UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Iztok Jeras

Predslike 2D celičnih avtomatov

MAGISTRSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Ljubljana, 2016

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

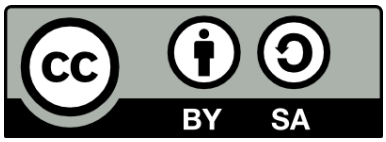
Iztok Jeras

Predslike 2D celičnih avtomatov

MAGISTRSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Branko Šter

Ljubljana, 2016



Besedilo je oblikovano z urejevalnikom besedil LaTeX.
Slike in grafi so narisani s pomočjo programa za vektorske ilustracije Inkscape [4].

[1] http://creativecommons.org/licenses/by-sa/4.0/
[2] http://www.ipi.si/sl/creative-commons-cc/o-uporabi-licence
[3] http://unlicense.org/
[4] https://inkscape.org

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Večna pot 113
1000 Ljubljana, Slovenija
telefon: 01 47 98 100
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Številka: 159-MAG-RI/2016
Datum: 06. 04. 2016

Iztok JERAS, univ. dipl. inž. el.

L j u b l j a n a

Fakulteta za računalništvo in informatiko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **Predslike 2D celičnih avtomatov**

Preimages of 2D cellular automata

Tematika naloge:

Raziščite problem iskanja predslik 2D celičnih avtomatov s pomočjo De Bruijn-ovih diagramov. Pri tem se opirajte na znane rešitve za 1D problem. Poiščite algoritem za določitev obstoja predslik in algoritem za njihovo preštevanje ter izpis.
Algoritem naj bo matematično formuliran z enačbami in grafično predstavitvijo ter implementiran s programsko opremo. S stališča procesne kompleksnosti primerjajte novo razviti algoritem z obstoječimi algoritmi, ki se uporabljajo za iskanje stanj tipa 'Garden of Eden' v celičnem avtomatu 'Game of Life'.

M e n t o r :

prof dr. Branko Šter

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

D e k a n :

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU
magistrskega dela

Spodaj podpisani **Iztok Jeras**, z vpisno številko **63030393**, sem avtor magistrskega dela z naslovom:

Predslike 2D celičnih avtomatov

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod vodstvom mentorja **prof. dr. Branka Štera**,
- so elektronska oblika magistrskega dela, naslova (slov., angl.), povzetka (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela
- in soglašam z javno objavo elektronske oblike magistrskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 30. avgust 2016 Podpis avtorja:

Zahvala

Zahvaljujem se prof. dr. Andreju Dobnikarju za pomoč pri pisanju člankov o algoritmih za štetje in izpis predslik pri enodimenzionalnih celičnih avtomatih.

Kazalo

Seznam uporabljenih kratic in simbolov

1D ..... enodimenzionalen

2D ..... dvodimenzionalen

3D ..... tridimenzionalen

CA ..... celični avtomati (angl. cellular automata)

GoL ..... Conwayeva igra življenja (angl. Conway's Game of Life)

GoE ..... rajski vrt (angl. Garden of Eden)

trid ..... okolica CA, sestavljena iz treh celic na heksagonalni mreži

quad ..... okolica CA, sestavljena iz štirih celic na kvadratni mreži

DFS ..... iskanje v globino (angl. depth first search)

SAT ..... problem izpolnjivosti (angl. boolean satisfiability problem)

C ..... poljubna konstanta

S ..... nabor stanj celice

$|S|$ ..... število možnih stanj celice

c ..... stanje posamezne celice (celoštevilska vrednost)

$c(x, y)$ ..... stanje celice na koordinatah (x, y) znotraj 2D polja celic

c^t ..... stanje celice v sedanjosti

c^{t+1} ..... stanje celice v prihodnosti (en korak)

N_x ..... velikost pravokotnega 2D polja celic v dimenziji X

N_y ..... velikost pravokotnega 2D polja celic v dimenziji Y

N ..... število celic v 1D ali 2D polju

M_x ..... velikost pravokotne 2D okolice celice v dimenziji X

M_y ..... velikost pravokotne 2D okolice celice v dimenziji Y

M ..... število celic v 1D ali 2D okolici

n ..... stanje okolice posamezne celice (celoštevilska vrednost)

$n(x, y)$ ..... stanje okolice celice na koordinatah (x, y) znotraj 2D polja celic

n^{t-1} ..... stanje okolice celice v preteklosti (en korak)

n^t ..... stanje okolice celice v sedanjosti

f ..... tranzicijska funkcija, ki definira časovno evolucijo avtomata

f^{-1} ..... obratna tranzicijska funkcija

$o_{\leftrightarrow}$ ..... prekrivanje okolic v dimenziji X

$o_{\updownarrow}$ ..... prekrivanje okolic v dimenziji Y

$o_{\times}$ ..... prekrivanje okolic v diagonalni smeri

Povzetek

Medtem ko je računanje predslik 1D celičnih avtomatov dobro raziskan in podrobno dokumentiran problem, je to področje pri 2D celičnih avtomatih manj raziskano. Za 1D problem poznamo algoritme za štetje in izpis predslik s procesno zahtevnostjo linearno odvisno od velikosti problema. Možno je tudi določiti, ali je 1D celični avtomat reverzibilen in kakšen je regularen jezik vseh stanj brez predslik. Pri 2D problemu sicer poznamo nekaj algoritmov za iskanje predslik, vendar so slabo teoretično raziskani. Vemo tudi, da je problem reverzibilnosti 2D celičnih avtomatov na splošno neodločljiv.

Mreža predslik, ki sem jo v preteklosti razvil za potrebe analize predslik 1D celičnih avtomatov, se je izkazala za praktično orodje pri razlagi algoritmov in pri dokazovanju njihove pravilnosti. Tukaj opišem kako se mrežo predslik tvori za 2D celične avtomate. Če je bila ta mreža za 1D problem navaden graf, je za 2D problem razširjena v tretjo dimenzijo. Predslike so namesto poti v grafu ploskve na mreži. Robni pogoj se spremeni iz uteži za vozlišča, ki zaključujejo pot v 1D problemu, v uteži za sklenjeno pot okoli ploskve predslike v 2D problemu.

Pri razvoju algoritma se je izkazalo, da štetja predslik 2D celičnega avtomata ni mogoče izvesti v linearni odvisnosti od velikosti problema. Zahtevnost podanega algoritma narašča eksponentno z velikostjo problema v eni dimenziji. Podani algoritem se ne razlikuje dosti od obstoječih. Celični avtomat razdeli na vrstice in išče predslike za posamezno vrstico od prve do zadnje. Vrstične rezultate nato združi v rešitev za celoten 2D problem. Podobno kot pri algoritmu za 1D probleme je analiza razdeljena v dva prehoda. V prvem prehodu po vrsticah algoritem prešteje predslike, v drugem opcijskem prehodu pa predslike izpiše. Drugi prehod poteka v obratni smeri kakor prvi prehod. Tudi procesiranje posamezne vrstice poteka iz drugega zornega kota.

Glavni napredek algoritma vidim v uporabi učinkovitih rešitev za 1D problem pri analizi vrstic ter v uporabi progresivnega kodiranja vmesnih rezultatov, kar lahko zmanjša porabo pomnilnika.

Ključne besede:

celični avtomati, predslike, predhodniki, računska zahtevnost, reverzibilnost, rajski vrt, Conwayeva igra življenja, trid, quad

Abstract

While computing preimages of 1D cellular automata is a well researched and documented problem, for 2D cellular automata there is less research available. For the 1D problem we know algorithms for counting and listing preimages where computational complexity is a linear function of the size of the problem. It is possible to determine whether a 1D cellular automaton is reversible, and what is the Garden of Eden sequence regular language. For the 2D problem we know a few algorithms, but they are poorly theoretically researched. We also know that the reversibility problem is in general undecidable for 2D cellular automata.

The preimage network, first developed for 1D cellular automata, was proved to be a useful tool for explaining algorithms and for constructing proofs. Here I explain how to construct the preimage network for 2D cellular automata. While for the 1D problem this network is a normal graph, for 2D it was extended into the third dimension. Preimages are transformed from paths in the graph in 1D into surfaces on the network in 2D. Edge conditions are transformed from weights for vertices ending a path in the 1D problem into weights for the closed path around a preimage surface in the 2D problem.

While developing the algorithm, it proved impossible to count preimages of 2D cellular automata with processing requirements growing linearly with problem size. Instead, processing requirements grow exponentially with the size in one of the dimensions. The described algorithm does not differ much from the existing ones. The cellular automaton is split into rows, the preimage list is first determined for each row from the first to the last. The row results are then combined into the result for the whole 2D problem. In a similar fashion to the 1D approach, the algorithm splits into two passes. In the first pass preimages are counted, in the second optional pass preimages are listed. The second pass is performed in the opposite direction, while rows are also observed from the opposite side.

I see the main advantage of the described algorithm in using existing solutions for row processing. Solutions proved to be effective in solving the 1D problem. Using progressive encoding of intermediate solutions also enables reducing memory consumption.

Key words:

cellular automata, preimages, predecessors, ataviser, computational complexity, reversibility, Garden of Eden, Conway's Game of Life, trid, quad

Poglavje 1

Uvod

1.1 Motivacija

1.1.1 Celični avtomati kot model vesolja

Ker lahko vsak univerzalen sistem modelira vsak drug univerzalen sistem, lahko predpostavimo, da lahko z univerzalnimi *celičnimi avtomati* (angl. *cellular automata, CA*) modeliramo vesolje. Samo modeliranje vesolja je še izven našega dosega, poizkuša pa se vsaj približati teorijo CA teoretični fiziki. S strani informacijske teorije in termodinamike je predvsem zanimiv model gravitacije kakor entropijske sile (angl. Entropic gravity) [42], ki predpostavlja, da je 3D vesolje projekcija procesov, ki se odvijajo na 2D ploskvi. Z druge strani pa CA tudi omogočajo opazovanje abstraktnega kopiranja informacij (replikacija) in evolucije [39]. Oba sta pomembna informacijska pojava v našem vesolju.

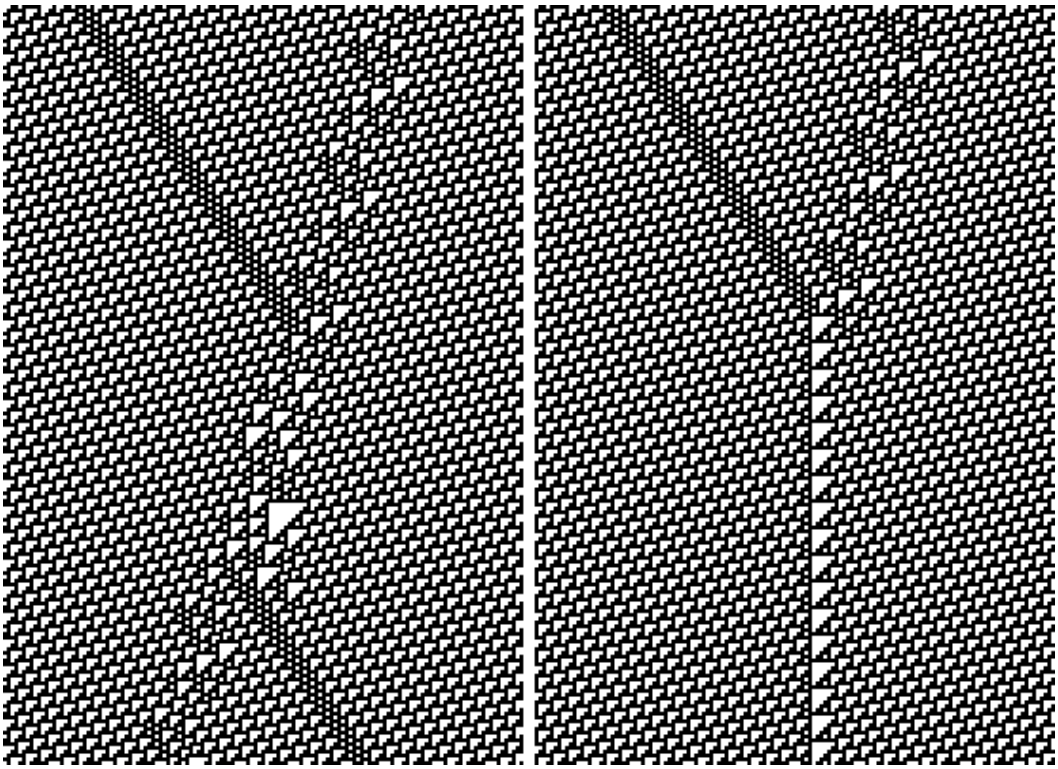
1.1.2 Informacijska dinamika

Informacijsko dinamiko CA se najpogosteje opisuje samo kakor reverzibilno ali ireverzibilno. Obstaja tudi nekaj člankov, ki opazujejo entropijo sistema. Pri reverzibilnem avtomatu se vsa informacija ohranja. Vsako stanje ima natanko eno prihodnost in eno preteklost. Za reverzibilen celični avtomat lahko tudi definiramo pravilo oziroma funkcijo, s katero preprosto procesiramo avtomat v preteklost namesto v prihodnost. Za reverzibilne CA je torej računanje predslik (preteklosti) trivialno in ne potrebuje tukaj opisanega algoritma. Za ireverzibilne CA tako reverzno pravilo ne obstaja. Trenutno stanje ima lahko nobeno, eno, ali več predslik. Opisani algoritem je namenjen štetju in izpisu predslik za dano sedanje stanje CA. Za take avtomate pravimo, da s časom izgubljajo informacijo, ker ne moremo enolično določiti preteklega stanja.

Pogosto je tudi opazovanje dinamike delcev pri *Igri življenja* (angl. *Game of Life, GoL*) [7] (slika 1.1) in elementarnem pravilu 110 [12] (slika 1.2). Pištole, trki delcev in podobni konstrukti obstajajo tako v GoL kakor pri elementarnem pravilu 110 in na splošno pri

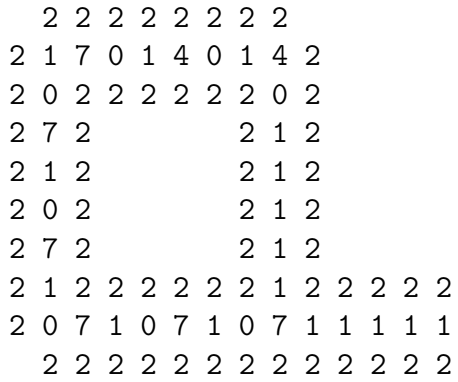
vsakem univerzalnem celičnem avtomatu. Pištole (slika 1.1) oddajajo delce, izvor in oddani delci skupaj rastejo v neskončnost. S tem se informacija, potrebna za opis sistema, povečuje. Slika 1.2 prikazuje dve različni interakciji med dvema delcema. V levem primeru se delca srečata, si nekaj časa delita skupen prostor in nato nadaljujejta pot vsak zase. V desnem primeru se delca srečata in po trku nastane nov delec. Veliko poklicnih in amaterskih raziskovalcev proučuje delce v GoL in njihovo dinamiko. S pomočjo osnovnih gradnikov je mogoče skonstruirati kompleksnejše sisteme, med katerimi so najzanimivejši Turingov stroj [38] in univerzalni konstruktor [20].

