond, het algoritme en de datastructuren geschreven. We hebben ook via de github samengewerkt en dit ging prima.

Leuke linkjes

De game of life binnen in de game of life:
`https://www.youtube.com/watch?v=xP5-iIeKXE8`
Een verzameling patronen voor de game of life:
`https://www.youtube.com/results?search_query=game+of+life`
De wiki voor de game of life:
`https://www.conwaylife.com/wiki/Main_Page`
logic gates in de game of life:
`https://www.youtube.com/watch?v=vGWGeund3eA`
Regel 30 en regel 90:
`https://www.youtube.com/watch?v=3MJ8deSCOCE`
Leuke patronen in verschillende automata:
`https://www.youtube.com/watch?v=VaWEKIbFKCg`
De generaties van regel 30 als input voor de game of life (met leuke kleurtjes):
`https://www.youtube.com/watch?v=IK7nBOLYzdE`
.

Bronvermelding

figuur 1:
`https://nl.wikipedia.org/wiki/Glider_(Game_of_Life)`
figuur 2:
`https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`
figuur 3:
`https://writings.stephenwolfram.com/2019/10/announcing-the-rule-30-prizes/`