Slika 1.1: Gosperjeva pištola z nekaj izstreljenimi drsalci.



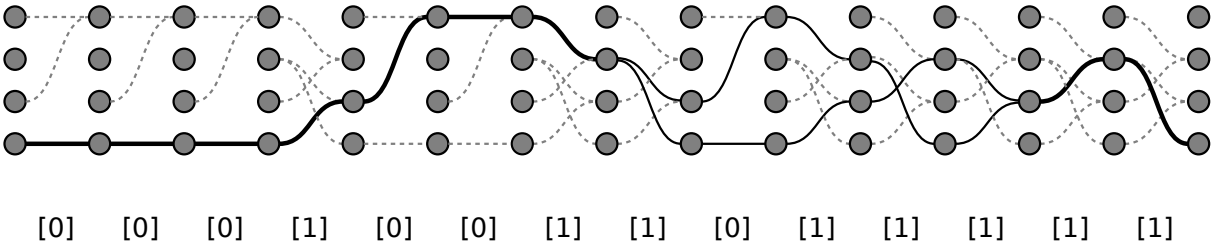
Slika 1.2: Trk para delcev v pravilu 110. Razlika v začetnem stanju med slikama je v razdalji (fazi) med delcema.

Celične Avtomate sta si zamislila matematika John von Neumann in Stanislaw Ulam, da bi lahko oblikovala teorijo univerzalnega konstruktorja [13], ki je sposoben iz zapisa na traku narediti kopijo samega sebe. Christopher Langton je vzel del univerzalnega konstruktorja in ga poenostavil v preprosto zanko [11], ki kopira samo sebe na podlagi informacije zapisane v notranjosti telesa (slika 1.3).



Slika 1.3: Langtonova zanka. Številke so stanja posamezne celice. Spodaj desno iz zanke raste nova zanka.

Zgoraj našteti primeri nekako opisujejo informacijsko dinamiko v CA. Ne obstaja pa še splošna teorija dinamike informacij v CA, ki bi dinamiko opisala kvantitativno in prostorsko. V svojem članku [30] in prispevkih na konferencah [25, 27, 29] sem grafično upodobil predslike trenutnega stanja za 1D problem. Iz upodobitve (slika 1.4) je videti, da se ponekod izgubi več informacije kakor drugod, kar kaže na možnost izpeljave splošne teorije dinamike informacij. Na žalost se ta možnost še ni udejanila. Podobno je možno grafično upodobiti predslike 2D CA, ter iz grafov sklepati o izgubi informacije v 2D CA.



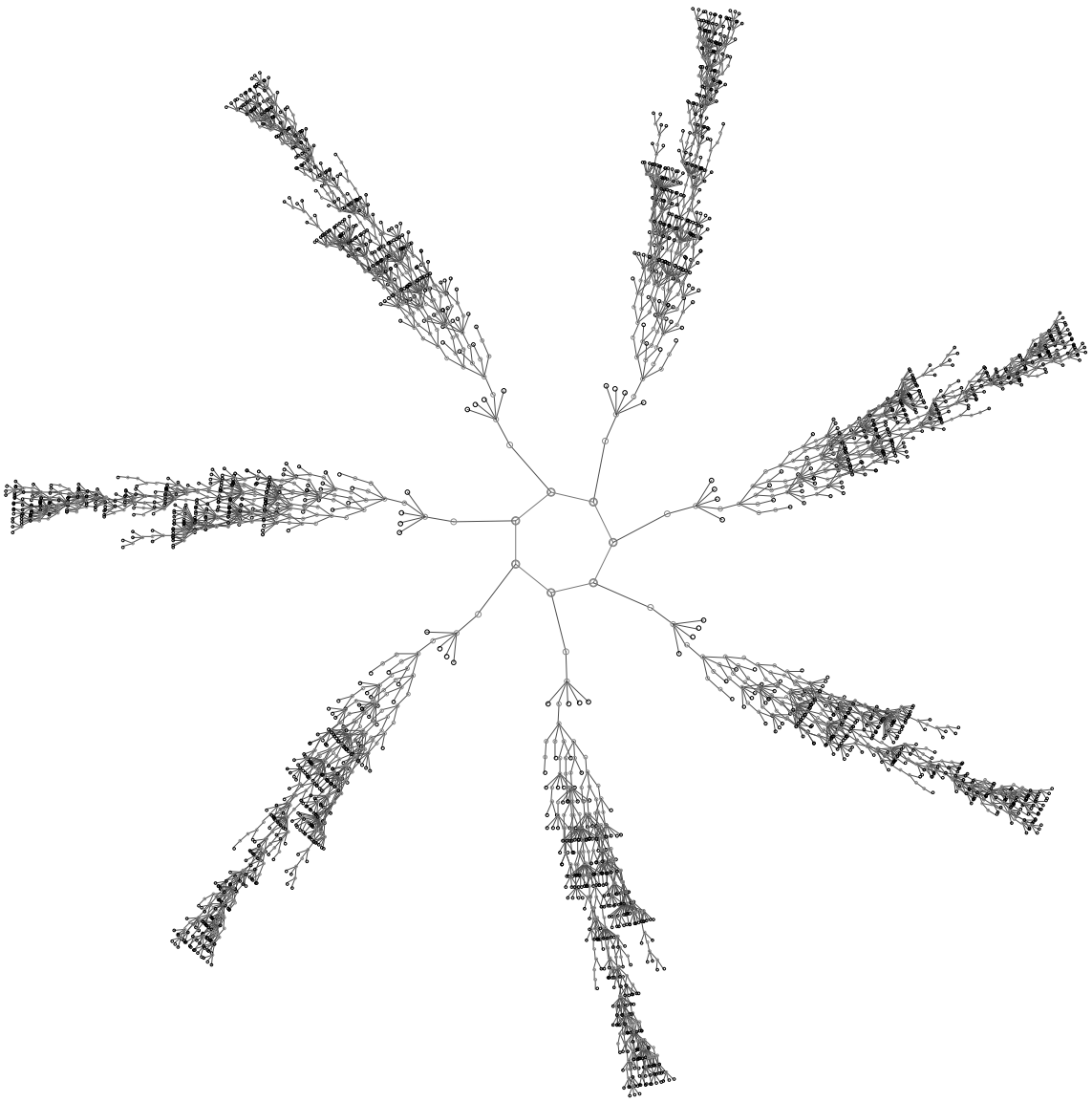
Slika 1.4: Mreža predslik 1D CA (pravilo 110) za tiho ozadje s cikličnim zaporedjem 00010011011111. Vsaka od dveh poudarjenih poti med levim in desnim predstavlja predsliko. V enem delu sta si predsliki enaki, drugod se razlikujeta. Laično lahko sklepamo, da se tam, kjer se predsliki razlikujeta, izgubi 1 bit informacije.

1.1.3 Atraktorjevo korito

Pomembno orodje za analizo časovno in prostorsko diskretnih dinamičnih sistemov je *atraktorjevo korito* (angl. *basin of attraction*) (slika 1.5). Andrew Wuensche že leta

proučuje atraktorjeva korita naključnih binarnih mrež in celičnih avtomatov [46, 47]. To je graf, kjer so vozlišča stanja cikličnega končnega CA, usmerjene poti pa povezujejo vsako stanje z njegovim časovnim naslednikom. Vsa stanja se lahko zberejo v eno ali več korit, odvisno od sistema. V vsakem CA, ki ni lokalno injektiven, se pojavljajo stanja brez predslik, imenovana *rajski vrt* (angl. *Garden of Eden, GoE*) [34, 35]. Stanja GoE so listi v grafu korita.

Sam atraktor je cikel v grafu korita. To je časovno periodično stanje, v katerem se končna časovna evolucija vsakega končnega diskretnega dinamičnega sistema. Cikel lahko vsebuje eno ali več stanj, njihovo število imenujemo perioda atraktorja. Atraktor predstavlja tudi stabilne objekte v neskončnem CA. Za GoL obstajajo katalogi takih objektov, kjer so ti urejeni po velikosti, periodi in hitrosti premikanja. Kot primer bi navedel *mrtvo ozadje* (angl. *quiescent background*) (perioda 1) in *drsalca* (angl. *glider*) (perioda 4).



Slika 1.5: Atraktorjevo korito za elementarno pravilo 110 in konfiguracijo 00010011011111 znotraj atraktorja. Korito vsebuje osrednji cikel atraktorja ter drevesa, ki rastejo iz cikla in se končujejo z listi (stanja GoE). Slika je narejena s programom `www.ddlab.com`.

1.2 Algoritmi za štetje in izpis predslik CA

Doslej sem že razvil napredne algoritme za štetje in izpis predslik 1D CA [30]. Skozi zgodovino so taki algoritmi napredovali, tako da je padala njihova računska zahtevnost in opisna/implementacijska zahtevnost:

1. algoritmi s *surovo silo* (angl. *brute force*) [6] imajo zahtevnost $O(C^N)$,
2. algoritmi z *iskanjem v globino* (angl. *depth first search, DFS*) [9],
   algoritmi s *sestopanjem* (angl. *backtacking*) [4],
   algoritmi, ki rešujejo *problem izpolnjivosti* (angl. *Boolean satisfiability problem, SAT*) [5],
3. optimalni algoritmi, kjer sta izpis in štetje strogo ločena.

Bistvo optimalnega algoritma je, da je proces štetja predslik optimalen. Za 1D CA je to doseženo, saj je zahtevnost štetja linearno odvisna od velikosti problema $O(N)$ (N je število opazovanih celic). Raziskave algoritma za štetje predslik 2D CA sem se lotil s predpostavko, da je tudi tu možno opraviti štetje predslik z linearno zahtevnostjo. Izkaže se, da 2D problem ni tako preprost. Predstavljeni so primeri, iz katerih je razvidno, da algoritem z linearno zahtevnostjo ne more pravilno opisati vseh situacij. Zahtevnost opisanega algoritma sicer raste eksponentno z velikostjo ene od dimenzij polja CA $O(C^{N_x})$, in linearno z velikostjo druge dimenzije $O(N_y)$ (C je konstanta, odvisna od števila stanj celice in velikosti okolice, ne pa tudi od velikosti polja). Ker pa nisem dokazal optimalnosti opisanega algoritma za štetje, dopuščam možnost, da obstaja algoritem z nižjo kompleksnostjo.

Proces izpisa predslik ima vedno eksponentno rastočo komponento. Ker število vseh stanj sistema raste eksponentno z velikostjo problema, posledično tudi povprečje števila predslik skozi vsa stanja raste eksponentno. Optimalen algoritem za izpis predslik je časovno in pomnilniško linearno odvisen od števila predslik. Moja algoritma za izpis predslik 1D in 2D CA sta optimalna in omogočata izpis predslik brez slepih poti, kar je posledica podrobne analize problema ob štetju.

Ostali znani algoritmi spadajo v drugo kategorijo. Večinoma izkoriščajo le lokalno znanje o sistemu (lokalno štetje predslik), ne pa tudi globalnih števcev. Zato zahajajo v slepe poti in se morajo vračati v prejšnje stanje ali pa opuščajo neustrezne rešitve, katerim so že namenili procesni čas.

Pomembno vlogo pri razvoju opisanega algoritma ima mreža predslik. To je grafična upodobitev problema, katere cilj je lažje razumevanje problema in rešitve. Osnova za oblikovanje mreže predslik so De Bruijnovi diagrami. Posamezen De Bruijnov diagram je mreža predslik ene celice. Mreže posameznih celic se nato povezujejo, tako da na koncu opisujejo celotno polje celic.

McIntosh in njegovi učenci so bili pobudniki uporabe De Bruijnovih diagramov za analizo 1D CA [33]. Uporabljali so jih za štetje in izpis predslik ter za analizo delcev in njihovih interakcij. Paulina Léon in Genaro Martínez (McIntoshev učenec) [32] sta začetnika apliciranja De Bruijnovih diagramov na 2D CA, vendar jih ne uporabljata za štetje in izpis predslik.

Največ raziskav s področja predslik 2D CA je bilo opravljenih ravno s ciljem iskanja stanj GoE v avtomatu GoL. S stališča algoritma za štetje predslik je stanje GoE tako stanje, katerega število predslik je nič. Algoritem za štetje predslik je možno pretvoriti v manj zahteven algoritem za preverjanje, ali je dano stanje stanje GoE, tako da se operacije nad celimi števili pretvori v logične operacije nad Boolovimi stanji.

1.2.1 Implementacija algoritma

Algoritem je implementiran kot računalniški program v jeziku C [26]. Knjižnica GMP [1] je uporabljena za zapis celih števil, večjih od 64 bitov. Poleg samega algoritma za štetje in izpis predslik sem pripravil tudi orodje za simulacijo binarnega CA z okolico quad [28], ki temelji na simulatorju za GoE [45]. Simulator se lahko zažene kar v internetnem brskalniku.

1.2.2 Primeri in ilustracije

Večina primerov in ilustracij opisuje binarni 2D CA z majhno okolico quad (štiri celice na kvadratnem polju, Toffoli 2008 [41]). Večina primerov in sploh ilustracije za GoL bi bile zaradi $2^{3 \cdot 3} = 512$ možnih stanj okolice preveč kompleksne in nepregledne (glej dodatek A). V nasprotju ima binarna okolica quad le $2^{2 \cdot 2} = 16$ možnih stanj.

Želel sem sicer uporabiti kako določeno pravilo za okolico quad, tako ki bi omogočalo zanimivo dinamiko delcev, vendar na tem področju še ni kaj dosti raziskav. Obstaja le dokaz univerzalnosti za binarni avtomat z okolico trid (Powley 2008 [37]).

Na slikah je uporabljena izometrična projekcija, saj je za potrebe analize osnovnemu 2D polju dodana tretja dimenzija, ki opisuje prostor predslik (mreža predslik).

1.3 Pregled vsebine

V poglavju 2 podajam definicijo 1D in 2D CA. V poglavju 3 opišem konstrukcijo mreže predslik za 2D CA in razložim njen odnos z naborom predslik za dano konfiguracijo. V poglavju 4 opišem na novo razvit algoritem za štetje in izpis predslik ter razložim, zakaj algoritem z linearno zahtevnostjo ni možen. V poglavju 5 opravim pregled obstoječih algoritmov ter jih v poglavju 6 primerjam s svojim algoritmom. Tukaj še navedem doprinos tega magistrskega dela k področju celičnih avtomatov in naštejem nekaj problemov, ki bi jih rad raziskal v prihodnje. V dodatku B navedem primer delovanja algoritma in v dodatku D navedem izvorno kodo algoritma.

[1] The GNU Multiple Precision Arithmetic Library `https://gmplib.org/`

Poglavje 2

Definicija celičnih avtomatov

V delu se omejujemo na celične avtomate, ki so:

- prostorsko diskretni (celice),
- časovno diskretni (korak),
- homogeni (pravila so enaka za vse celice in vse celice se posodobijo naenkrat) in
- deterministični (novo stanje je odvisno le od trenutnega stanja).

Vsaka celica ima diskretno vrednost c iz nabora stanj celice S. Stanja so oštevilčena:

$$c \in S \quad \text{in} \quad S = \{0, 1, \ldots, |S| - 1\} \tag{2.1}$$

Za potrebe implementacije algoritma je pomembno, da smo stanje elementov CA, kakor so pravilo, vrednost okolice in podobno, sposobni indeksirati. Uporabljena je metoda, ki je bila v osnovi namenjena zapisu pravila. Posplošeno, vrednosti celic $c(i)$ so zložene v niz $A = [c(0), c(1), \ldots, c(|A| - 1)]$ dolžine $|A|$, ki je interpretiran kakor $|A|$ mestno pozitivno celo število num v S-iškem številskem sestavu:

$$num = \sum_{i=0}^{i=|A|-1} |S|^i \cdot c(i) \tag{2.2}$$

Za podane primere so uporabljeni binarni CA kjer je $S = 2$, tako da so števila v dvojiškem sestavu.

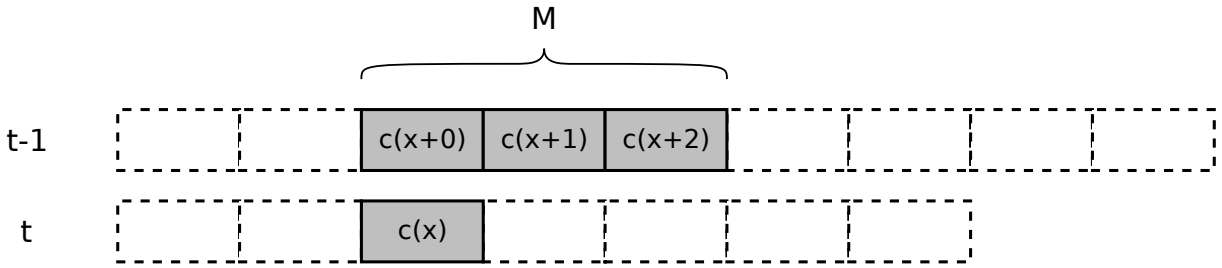
2.1 Definicija 1D celičnih avtomatov

Definicijo 1D CA povzamemo po [30]. Za 1D CA se celice združujejo v regularen niz. Na splošno je dolžina niza lahko neskončna, bolj običajni pa so končni nizi dolžine N. Nize zapisujemo z malimi grškimi črkami α ali β, kjer je $c(x)$ celica na koordinati x:

$$\alpha = [c(0), c(1), \ldots, c(N - 1)] \tag{2.3}$$

2.1.1 Okolica

Za potrebe enostavnosti implementacije se okolica celice širi le v pozitivni smeri (slika 2.1). Torej je za celico $c(x)$ njena okolica velikosti M enaka $n(x) = \{c(x), c(x+1), \ldots, c(x+M-1)\}$. V praksi se podana definicija okolice razlikuje od bolj običajne centrirane okolice le v tem, kako je časovno sosledje stanj poravnano v grafični upodobitvi. Nekoliko neobičajno definicijo sem uporabil zato, ker ne potrebuje ločene interpretacije za okolice sode velikosti. Hkrati sem se tako izognil uporabi negativnih števil v implementaciji algoritma.



Slika 2.1: Okolica 1D CA velikosti $M = 3$.

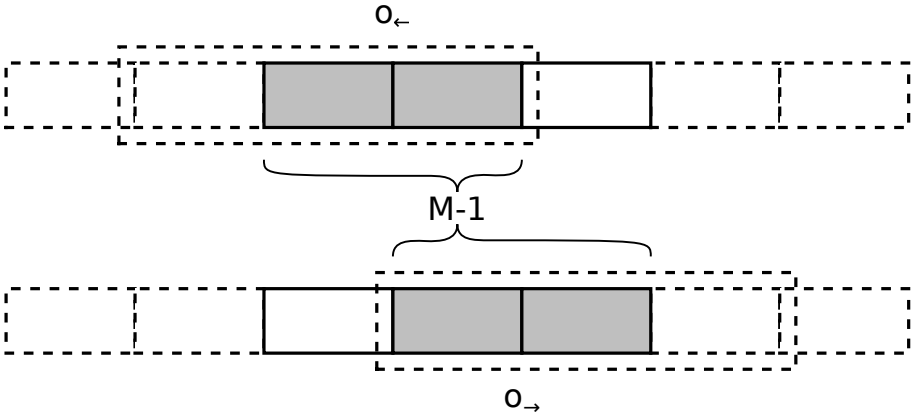
Indeks okolice $n(x)$ in nabor možnih okolic sta:

$$n(x) = \sum_{i=0}^{i=M-1} |S|^i \cdot c(x+i) \tag{2.4}$$

$$n \in \{0, 1, \ldots, |S|^M - 1\} \tag{2.5}$$

2.1.2 Prekrivanje okolic

Dana okolica $n(x)$ se za $M-1$ celic prekriva z okolicama sosednjih celic (slika 2.2). Na levi imamo prekrivanje $o_{\leftarrow}(x)$ med okolicama $n(x-1)$ in $n(x)$. Na desni imamo prekrivanje $o_{\rightarrow}(x)$ med okolicama $n(x)$ in $n(x+1)$.



Slika 2.2: Prekrivanje okolic 1D CA velikosti $M - 1 = 2$.

Indeksa levega prekrivanja $o_{\leftarrow}(x)$ ter desnega prekrivanja $o_{\rightarrow}(x)$ in nabor možnih prekrivanj so:

$$o_{\leftarrow}(x) = \sum_{i=0}^{i=M-2} |S|^i \cdot c(x+i) \tag{2.6}$$

$$o_{\rightarrow}(x) = \sum_{i=1}^{i=M-1} |S|^i \cdot c(x+i) \tag{2.7}$$

$$o \in \{0, 1, \ldots, |S|^M - 2\} \tag{2.8}$$

2.1.3 Lokalna tranzicijska funkcija in pravilo

Preslikava sedanje okolice $n^t(x)$ v prihodnjo istoležno celico $c^{t+1}(x)$ je definirana s tranzicijsko funkcijo f, ki vsaki vrednosti okolice pripiše vrednost celice:

$$c^{t+1}(x) = f(n^t(x)) \tag{2.9}$$

Za potrebe iskanja predslik je zanimiva obratna funkcija f^{-1}, ki ob podanem stanju trenutne celice $c^t(x)$ vrne množico okolic $n^{t-1}(x)$, ki se preslikajo v to vrednost:

$$f^{-1}(c^t(x)) = \{n^{t-1}(x) \in S^m \mid f(n^{t-1}(x)) = c^t(x)\} \tag{2.10}$$

Tranzicijsko funkcijo je možno definirati s pravilom r. Pravilo je indeks izbrane funkcije znotraj celotnega nabora $|S|^{|S|^M}$ funkcij. Definirano je kakor celo število v S-iškem številskem sestavu, kjer so cifre zaporedje vrednosti celic v katere tranzicijska funkcija preslika vsako od $|S|^M$ možnih okolic:

$$r = \sum_{n=0}^{n=|S|^M-1} |S|^n \cdot f(n) \tag{2.11}$$

$$r \in \{0, 1, \ldots |S|^{|S|^M} - 1\} \tag{2.12}$$

Globalna tranzicijska funkcija se pogosto zapisuje kakor tranzicijska tabela, ki navaja vrednost tranzicijske funkcije $f(n)$ za vsako okolico n.

2.1.4 Robni pogoji

Neskončen niz celic nima roba in zanj ne potrebujemo definicije robnih pogojev. Hkrati pa v praksi ne moremo računati z neskončnostjo, zato obravnavamo končne nize celic.

Če smatramo opazovani končen niz za del neskončnega niza, potem se moramo opredeliti, kako bomo obravnavali vrednosti celic izven opazovanega niza. Tukaj je opisan samo odprt robni pogoj, kjer celice izven roba niso definirane, oziroma lahko zasedejo poljubno vrednost. Za niz dolžine N je potemtakem definiranih le $N - (M - 1)$ okolic.

Drugi običajen robni pogoj je cikličen, kjer je niz celic sklenjena zanka. Ciklično koordinato celice $x_{\circlearrowleft}$ izračunamo iz x po modulu N:

$$x_{\circlearrowleft} = x \bmod N \tag{2.13}$$

2.1.5 Globalna tranzicijska funkcija

Globalna tranzicijska funkcija F s pomočjo lokalne tranzicijske funkcije $f(n)$ preslika vsako okolico $n^t(x)$ iz niza celic α^t v času t v celico $c^{t+1}(x)$ iz niza celic β^{t+1} v času $t+1$:

$$\beta^{t+1} = F(\alpha^t) \tag{2.14}$$

Za odprti robni pogoj vrne globalna funkcija za vhodni niz dolžine N izhodni niz dolžine $N - (M - 1)$, za cikličen robni pogoj pa niz dolžine N.

2.1.6 Predslike

Vsaka predslika β^{t-1} dolžine $N + M - 1$ danega niza α^t dolžine N mora ustrezati dvema pogojema:

1. Za vsako celico $c^t(x)$ mora okolica $n^{t-1}(x)$ ustrezati inverzni tranzicijski funkciji f^{-1}:

$$\forall x : c^t(x) = f(n^{t-1}(x)) \tag{2.15}$$

2. Vsak par okolic $\{n^{t-1}(x), n^{t-1}(x+1)\}$ se mora ujemati v prekrivanju:

$$\forall x : o^{t-1}(x) = o^{t-1}_{\rightarrow}(x) = o^{t-1}_{\leftarrow}(x+1) \tag{2.16}$$

Opazovanje prekrivanja okolic je glavni element algoritmov za iskanje predslik.

2.2 Definicija 2D celičnih avtomatov

Za 2D CA se celice združujejo v regularno 2D polje. Mreža polja je lahko pravokotna, šestkotna ali celo kvazikristalna. Tukaj se bomo omejili na pravokotno mrežo. Na splošno je velikost polja lahko neskončna, bolj običajna pa so končna polja, definirana kakor pravokotnik velikosti $N_x \times N_y$. Skupno število celic v končnem polju je $N = N_x \cdot N_y$.

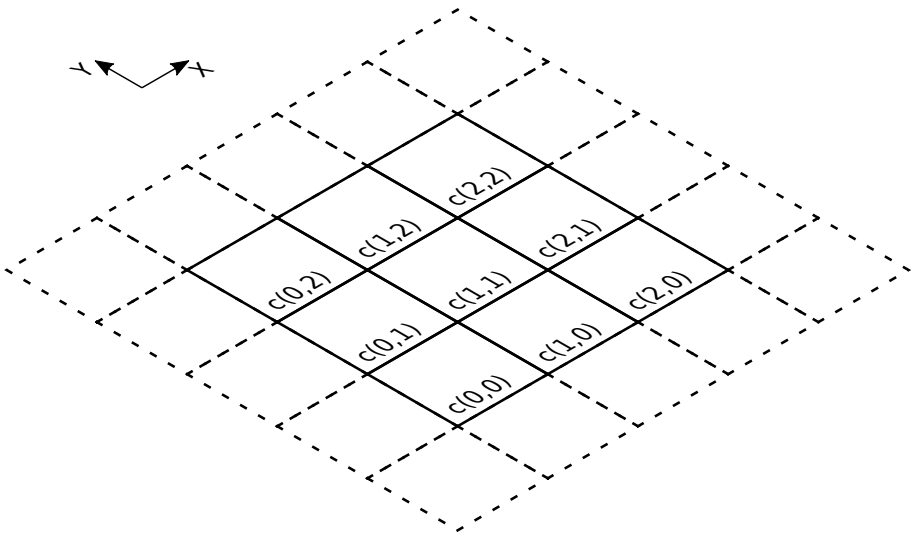
Za potrebe indeksiranja elementov CA so ploskve razdeljene v vrstice, nato pa so vrstice sestavljene v niz, ki je interpretiran kakor število. Cifre v številu si sledijo od spodaj levo do zgoraj desno znotraj okolice (sliki 2.3 in 2.4).

2.2.1 Okolica

Prihodnje stanje neke celice $c_(x, y)$ s koordinatami (x, y) je odvisno od trenutnega stanja pripadajoče okolice $n_(x, y)$ (slika 2.5). Tudi pri obliki okolice se bomo omejili na pravokotnik velikosti $M_x \times M_y$. Število celic v okolici je $M = M_x \cdot M_y$. Indeks okolice (sliki 2.3 in 2.4) in nabor možnih okolic sta:

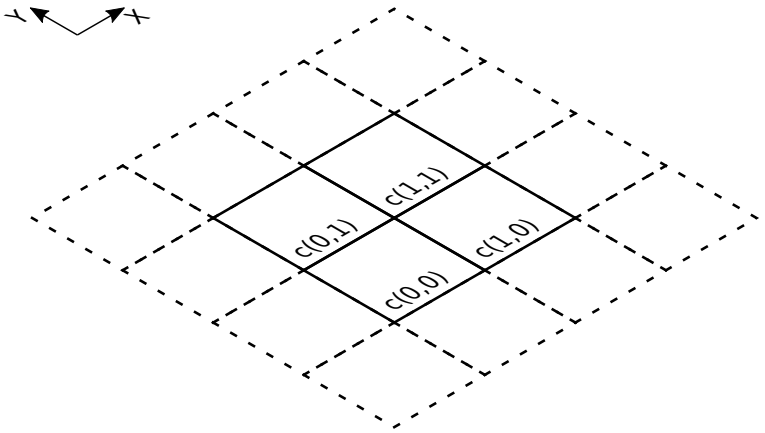
$$n(x, y) = \sum_{\substack{i=0 \\ j=0}}^{\substack{i=M_x-1 \\ j=M_y-1}} |S|^{M_x j+i} \cdot c(x+i, y+j) \tag{2.17}$$

$$n \in \{0, 1, \ldots, |S|^{M_x M_y} - 1\} \tag{2.18}$$

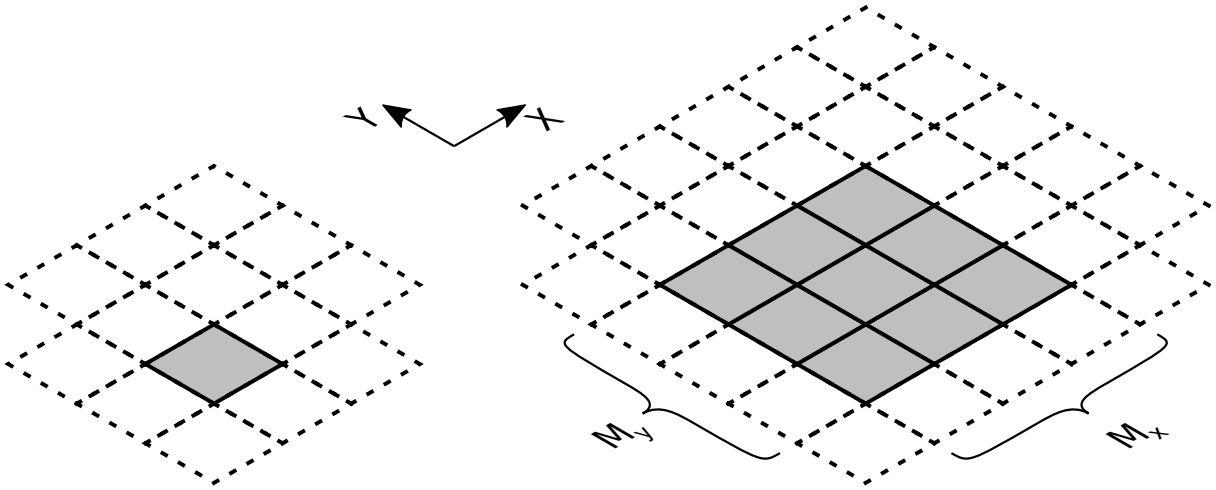


Slika 2.3: Indeksiranje okolice celice z dimenzijama $M_x = M_y = 3$.

Običajno se smatra, da je celica poravnana s sredino svoje okolice. Tukaj pa je celica poravnana v levi spodnji kot svoje okolice. To omogoča uporabo enake definicije tudi za sode velikosti okolic in odpravlja potrebo po uporabi negativnih števil v algoritmu.



Slika 2.4: Indeksiranje okolice celice z dimenzijama $M_x = M_y = 2$.



Slika 2.5: Celica $c(x, y)$ in pripadajoča okolica $n(x, y)$ z dimenzijama $M_x = M_y = 3$.

2.2.2 Prekrivanje okolic

Okolice se v smeri dimenzije X prekrivajo za ploskev velikosti $(M_x - 1) \times M_y$ (sliki 2.6 in 2.7). To ob indeksiranju da nabor:

$$o_{\leftarrow}(x, y) = \sum_{\substack{i=0 \\ j=0}}^{\substack{i=M_x-2 \\ j=M_y-1}} |S|^{(M_x-1)j+i} \cdot c(x + i, y + j) \tag{2.19}$$

$$o_{\rightarrow}(x, y) = \sum_{\substack{i=1 \\ j=0}}^{\substack{i=M_x-1 \\ j=M_y-1}} |S|^{(M_x-1)j+i} \cdot c(x + i, y + j) \tag{2.20}$$

$$o_{\leftrightarrow} \in \{0, 1, \ldots, |S|^{(M_x-1)M_y} - 1\} \tag{2.21}$$

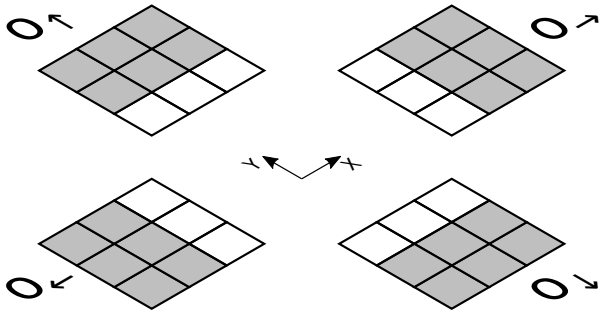
Okolice se v smeri dimenzije Y prekrivajo za ploskev velikosti $M_x \times (M_y - 1)$ (sliki 2.6 in

2.7). To ob indeksiranju da nabor:

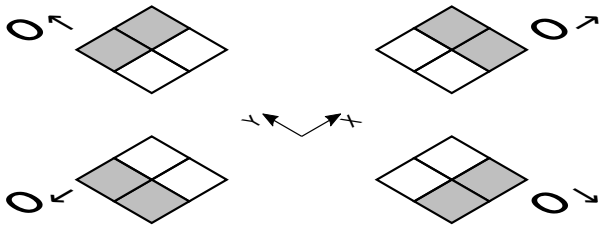
$$o_{\downarrow}(x,y) = \sum_{\substack{i=0 \\ j=0}}^{\substack{i=M_x-1 \\ j=M_y-2}} |S|^{M_x j+i} \cdot c(x+i, y+j) \tag{2.22}$$

$$o_{\uparrow}(x,y) = \sum_{\substack{i=0 \\ j=1}}^{\substack{i=M_x-1 \\ j=M_y-1}} |S|^{M_x j+i} \cdot c(x+i, y+j) \tag{2.23}$$

$$o_{\updownarrow} \in \{0, 1, \ldots, |S|^{M_x(M_y-1)} - 1\} \tag{2.24}$$



Slika 2.6: Prekrivanje okolic sosednjih celic v smeri dimenzij X in Y, za velikost okolice $M_x = M_y = 3$. Okolice se prekrivajo v 6 celicah od 9.



Slika 2.7: Prekrivanje okolic sosednjih celic v smeri dimenzij X in Y, za velikost okolice $M_x = M_y = 2$. Okolice se prekrivajo v 2 celicah od 4.

Okolice se v diagonalni smeri prekrivajo za ploskev velikosti $(M_x - 1) \times (M_y - 1)$ (sliki 2.8 in 2.9). To ob indeksiranju da nabor:

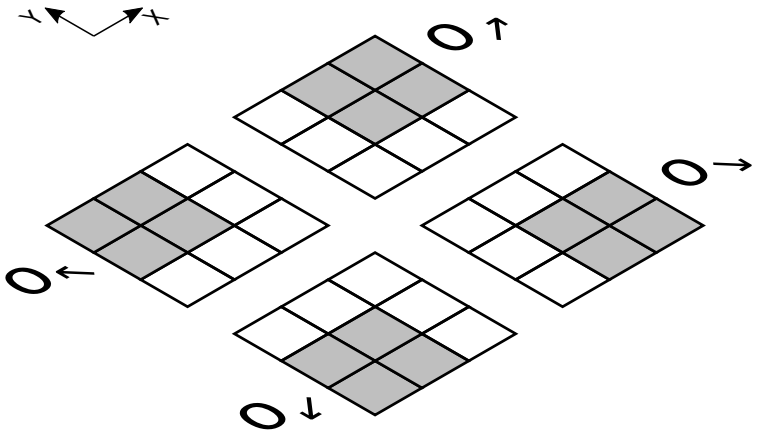
$$o_{\swarrow} = \sum_{\substack{i=0 \\ j=0}}^{\substack{i=M_x-2 \\ j=M_y-2}} |S|^{(M_x-1)j+i} \cdot c(x+i, y+j) \tag{2.25}$$

$$o_{\searrow} = \sum_{\substack{i=1 \\ j=0}}^{\substack{i=M_x-1 \\ j=M_y-2}} |S|^{(M_x-1)j+i} \cdot c(x+i, y+j) \tag{2.26}$$

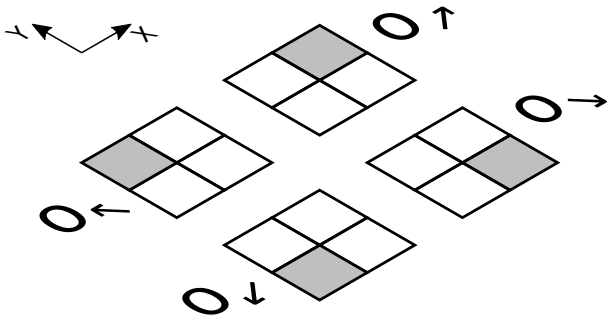
$$o_{\nwarrow} = \sum_{\substack{i=0 \\ j=1}}^{\substack{i=M_x-2 \\ j=M_y-1}} |S|^{(M_x-1)j+i} \cdot c(x+i, y+j) \tag{2.27}$$

$$o_{\nearrow} = \sum_{\substack{i=1 \\ j=1}}^{\substack{i=M_x-1 \\ j=M_y-1}} |S|^{(M_x-1)j+i} \cdot c(x+i, y+j) \tag{2.28}$$

$$o_{\times} \in \{0, 1, \ldots, |S|^{(M_x-1)(M_y-1)} - 1\} \tag{2.29}$$



Slika 2.8: Prekrivanje okolic sosednjih celic v diagonalni smeri, za velikost okolice $M_x = M_y = 3$. Okolice se prekrivajo v 4 celicah od 9.



Slika 2.9: Prekrivanje okolic sosednjih celic v diagonalni smeri, za velikost okolice $M_x = M_y = 2$. Okolice se prekrivajo v eni celici od 4.

2.2.3 Lokalna tranzicijska funkcija in pravilo

Preslikava sedanje okolice $n^t(x, y)$ v prihodnjo istoležno celico $c^{t+1}(x, y)$ je definirana s tranzicijsko funkcijo f, ki vsaki vrednosti okolice pripiše vrednost celice:

$$c^{t+1}(x, y) = f(n^t(x, y)) \tag{2.30}$$

Za potrebe iskanja predslik je zanimiva obratna funkcija f^{-1}, ki ob podanem stanju trenutne celice $c^t(x,y)$ vrne množico okolic $n^{t-1}(x,y)$, ki se preslikajo v to vrednost:

$$f^{-1}(c^t(x,y)) = \{n^{t-1}(x,y) \in S^m \mid f(n^{t-1}(x,y)) = c^t(x,y)\} \tag{2.31}$$

Tranzicijsko funkcijo je možno definirati s pravilom r. Pravilo je indeks izbrane funkcije znotraj celotnega nabora $|S|^{|S|^{M_x M_y}}$ funkcij. Definirano je kakor celo število v S-iškem številskem sestavu, kjer so cifre zaporedje vrednosti celic, v katere tranzicijska funkcija preslika vsako od $|S|^{M_x M_y}$ možnih okolic:

$$r = \sum_{n=0}^{n=|S|^{M_x M_y}-1} |S|^n \cdot f(n) \tag{2.32}$$

$$r \in \{0, 1, \ldots |S|^{|S|^{M_x M_y}} - 1\} \tag{2.33}$$

2.2.4 Robni pogoji

Neskončno polje nima roba in zanj ne potrebujemo definicije robnih pogojev. Hkrati pa v praksi ne moremo računati z neskončnostjo, zato obravnavamo končna polja celic.

Če smatramo opazovano končno polje za del neskončnega polja, potem se moramo opredeliti, kako bomo obravnavali vrednosti celic izven opazovanega polja. Tukaj je opisan samo odprt robni pogoj, kjer celice izven roba niso definirane, oziroma lahko zasedejo poljubno vrednost. Za polje velikosti $N_x \times N_y$ je potemtakem definiranih le $(N_x + (M_x - 1)) \times (N_y + (M_y - 1))$ okolic.

Drugi običajen robni pogoj je cikličen, kjer je polje celic torus. Ciklično koordinato celice $(x_\circlearrowleft, x_\circlearrowleft)$ izračunamo iz (x,y) po modulu N_x oziroma N_y.

$$x_\circlearrowleft = x \bmod N_x \qquad y_\circlearrowleft = y \bmod N_y \tag{2.34}$$

2.2.5 Globalna tranzicijska funkcija

Globalna tranzicijska funkcija F s pomočjo lokalne tranzicijske funkcije $f(n)$ preslika vsako okolico $n^t(x,y)$ iz niza celic A^t v času t v celico $c^{t+1}(x)$ iz niza celic B^{t+1} v času $t+1$.

$$B^{t+1} = F(A^t) \tag{2.35}$$

Za odprti robni pogoj vrne globalna funkcija za vhodno polje velikosti $N_x \times N_y$ izhodno polje velikosti $(N_x + (M_x - 1)) \times (N_y + (M_y - 1))$, za ciklični robni pogoj pa polje velikosti $N_x \times N_y$.

2.2.6 Predslike

Vsaka predslika B^{t-1} velikosti $(N_x+1)\times(N_y+1)$ danega polja A^t velikosti $N_x \times N_y$ mora ustrezati naslednjim pogojem:

1. Za vsako celico $c^t(x,y)$ mora okolica $n^{t-1}(x,y)$ ustrezati invezni funkciji f^{-1}:

$$\forall x,y : c^t(x,y) = f(n^{t-1}(x,y)) \tag{2.36}$$

2. Vsak par okolic v dimenziji X $\{n^{t-1}(x,y), n^{t-1}(x+1,y)\}$ se mora ujemati v prekrivanju:

$$\forall x,y : o^{t-1}_{\leftrightarrow}(x,y) = o^{t-1}_{\rightarrow}(x,y) = o^{t-1}_{\leftarrow}(x+1,y) \tag{2.37}$$

3. Vsak par okolic v dimenziji Y $\{n^{t-1}(x,y+0), n^{t-1}(x,y+1)\}$ se mora ujemati v prekrivanju:

$$\begin{aligned}\forall x,y : o^{t-1}_{\updownarrow}(x,y) &= o^{t-1}_{\downarrow}(x,y+1)\\ &= o^{t-1}_{\uparrow}(x,y+0)\end{aligned} \tag{2.38}$$

4. Vsaka četvorka diagonalnih okolic $\{n^{t-1}(x,y), n^{t-1}(x+1,y), n^{t-1}(x,y+1), n^{t-1}(x+1,y+1)\}$ se mora ujemati v prekrivanju:

$$\begin{aligned}\forall x,y : o^{t-1}_{\times}(x,y) &= o^{t-1}_{\searrow}(x+0,y+1) = o^{t-1}_{\swarrow}(x+1,y+1)\\ &= o^{t-1}_{\nearrow}(x+0,y+0) = o^{t-1}_{\nwarrow}(x+1,y+0)\end{aligned} \tag{2.39}$$

Za potrebe nekaterih ilustracij je uporabljena bolj splošna definicija inverzne lokalne tranzicijske funkcije. Za vsako celico je lahko definiran lasten nabor predslik $n^{t-1} \in S^M$, ki je na splošno neodvisen od stanja celice. Ta posplošitev je uporabljena za konstrukcijo umetnih mrež predslik, ki poudarjajo konkretne probleme, povezane s kompleksnostjo algoritma.

2.2.7 Prekrivanje predslik in prerez

Če polje 2D celičnega avtomata razdelimo na dva dela, lahko definiramo predsliko za vsak del posebej. Tukaj se bomo omejili na ravne prereze polja, čeprav je na splošno mogoče definirati poljuben prerez.

Indeks prereza v smeri dimenzije X na koordinati y in njegov nabor sta definirana kot:

$$e_{\leftrightarrow}(y) = \sum_{\substack{i=0\\j=0}}^{\substack{i=N_x+M_x-1\\j=M_y-1}} |S|^{(N_x-1)j+i} \cdot c(i,y+j) \tag{2.40}$$

$$e_{\leftrightarrow} \in \{0,1,\ldots,|S|^{(N_x+M_x-1)(M_y-1)}-1\} \tag{2.41}$$

Indeks prereza v smeri dimenzije Y na koordinati x in njegov nabor sta definiran kakor:

$$e_{\updownarrow}(x) = \sum_{\substack{i=0 \\ j=0}}^{\substack{i=M_x-1 \\ j=N_y+M_y-1}} |S|^{(N_x-1)j+i} \cdot c(x+i,j) \tag{2.42}$$

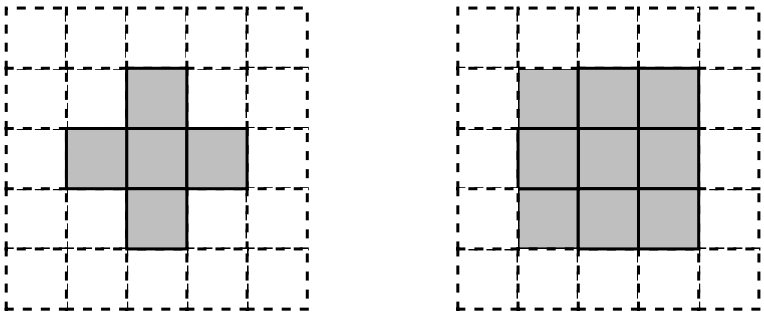
$$e_{\updownarrow} \in \{0, 1, \ldots, |S|^{(M_x-1)(N_y+M_y-1)} - 1\} \tag{2.43}$$

Prerezi se uporabljajo za zapis vmesnih rezultatov pri algoritmih za iskanje predslik 2D CA.

2.2.8 Primeri

Klasične 2D okolice

Za klasične 2D CA sta definirani von Neumannova in Mooreova okolica (slika 2.10). Von Neumanova okolica je podmnožica Mooreove okolice, kjer so stanja vogalnih okolic ignorirana. To pomeni, da tukaj opisana splošna definicija zaobjame obe klasični okolici.



Slika 2.10: Klasični 2D okolici, von Neumannova (levo) in Mooreova (desno).

Okolica quad

V primerih je uporabljen binarni ($|S| = 2$) 2D CA z okolico quad $M_x = M_y = 2$ (slika 2.4). V okolici n je $M = M_x \cdot M_y = 2 \cdot 2 = 4$ celic, kar da nabor velikosti $|S|^4 = 16$. V prekrivanjih $o_{\leftrightarrow}$ in $o_{\updownarrow}$ v smereh X in Y sta $M = (M_x - 1) \cdot M_y = M_x \cdot (M_y - 1) = 2$ celici, kar da nabor velikosti $|S|^2 = 4$. V prekrivanjih $o_{\times}$ v diagonalnih smereh je $M = (M_x - 1) \cdot (M_y - 1) = 1 \cdot 1 = 1$ celic, kar da nabor velikosti $|S|^1 = 2$.

Conwayev GoL

GoL je binaren celični avtomat ($|S| = 2$), ki uporablja Mooreovo okolico $M_x = M_y = 3$ (slika 2.3). V okolici n je $M = M_x \cdot M_y = 3 \cdot 3 = 9$ celic, kar da nabor velikosti $|S|^9 = 512$. V prekrivanjih $o_{\leftrightarrow}$ in $o_{\updownarrow}$ v smereh X in Y je $M = (M_x - 1) \cdot M_y = M_x \cdot (M_y - 1) = 6$ celic, kar da nabor velikosti $|S|^9 = 64$. V prekrivanjih $o_{\times}$ v diagonalnih smereh je $M = (M_x - 1) \cdot (M_y - 1) = 2 \cdot 2 = 4$ celic, kar da nabor velikosti $|S|^4 = 16$.

Nabor vseh pravil n za to okolico je velikosti $|S|^{|S|^M} = 2^{512}$. Lokalna tranzicijska funkcija je definirana opisno [7]. Stanji celice sta opisani kakor mrtvo ($c = 0$) in živo ($c = 1$). V vsakem časovnem koraku se z vsako celico zgodi naslednje:

1. živa celica z manj kakor dvema živima sosedoma umre,
2. živa celica z dvema ali tremi živimi sosedi živi naprej v naslednji korak,
3. živa celica z več kakor tremi živimi sosedi umre,
4. mrtva celica z natanko tremi živimi sosedi oživi.

Pravilo GoL je 512-bitna binarna vrednost, v šestnajstiškem zapisu je r:

```
00000000000100010001000101170116000100010117011601170116177E1668
0001000101170116011701161776801170116177E1668177E16687EE86880
```

Poglavje 3

Konstrukcija mreže predslik

Mreža predslik je grafični konstrukt, ki omogoča upodobitev okolice, prekrivanja okolic in tranzicijske funkcije kot ločene grafične elemente (vozlišča, poti in ploskve). Odnosi med temi elementi definirajo pravila, na katerih se gradijo algoritmi za iskanje predslik.

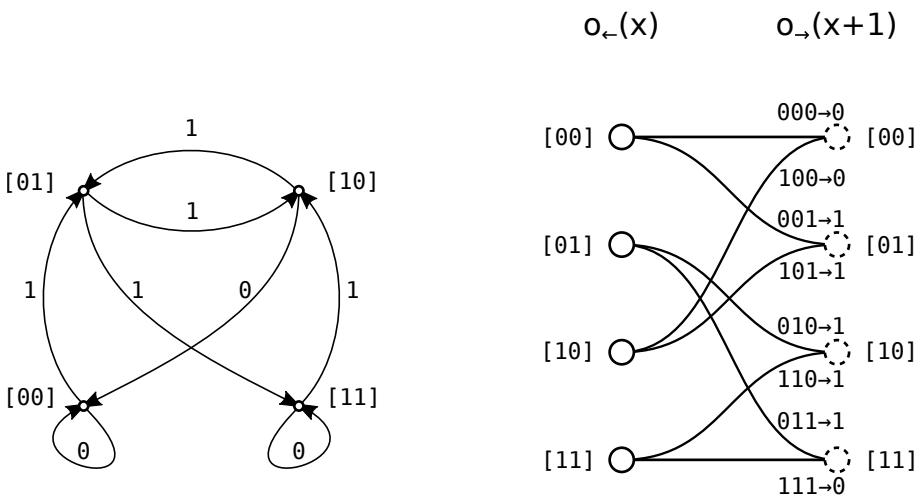
3.1 De Bruijnov diagram

Osnovni element grafične upodobitve je nekoliko svobodna interpretacija De Bruijnovega diagrama [8]. V osnovi ta obravnava ciklične premike končnih zaporedij simbolov ter njihovo prekrivanje. Harold V. McIntosh [33] in njegova skupina [40] so priredili De Bruijnove grafe za potrebe analize 1D CA (slika 3.1 levo). Vozlišča predstavljajo nabor možnih prekrivanj med okolicami, poti med njimi pa okolice, ki se v vrednosti celic ujemajo s temi prekrivanji. Tranzicijska funkcija pripiše vsaki okolici stanje celice; posledično so poti med vozlišči označene z vrednostmi iz tranzicijske tabele. De Bruijnove grafe nato razdelimo na podgrafe za vsako stanje celice. Podgraf vsebujejo le tiste poti, katerih pripadajoče okolice se preslikajo v eno od stanj celice. De Bruijnov podgraf predstavlja nabor vseh preteklosti celice z danim stanjem.

3.2 Mreža predslik 1D celičnih avtomatov

Razvil sem modificirano upodobitve De Bruijnovega grafa, kjer so originalna vozlišča (prekrivanja o) podvojena in gredo poti (okolice n) vedno od originala $o_{\leftarrow}$ proti dvojniku $o_{\rightarrow}$ (slika 3.1 desno) [30]. Tako upodobitev sem poimenoval **graf predslik**.

Graf predslik omogoča **nizanje** grafov v **mrežo predslik**. Dvojniki vozlišč za trenutno celico $o_{\rightarrow}(x)$ se ujemajo z originalnimi vozlišči za naslednjo celico $o_{\leftarrow}(x+1)$ (slika 1.4). Medtem ko osnovni De Bruijnov graf opisuje okolico ene same celice, nizan graf opisuje okolice niza celic. Nizan graf sem poimenoval mreža predslik, saj predstavlja nabor vseh



Slika 3.1: De Bruijnov graf za elementarni 1D CA, pravilo 110. Na levi je McIntosheva upodobitev, na desni pa moja, ki omogoča nizanje. S prekinjeno črto so predstavljene kopije vozlišč.

predslik celotne konfiguracije. Število predslik je enako številu vseh poti, ki povezujejo en in drugi rob.

Robni pogoji 1D CA so definirani na levem in desnem koncu, ki omejujeta končno število celic znotraj neskončnega niza (premica). Robni pogoj definira, katera prekrivanja $o_{\leftrightarrow}$ in s kakšnimi utežmi so dovoljena ob robu. Lahko si jih predstavljamo tudi kakor vpliv pol neskončnega niza celic (poltraka), ki sega izven roba opazovane konfiguracije. Podrobneje so robni pogoji 1D CA definirani v [30].

3.3 Mreža predslik 2D celičnih avtomatov

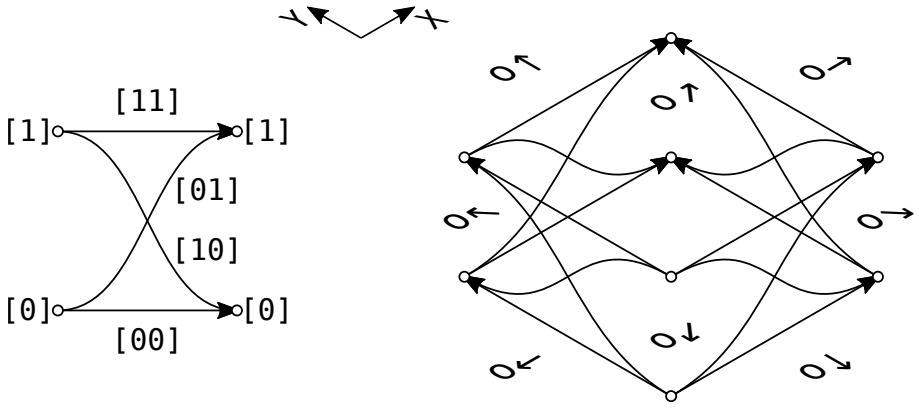
3.3.1 Graf predslik

Za potrebe opisa 2D CA je bila elementom grafa predslik dodana nova dimenzija. Povezave med vozlišči se spremenijo v ploskve, vozlišča pa se spremenijo v robove ploskev. Elementi celičnega avtomata, ki se preslikajo v graf, so:

- nabor **okolic** (n) postane nabor **ploskev** (slika 3.3),
- nabor **prekrivanj v smeri dimenzij X in Y** ($o_{\leftrightarrow}$ in $o_{\updownarrow}$) (slika 2.7) postane nabor **povezav**,
- nabor **prekrivanj v diagonalni smeri** ($o_{\times}$) (slika 2.9) postane nabor **vozlišč**.

Graf predslik je postavljen na kvadrat, ki predstavlja okolico n. Nad vsakim vogalom je nabor vozlišč, ki predstavljajo nabor diagonalnih prekrivanj $o_{\swarrow}$, $o_{\searrow}$, $o_{\nwarrow}$ in $o_{\nearrow}$.

Vozlišči sosednjih vogalov v dimenziji X ($o_{\swarrow}$ in $o_{\searrow}$) sta povezani, če sta del iste konfiguracije celic, ki predstavlja povezavo v smeri X ($o_{\downarrow}$). Vozlišči sosednjih vogalov v dimenziji



Slika 3.2: Mreža ene celice za binarni CA z okolico quad $M_x = M_y = 2$.

Y ($o_{\swarrow}$ in $o_{\nwarrow}$) sta povezani, če sta del iste konfiguracije celic, ki predstavlja povezavo v smeri Y ($o_{\leftarrow}$).

Nastali graf (slika 3.2 desno) ima poleg vozlišč in povezav med njimi tudi ploskve. Vogali $o_{\swarrow}$, $o_{\searrow}$, $o_{\nwarrow}$ in $o_{\nearrow}$ pripadajo določeni ploskvi n, če so del iste konfiguracije celic. Podobno velja za robove; $o_{\leftarrow}$, $o_{\rightarrow}$, $o_{\downarrow}$ in $o_{\uparrow}$ pripadajo določeni ploskvi n, če so del iste konfiguracije celic.

Za primer je uporabljena binarna okolica quad. Ploskev $n = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ ima vogale $o_{\swarrow} = 1$, $o_{\searrow} = 0$, $o_{\nwarrow} = 0$ in $o_{\nearrow} = 1$ ter robove $o_{\leftarrow} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$, $o_{\rightarrow} = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$, $o_{\downarrow} = \left[\begin{smallmatrix} 0 & 1 \end{smallmatrix}\right]$ in $o_{\uparrow} = \left[\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right]$.

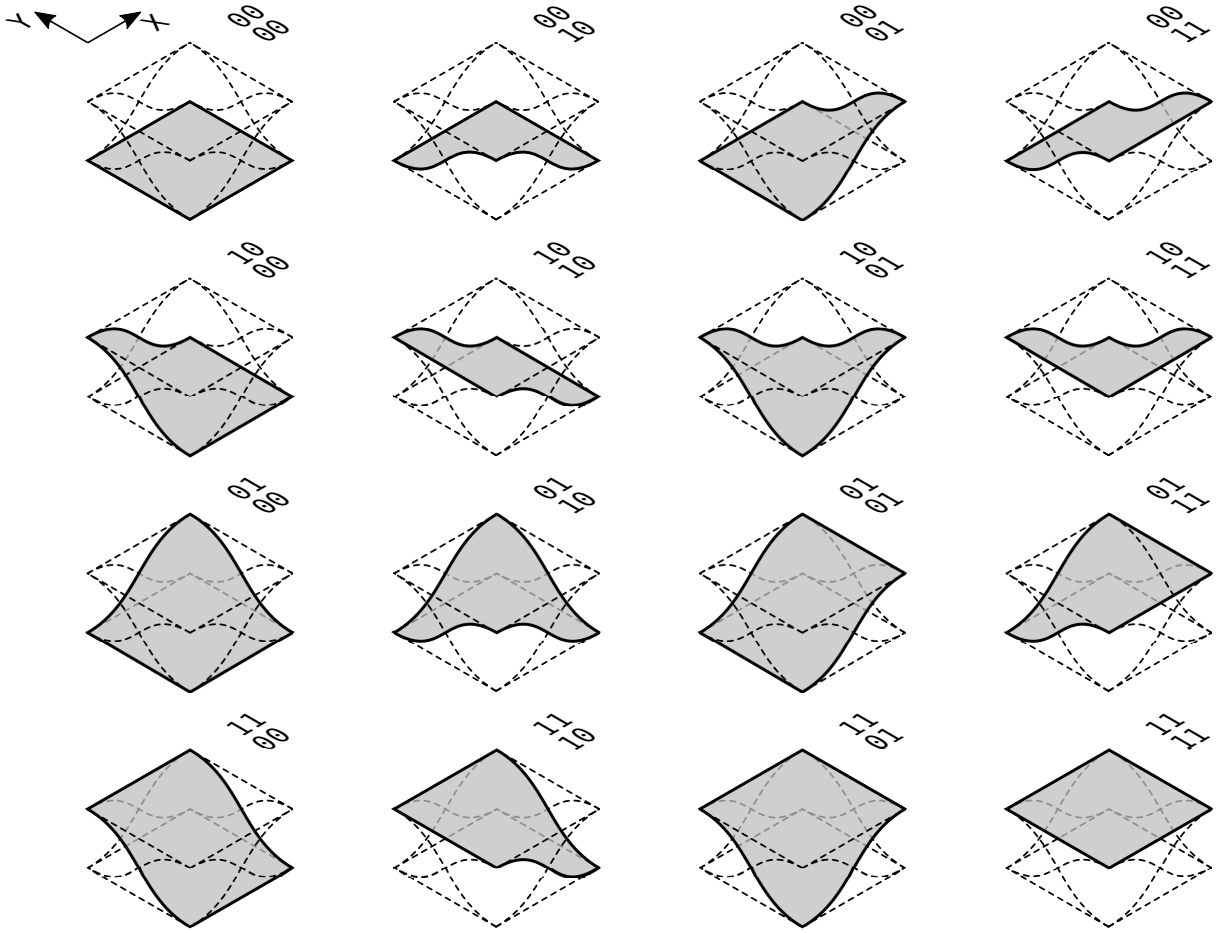
3.3.2 Mreža predslik

Podobno kot je pri 1D CA možno grafe predslik nizati v mrežo predslik, je pri 2D CA grafe predslik možno **tlakovati** v **mrežo predslik**, ki opisuje polje več celic. Nabor predslik celotnega polja je ekvivalenten naboru vseh *zveznih* ploskev, ki prekrivajo celotno polje (zahteva po prekrivanju okolic) in ki jih je možno sestaviti iz naborov ploskev diagramov posameznih celic (zahteva po ujemanju z lokalno tranzicijsko funkcijo).

Preslikava iz zvezne ploskve v mreži predslik v konfiguracijo polja celic predslike B^{t-1} je enolična. Iz ploskve vsake okolice $n(x, y)$ vzamemo izhodiščno celico $c_{\swarrow}(x, y)$ (celica spodaj levo znotraj okolice). Nato dodamo vrstico celic višine $M_y - 1$ iz zgornjega prekrivanja na vrhu $o_{\uparrow}(x, y = N_y)$, ter stolpec celic širine $M_x - 1$ iz desnega prekrivanja na skrajni desni $o_{\rightarrow}(x = N_x, y)$.

Podani primeri uporabljajo binarni CA z okolico quad. Za ta CA je velikost diagonalnega prekrivanja okolic ena sama celica (slika 2.9); posledično ima nabor vozlišč le dve vrednosti, ki neposredno predstavljata vrednosti celic v predsliki (slika 3.4).

Pravilnost razširitve diagrama ene celice v mrežo lahko dokažemo z indukcijo. Dokazati želimo, da je neka konfiguracija celic predslika dane sedanjosti, *če in samo če* je ta ekvivalentna zvezni ploskvi v mreži predslik.

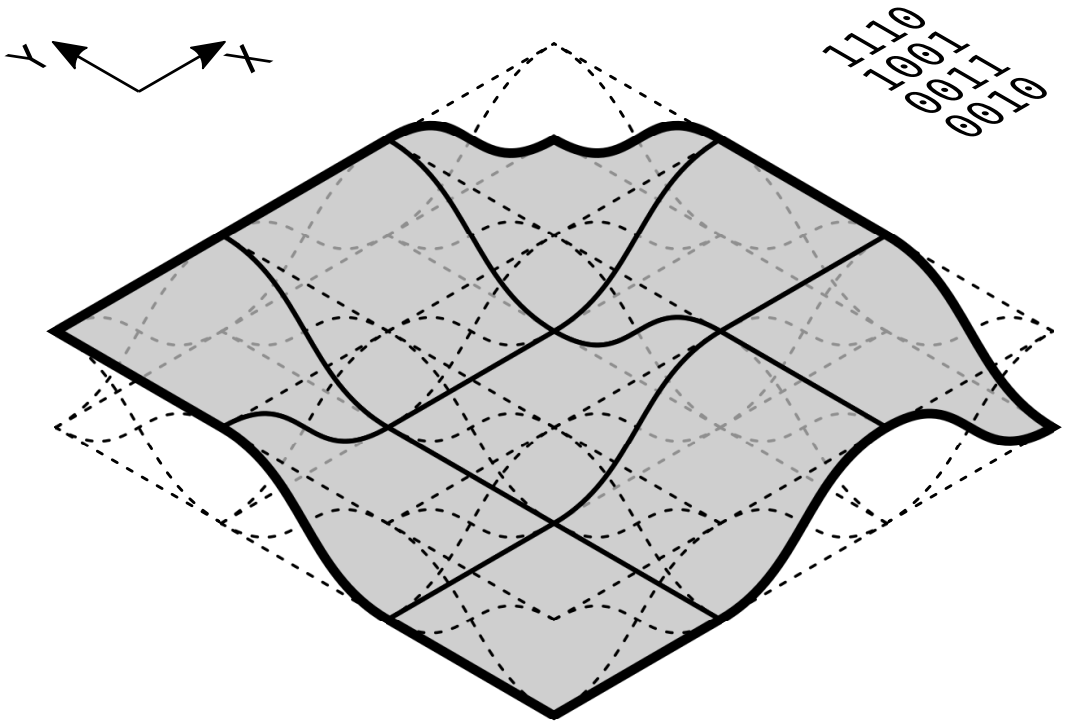


Slika 3.3: Nabor ploskev za vseh 16 možnih vrednosti okolic za binarni CA z okolico quad $M_x = M_y = 2$.

Prvi element: Iz definicije *okolice/ploskve* velja, da je za eno samo celico v mreži nabor ploskev enak naboru vseh predslik. **Naslednji element:** Obstoječi mreži predslik dodamo novo celico. Ploskev iz nabora dodane celice se zvezno veže s ploskvijo iz obstoječega nabora zveznih ploskev, ko se z njo ujema v robu (vozliščema in povezavi med njima). Robovi ploskev se ujemajo natanko v primeru, ko se ujemajo prekrivanja okolic. Temu je tako, ker so indeksi vozlišč in povezav ekvivalentni vrednosti prekrivanj okolic v diagonali in v smeri koordinatnih osi.

3.3.3 Robni pogoj

Pri 2D CA je robni pogoj definiran kakor utež sklenjene poti okoli ploskve opazovane konfiguracije celic. V mreži predslik za 2D CA povezave med vozlišči definirajo rob ploskve (slika 3.4). Vsaka zvezna ploskev v mreži ima zvezen rob. Robni pogoj določa, kako je ta ploskev obravnavana, oziroma, ali je sploh obravnavana. Ker uporabna vrednost splošnega robnega pogoja še ni znana, in bi splošnost izrazito povečala zahtevnost algoritma za iskanje predslik, so tukaj vsi robovi obravnavani enako, z utežjo ena. Temu bomo rekli odprti rob, ker ta ne definira nobenih omejitev, katere zvezne ploskve v mreži predslik so dovoljene in katere ne.



Slika 3.4: Mreža velikosti $N_x = 3$ in $N_y = 3$ za binarni CA z okolico quad. Poudarjena je ena zvezna ploskev in njen zvezen rob. Konfiguracija pripadajoče predslike je izpisana.

Obstaja še en enostaven robni pogoj, ki je definiran za ciklično sklenjena končna polja CA (torus). Ta tip robnega pogoja tukaj ne bo obravnavan, ker še dodatno poveča kompleksnost algoritmov. Potrebno bi bilo namreč opraviti celoten postopek iskanja predslik za vsak začetni rob posebej. Saj je zvezna ploskev predslika v cikličnem polju le, če se njen začetni in končni rob ujemata. Pojma začetnega in končnega roba sta opisana v poglavju o algoritmu. V članku [30] je podrobneje opisan ta problem za 1D CA.

Poglavje 4

Algoritem za štetje in izpis predslik

Pri 1D CA [30] ima algoritem za štetje predslik linearno $O(N)$ procesno in pomnilniško zahtevnost. Posplošeno to pomeni, da se vsaka celica pojavi v izračunu samo enkrat. Vsak algoritem za štetje ima tudi logaritmično komponento, saj število bitov, potrebnih za zapis števcev, raste logaritmično s preštetim številom. Ta komponenta se pozna pri implementaciji, šele ko velikost števcev preseže velikost registrov. 64 bitov zadošča za binarno polje celic 8×8.

Za 2D CA se je izkazalo, da obstajajo problemi, ki niso rešljivi z linearno kompleksnostjo. Prikazanemu algoritmu raste maksimalna procesna zahtevnost eksponentno v odvisnosti od ene dimenzije in linearno v odvisnosti od druge dimenzije, torej $O(C^{N_x}N_y)$ ali $O(N_xC^{N_y})$ ($C = |S|^M$ je konstantna velikost nabora okolic). Za sedaj je še neznano, če obstoja algoritem z manjšo maksimalno kompleksnostjo.

Algoritem za izpis predslik ima ne glede na število dimenzij neizogibno eksponentno kompleksnost $O(C^N)$ (C je konstanta, odvisna od optimizacij), saj maksimalno in povprečno število predslik raste eksponentno v odvisnosti od števila celic.

4.1 Štetje poti v grafu

Algoritem za štetje uporablja principe iz teorije grafov [14]. Število poti med dvema vozliščema v acikličnem usmerjenem grafu se računa tako, da se vsakemu vozlišču pripiše utež w. V prvem koraku začetno vozlišče dobi utež $w(zacetek) = 1$. Vse poti, ki izhajajo iz obteženega vozlišča, imajo utež enako uteži izvornega vozlišča. V naslednjih korakih se računajo uteži vozlišč tako, da je izhodna utež vsota vhodnih uteži $w_{izhod} = \Sigma w_{vhod}$. Algoritem se zaključi, ko število opravljenih korakov doseže dolžino najdaljše poti. Na koncu je utež končnega vozlišča $w(konec)$ enaka številu vseh poti med začetnim in končnim vozliščem.

Če želimo vedeti koliko od teh poti gre skozi posamezno vozlišče ali po posamezni povezavi med vozlišči, potem postopek ponovimo v obratni smeri. Vlogi začetnega in končnega

vozlišča se zamenjata, in obrnejo se smeri vseh poti. Skupna utež vozlišča je zmnožek uteži za naprej in nazaj $w_{skupna} = w_{naprej} \cdot w_{nazaj}$.

Pri celičnih avtomatih je opazovani graf mreža predslik, ki ima zelo regularno strukturo. Dolžina najdaljše poti je enaka velikosti ene od dimenzij polja CA. Poleg tega ob vsakem koraku opazujemo samo del vozlišč, ki pripadajo enemu prerezu.

4.2 Štetje predslik

Opisani algoritem razdeli polje celic na vrstice v dimenziji X. Delitev po stolpcih v dimenziji Y bi bila ekvivalentna, tako da je to arbitrarna odločitev. S stališča procesne zahtevnosti je najbolje izbrati krajšo dimenzijo.

Ker mreža predslik ni navaden graf, temveč je razširjen s pojmom ploskve, je potrebno mrežo predslik pred začetkom štetja najprej preslikati v navaden graf. Vozlišča predstavljajo vsi možni robovi $e_{\leftrightarrow}(y)$ na vseh presekih med vrsticami. Vozlišča so fiksni elementi v grafu, obstajajo ne glede na konfiguracijo in stanje predslik.

Povezave med vozlišči so ploskve med dvema roboma $e_{\leftrightarrow}(y)$ in $e_{\leftrightarrow}(y+1)$. Vsaka povezava predstavlja eno predsliko vrstice celic na koordinati y. Povezava v grafu obstaja, če obstaja ekvivalentna predslika vrstice, torej je za izgradnjo grafa potrebno poiskati vse predslike za vsako vrstico. Ker je vrstica v 2D polju 1D objekt, lahko problem pretvorimo v 1D CA in uporabimo za izračun predslik vrstice algoritme za štetje in izpis predslik 1D CA.

Z uporabo Boolove algebre namesto operacij množenja in seštevanja je možno poenostaviti iskanje predslik samo na njihov obstoj za opazovani robni pogoj. Uteži postanejo binarne vrednosti $w \in \{0, 1\}$. Predslika ali delna predslika obstaja, če je njena utež neničelna $w > 0$. Za preverjanje tega pogoja ni potrebno štetje, zadoščajo že logične operacije.

Tukaj opisana implementacija algoritma (izvorna koda je v dodatku D) ne išče vseh predslik za vrstico hkrati, temveč se omeji le na predslike, ki so skladne z enim od možnih robov $e_{\leftrightarrow}$ in to naredi za vse robove, ki so na voljo.

4.2.1 Procesiranje niza v dimenziji X

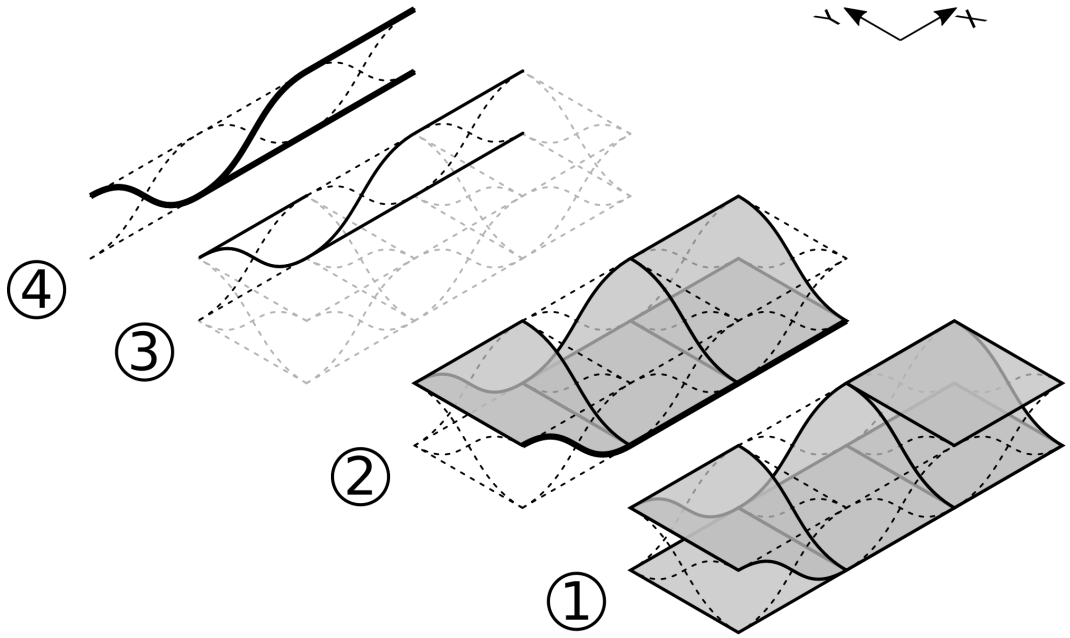
Procesiranje se začne z 1D nizom celic (vrstico). Vsaki celici v nizu pripada lastna mreža predslik, eno-dimenzionalnemu nizu celic posledično pripada povezan niz mrež predslik (slika 4.1, mreža **1**). Vsak segment mreže v nizu ima svoj nabor veljavnih okolic, ki je definiran s tranzicijsko funkcijo (ali pa je posplošeno poljuben nabor). Na začetku se ploskve okolic sosednjih celic še ne povezujejo v zvezno ploskev čez celoten niz. Izločiti je potrebno vse ploskve okolic, ki se ne povezujejo z okolicami svojih sosednjih celic.

Ker se vsaka vrstica povezuje s predhodno in naslednjo vrstico, je potrebno upoštevati tudi zveznost ploskev na prehodu med vrsticami. Ta povezava med vrsticami je prerez opazovane ploskve na meji med vrsticama. Vsako ploskev obravnavamo posebej in prerez

izrazimo kakor robni pogoj. Začetni robni pogoj $e_{\downarrow}$ za vsako vrstico je nabor poti med vozlišči na spodnji strani vrstice. Vsak začetni pogoj obravnavamo posebej (slika 4.1, mreža **2**, odebeljena pot).

Začetni robni pogoj $e_{\downarrow}$ apliciramo tako, da izločimo iz obravnave vse ploskve, ki nimajo skupnega roba z robnim pogojem (slika 4.1, prehod iz mreže **1** v mrežo **2**). Po tem koraku so vse okolice definirane na spodnjem prekrivanju. Do nezveznosti lahko prihaja le še na robu, nasprotnem začetnemu robu $e_{\uparrow}$.

Problem se reducira iz 3D mreže in procesiranja ploskev v 2D mrežo, kjer se procesirajo poti med vozlišči (slika 4.1, prehod iz mreže **2** v mrežo **3**). To je problem, ekvivalenten štetju in izpisu predslik za 1D CA, za kar sem uporabil algoritem, opisan v [30]. Rezultat je nabor končnih robnih pogojev $e_{\uparrow}$ (slika 4.1, mreža **4**, končni robni pogoj je odebeljen).



Slika 4.1: Mreža vrstice velikosti $N_x = 3$ za binarni CA z okolico quad. Nabor okolic/ploskev za posamezno celico je arbitraren. Izbran je tako, da poudari korake algoritma.

Po procesiranju nabora vseh začetnih robnih pogojev $e_{\downarrow}$ nastane nabor vseh končnih robnih pogojev $e_{\uparrow}$. Ta nabor se v naslednjem koraku uporabi kakor začetni robni pogoj za naslednjo vrstico.

Zaradi procesa izpisa poti v grafu, procesna zahtevnost raste eksponentno z dolžino vrstice $O(C^{N_x})$.

4.2.2 Procesiranje polja v dimenziji Y

V drugi dimenziji se procesira zaporedje vrstic od prve do zadnje (slika 4.2). Vsaka vrstica se začne in konča z robnim pogojem. Vsako vrstico je potrebno v drugi dimenziji

procesirati le enkrat, kar pomeni, da je računska zahtevnost linearno odvisna od števila vrstic $O(N_y)$.

Vsakemu robnemu pogoju je pripisana utež. Začetni robovi za prvo vrstico imajo pripisano utež $w = 1$. Na splošno se lahko več začetnih robov preslika v isti končni rob ene vrstice. Utež, pripisana končnemu robu dane vrstice, je vsota uteži vseh začetnih robnih pogojev, ki se preslikajo vanj. Utež tako predstavlja število vseh predslik za dano in prejšnje vrstice, ki se končajo z opazovanim robom. Robovi, s katerimi se ne konča nobena od predslik, dobijo utež $w = 0$.

Za tem, ko so procesirane vse vrstice, je znan končni rob nabora vseh zveznih ploskev na celotni mreži predslik. Vsota uteži vseh končnih robov daje število predslik celotne mreže.

4.3 Izpis predslik

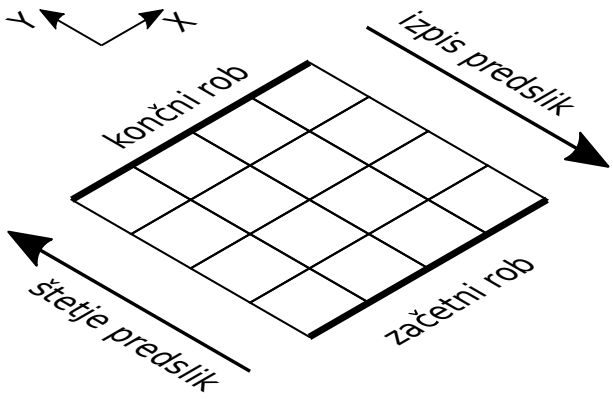
Algoritem za izpis predslik (izvorna koda je v dodatku D) je nadaljevanje algoritma za štetje. Začne z znanim številom predslik, ki ga je dalo štetje, in izpisuje predslike po vrsticah v obratni smeri, kakor je potekalo štetje (od zadnje do prve vrstice). Pridobljen seznam predslik je sortiran. Smer sortiranja je odvisna od smeri procesiranja. Smer procesiranja je možno po potrebi spremeniti.

Ni nujno, da se vsak vrstični začetni pogoj preslika v nabor končnih pogojev. Možno je, da nobena pot na končnem robu ne zadošča začetnemu pogoju in mreži predslik. Torej po prejšnjih korakih še ni točno določeno, katere ploskve se združujejo v zvezno celoto in katere ne. Možne so ploskve, ki prekrivajo polje samo do neke vrstice in ne naprej.

Procesiranje po drugi dimenziji poteka v obratni smeri kakor pri štetju; začne pri zadnji vrstici in konča pri prvi vrstici (slika 4.2). Z vsakim korakom se izloča še preostale slepe poti iz mreže predslik. Za vsako vrstico je potrebno ponovno rešiti 1D problem, le da sta začetni in končni rob zamenjana.

Hkrati je možno še izpisovati predslike. Že na začetku procesiranja v obratni smeri je znano število vseh predslik, kar omogoča rezervacijo pomnilnika in inicializacijo seznama predslik. Končne robne poti in njihove uteži se uporabijo za popis tekoče vrstice celic v predslikah. Indeks robne poti (določeno zaporedje prekrivajočih okolic) da vrednost celic v vrstici predslike. Uteži, izračunane v smeri štetja pa dajo število predslik, ki jih je potrebno popisati z dano vrednostjo za vrstico celic.

Z vsakim korakom v obratni smeri se popiše nova vrstica za vsako od preštetih predslik. Ko pride algoritem spet do začetne vrstice, so vse predslike popisane. Hkrati so iz mreže predslik izločene vse slepe poti; ostanejo le še ploskve, ki tvorijo zvezne ploskve na celotnem polju celic.



Slika 4.2: Potek smeri procesiranja pri algoritmu za štetje in izpis predslik.

4.4 Nezmožnost štetja z linearno zahtevnostjo

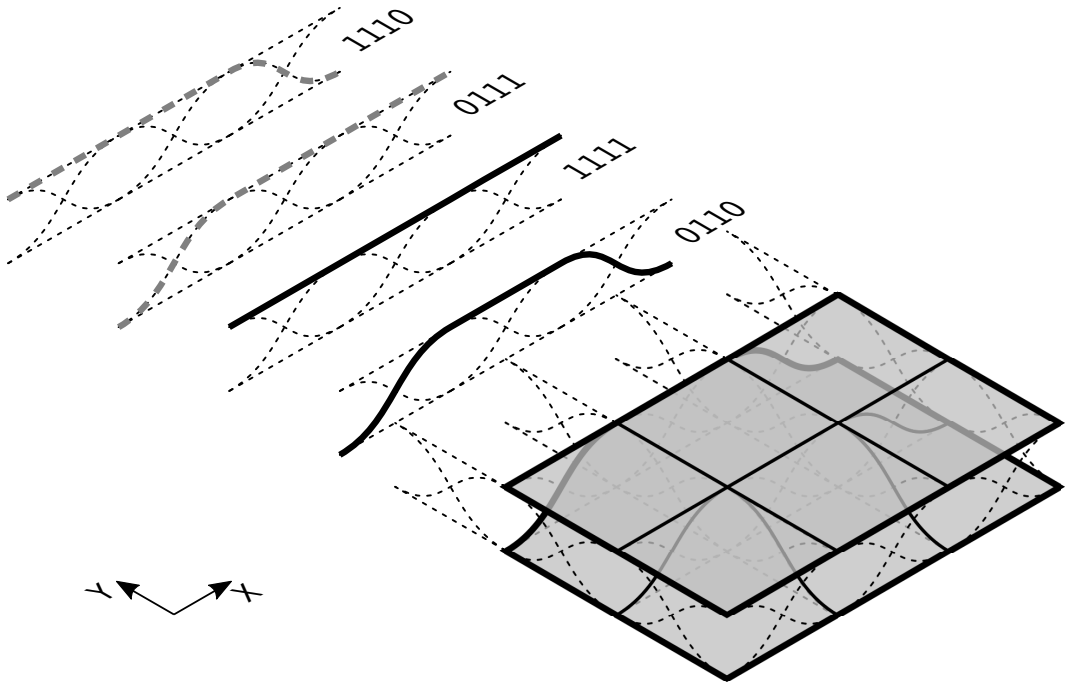
Podan je primer, ki kaže, zakaj procesiranje z linearno zahtevnostjo ni mogoče.

Cilj raziskovalnega dela je bil poiskati učinkovit algoritem za iskanje predslik 2D CA. Na podlagi izkušenj z 1D CA sem optimistično pričakoval, da bo možno problem rešiti v linearnem času.

Algoritem v linearnem času bi vsako celico obravnaval le enkrat; na splošno bi bilo število obravnav majhna konstanta (4-krat, če se izvaja procesiranje skozi mrežo predslik v 4 smereh/prehodih). Tak algoritem predpostavlja, da je možno celoten nabor začetnih robnih pogojev upoštevati hkrati v enem samem prehodu. V koraku **2** pri procesiranju vrstice (slika 4.1) bi se izločilo le ploskve, ki ne ustrezajo nobenemu od začetnih robnih pogojev. Po nekaj poizkusih sem ugotovil, da tak algoritem ne deluje pravilno. Vsaj nekatere poti iz nabora je potrebno upoštevati ločeno.

V podanem primeru (slika 4.3) se prvi dve vrstici zaključita z naborom dveh končnih poti **0110** in **1111**. Če bi ti dve poti uporabili hkrati, kakor začetni robni pogoj za naslednjo vrstico, bi dobili enak rezultat, kakor če bi bili del robnega pogoja tudi poti **0111** in **1110**. To pa zato, ker algoritem, ki poti ne obravnava ločeno, ne more vedeti, kje se je pot začela, da bi jo lahko tudi pravilno zaključil. Posledično lahko linearen algoritem vzame začetek ene od poti in ga na vozlišču, skupnem obema potema, nadaljuje po drugi poti. Iz mreže je razvidno, da ti dve poti nista preseka neke zvezne ploskve. Skratka, hkratno obravnavanje celotnega nabora robnih pogojev ni možno, kar pomeni, da je število obravnav neke celice odvisno od števila poti in posledično od eksponenta števila celic v vrstici.

Možno je, da obstajajo drugačne optimizacije algoritma, ki lahko zmanjšajo procesno kompleksnost.



Slika 4.3: Ovira za procesiranje z linearno zahtevnostjo. Veljavni robni pogoj je odebeljen, neveljavni pa črtkan.

Poglavje 5

Pregled obstoječih algoritmov in implementacij

Obstoječi algoritmi za predslike 2D CA se osredotočajo izključno na GoL. Največ algoritmov je namenjenih iskanju stanj GoE, tako da le-ti zgolj preverjajo obstoj predslik in jih ne štejejo ali izpisujejo. Naslednja skupina algoritmov sicer izpisuje predslike, ampak tega ne počne sistematično, saj je njihov cilje poiskati le eno ali nekaj predslik. Algoritmi, ki so namenjeni dejanskemu štetju in izpisu vseh predslik so redki. Zelo redki so tudi akademski prispevki, kjer bi bil kak algoritem povezan s predslikami 2D CA tudi strokovno opisan.

5.1 Iskanje stanj GoE za GoL

Conway je leta 1970 predstavil CA po imenu *Game of Life*, ki je kmalu začel nabirati navdušence. Del teh je zanimalo, če v GoL obstajajo stanja brez predslik imenovana *Garden of Eden*. Nekateri so se lotili raziskovanja in Martin Gardner je obstoj GoE stanj dokazal teoretično [10]. Spet drugi so se lotili iskanja takega stanja. Že leta 1971 sta Roger Banks in Steve Ward predstavila prvo stanje GoE velikosti 9×33 celic. Odkritje je bilo objavljeno v [43]. Kasneje je Don Woods [43, 44] s pomočjo računalnika dokazal, da je stanje res GoE.

Woodsov algoritem deluje po principu sestopanja (glej dodatek C). Polje celic razdeli v vrstice. Začne z eno vogalno celico, katere nabor predslik je kar nabor okolic, ki se preslikajo v stanje celice. V naslednjem koraku opazuje sosednjo celico v vrstici. Za vse kombinacije okolic dodane celice in obstoječi nabor predslik preveri, če se ujemajo, kjer se okolice prekrivajo. Če se ujemajo, doda sestavljeno predsliko v nabor, sicer predslika izpade iz nabora. Ko pride algoritem do konca vrstice, nadaljuje v naslednji vrstici, le da je sedaj prekrivanje med naborom predslik in okolico dodane celice drugačno (več sosednjih celic namesto ene). Tako algoritem nadaljuje do konca celotne konfiguracije. Če je končni nabor predslik prazen, je opazovana konfiguracija GoE. Nabor predslik in s tem poraba pomnilnika in procesnega časa ostajajo relativno majhni v primerjavi z

naborom vseh stanj. Temu je tako, ker imajo v praksi tudi deli konfiguracije GoE malo predslik. Algoritem bi bil za konfiguracije z dosti predslikami veliko bolj pomnilniško in časovno potraten.

Nicolay Beluchenko je poiskal večje število stanj GoE velikosti 11×11 [2]. Vsaj v enem primeru je uporabljal podoben algoritem kakor Woods [3]. Začel je z osrednjo celico in dodajal nove celice v spirali okoli osrednje. Za vrednost dodane celice je uporabil tisto, ki je dala manjši nabor predslik. Algoritem zaključi, ko z novo dodano celico ni več možno tvoriti predslik.

Leta 1974 je Duparc [21, 22] razvil drugačen algoritem, ter z njim poiskal stanja GoE velikosti 6×122 in 6×117. Algoritem uporablja teorijo končnih avtomatov in regularnih jezikov, ki je v osnovi namenjena 1D sistemom. Duparc je celice iz vrstice 2D polja združil v simbole regularnega jezika, zaporedje več vrstic pa predstavlja besedo. Duparcov algoritem ne obravnava vseh stanj vrstice enakovredno, temveč se osredotoča na vrstice, ki zmanjšujejo nabor predslik. Obravnavanje vseh stanj vrstic bi namreč preseglo pomnilniške sposobnosti takratne in sedanje strojne opreme že za kratke vrstice. Dokaže le, da za vrstice dolžine 1 ne obstaja stolpec, ki bi bil stanje GoE. Originalni članek je v francoščini, tako da lahko algoritem opišem le na podlagi tega, kako ga opisujejo drugi [10, 15].

V zadnjih letih podoben algoritem uporablja Steven Eker (CSL, SRI International, California, USA) [19]. Dokazal je, da za pravokotne konfiguracije višine 2 in 3 ne obstajajo stanja GoE. Dokaz za konfiguracije višine 4 z danim algoritmom še vedno presega sposobnosti sodobne strojne opreme. Število stanj avtomata, ki opisuje jezik GoE, namreč raste eksponentno z višino. Elker je poiskal stanja GoE velikosti 9×11, 8×12 in 5×83. Na žalost raziskovalcu delodajalec prepoveduje objavo rezultatov in podrobnosti, dokler ni opravljen interni pregled. Tako imamo za sedaj na voljo le zgoraj navedene podatke.

Drugačen algoritem uporablja Marijn Heule [23]. S sodelavci je zapisal vsa prekrivanja za določeno stanje v obliki binarnih enačb. Te je nato predal orodju za reševanje problemov SAT. Če rešitev ne obstaja, pomeni, da je stanje GoE. Uporabljena je dodatna predpostavka, da bo rešitev zrcalno simetrična v obeh dimenzijah, kar občutno zmanjša velikost problema. Algoritem je še dodatno optimiziran za iskanje stanj GoE in omogoča pregled večjega števila podobnih stanj. Ob spremembi vrednosti ene same celice omogoča uporabo delnih rezultatov prejšnjega izračuna.

5.2 Nesistematično iskanje predslik GoL

Erlan [18] je predstavil igro, pri kateri se za dano stanje GoL išče predslike. To dejansko ni implementacija algoritma za iskanje predslik, je pa vzpodbudil druge k iskanju takega algoritma.

Spletna stran `kaggle.com` je pripravila tekmovanje [1], kjer so morali kandidati reševati problem iskanja predslik. Podali so nabor stanj, za katere je treba poiskati predslike več korakov v preteklost. Podali so tudi nabor učnih sekvenc. Pričakovali so rešitev, ki temelji na strojnem učenju ali optimizaciji, torej fokus ni bil na eksaktni rešitvi. Cilj tudi nikakor

ni bil prešteti ali izpisati vse predslike, temveč so se morali kandidati le čim bolj približati učni sekvenci.

Pri iskanju implementacij algoritma, sem našel Atabot [16], Celični kronometer [17] in program, baziran na metodi SAT [36]. Cilj teh algoritmov sicer je iskanje predslik, ampak iskanje ni sistematično in cilj ni nabor vseh možnih predslik. Algoritmi se tudi poslužujejo določene hevristike pri iskanju.

Našel sem še aplikacijo, napisano v jeziku APL [24]. Iz navedenega primera je razvidno, da je aplikacija sposobna izpisovati predslike. Avtorja mi ni uspelo kontaktirati, tako da ne morem podati več podrobnosti.

5.3 Sistematični algoritmi za štetje in izpis predslik

Našel sem le eno aplikacijo, imenovano *RetroGUI*, ki je namenjena štetju in izpisu vseh predslik [15]. Neil Bickford prav tako kakor ostali procesira konfiguracijo po vrsticah, znotraj vrstice pa lahko uporablja različne algoritme (Woods, Duparc). Bickford opiše tudi algoritem, ki namesto procesiranja po vrsticah združuje predslike kvadratnih podsklopov konfiguracij. Na žalost poda primerjavo med različnimi algoritmi le kakor procesni čas za posamezno implementacijo, ne pa tudi bolj splošne ocene maksimalne in povprečne procesne/pomnilniške zahtevnosti.

Poglavje 6

Sklepne ugotovitve

6.1 Dokazovanje

Magistrska naloga je bolj skopa z dokazi. Za 1D CA sem uporabljal neposredno dokaze iz teorije grafov [30], pri 2D CA pa mreža predslik ni običajen graf, tako da dokazovanje s teorijo grafov ni tako preprosto. Verjetno pa je možno pretvoriti dani graf v dualno obliko s transformacijo zvezda-trikot. Če obstaja dualna oblika grafa, kjer ni ploskev, temveč samo vozlišča in povezave, bi jo lahko uporabili za izpeljavo in dokazovanje dejanske računske kompleksnosti problema.

Nekajkrat sem preveril rezultate, ki jih daje implementacija algoritma. Za dano vhodno stanje sem vsako predsliko s tranzicijsko funkcijo pretvoril nazaj v sedanjost in rezultat primerjal z vhodnim stanjem. Preveril sem še, če so vse naštete predslike enolične. Za nekaj manjših primerov, sem tudi poiskal predslike z metodo surove sile. Na ta način nisem našel nobene napake v algoritmu.

Nazadnje sem še primerjal rezultate s programom RetroGUI [15] in ugotovil, da se ujemajo.

6.2 Doprinos

Večina znanih algoritmov (naprimer Woods) uporablja metodo sestopanja. Izkažejo se, če je predslik relativno malo (naprimer iskanje GoE). So pa neprimerni za situacije, ko je predslik relativno veliko, ker hranijo vse predslike v pomnilniku ves čas izvajanja. Morda bi bilo možno optimizirati porabo pomnilnika, tako da bi algoritem hranil le tekoče razlike med predslikami, namesto da jih hrani v celoti.

Glavni doprinos magistrske naloge je aplikacija naprednih algoritmov razvitih za 1D CA na 2D problem. Tukaj opisan algoritem za štetje predslik ima predvidljivo porabo pomnilnika, ki je asimptotično manjša od porabe pomnilnika pri sestopanju $O(C^{N_x N_y})$ (podajam

brez dokaza). Algoritem za izpis predslik je striktno ločen od štetja in ima linearno zahtevnost v odvisnosti od števila predslik, kar je optimalno.

Grafična upodobitev mreža predslik omogoča boljše razumevanje problema predslik in lahko olajša dokumentiranje algoritmov. Obtežena mreža predslik tudi nakazuje, prostorsko razporeditev izgube informacije.

Na zadnje, je opisani algoritem zamišljen univerzalno, za poljuben 2D CA, čeprav univerzalnost v praksi ni ravno pomembna, saj izven raziskovalcev GoL ni dosti zanimanja za teoretične probleme, kakor je iskanje predslik.

6.3 Zanimivi tematsko povezani problemi

Probleme, povezane z iskanjem predslik, lahko delimo v nivoje glede na zahtevnost:

1. Določitev, ali obstajajo predslike za dano trenutno stanje sistema.
2. Štetje predslik za dano trenutno stanje sistema.
3. Naštevanje konfiguracij predslik.
4. Vprašanje reverzibilnosti sistema.
5. Jezik vseh stanj GoE.

V magistrskem delu opišem algoritem, ki rešuje prve tri probleme. Preostala problema sta si sorodna. Problem reverzibilnosti je na splošno dokazano nerešljiv [31]. Problem jezika stanj GoE je za 1D preprosto rešljiv, za 2D pa je verjetno podobno kakor vprašanje reverzibilnosti nerešljiv.

6.3.1 Analiza 2D celičnih avtomatov s pomočjo končnih avtomatov

Problem jezika stanj brez predslik bi potreboval teorijo 2D formalnega jezika, ki še ne obstaja. Kljub temu je možno zastaviti probleme, ki se jih da preslikati v končni avtomat. Na primer, če obravnavamo vrstice kakor simbole jezika, lahko sestavimo končni avtomat. Simboli (stanja vrstic) definirajo prehode med stanji končnega avtomata. Stanja avtomata pa so vsi možni nabori robov vrstice. Pomembna je le prisotnost roba, ne pa tudi utež. Poln nabor vsebuje vse možne robove, prazen nabor pa ne vsebuje nobenega roba. Vsako zaporedje vrstic (simbolov), ki v avtomatu pelje iz polnega v prazno stanje, je konfiguracija GoE. Zanke v takem avtomatu pa so sorodne delcem.

Opisani pristop postane nepraktičen že za kratke vrstice. Število vseh možnih robov raste eksponentno z dolžino vrstice N_x. To število, $|S|^{(M_y-1)(M_x-1+N_x)}$, je enako številu vseh načinov, na katere se lahko okolici dveh vrstic prekrivata. Stanja avtomata pa so vsi možni nabori robov. Nabor lahko opišemo kakor binarni niz, kjer vsak bit pomeni prisotnost ali odsotnost enega od robov. Vseh stanj končnega avtomata je tako $2^{|S|^{(M_y-1)(M_x-1+N_x)}}$. Število stanj avtomata postane neobvladljivo že za kratke vrstice.

Najkrajša beseda GoE znotraj regularnega jezika stanj GoE je tista, ki po najkrajši poti in brez zank pripelje iz polnega v prazen nabor. Pri 1D CA se je izkazalo, da je dolžina najkrajše besede GoE povezana s kompleksnostjo prostorsko-časovnih vzorcev, ki se pojavljajo pri danem pravilu. Podobno so verjetno tudi pri 2D CA konfiguracije GoE večje pri pravilih, ki omogočajo kompleksno dinamiko delcev. Verjetno bi bilo možno to izkoristiti za iskanje zanimivih pravil v okolici quad.

6.3.2 Izboljšave podanega algoritma

Implementacijo algoritma je možno izboljšati, tako da zahtevnost ne raste tako hitro z velikostjo problema. Za sedaj vidim samo rešitev, ki se osredotoča na povprečno zahtevnost, namesto na maksimalno zahtevnost. Na primer, sedaj algoritem rezervira pomnilnik za uteži vseh možnih robov, dovolj pa bi bilo, če bi bil rezerviran pomnilnik samo za robove z utežjo večjo od nič. Sploh pri iskanju stanj GoE je robov malo v primerjavi z vsemi možnimi. Tak pristop bi dal dosti manjšo porabo pomnilnika in verjetno tudi znižal čas procesiranja.

Morda bi bilo možno vnaprej določiti zgornjo mejo števila ne-ničelnih robov. To bi omogočalo zgodnjo oceno, če je izračun za dan problem izvedljiv. Na primer, če bi v vrstičnem procesiranju obravnavali vse začetne robove hkrati, namesto vsakega posebej, bi dobili prevelik nabor končnih robov. Tak nabor bi vedno vseboval vse pravilne končne robove in še nekaj napačnih. Ampak skupno število bi bilo lahko občutno manjše od polnega nabora. To število bi lahko uporabili za oceno porabe pomnilnika in časa. Na ta način bi bilo možno reševati probleme, ki so na meji tega, kar zmore sodobna strojna oprema.

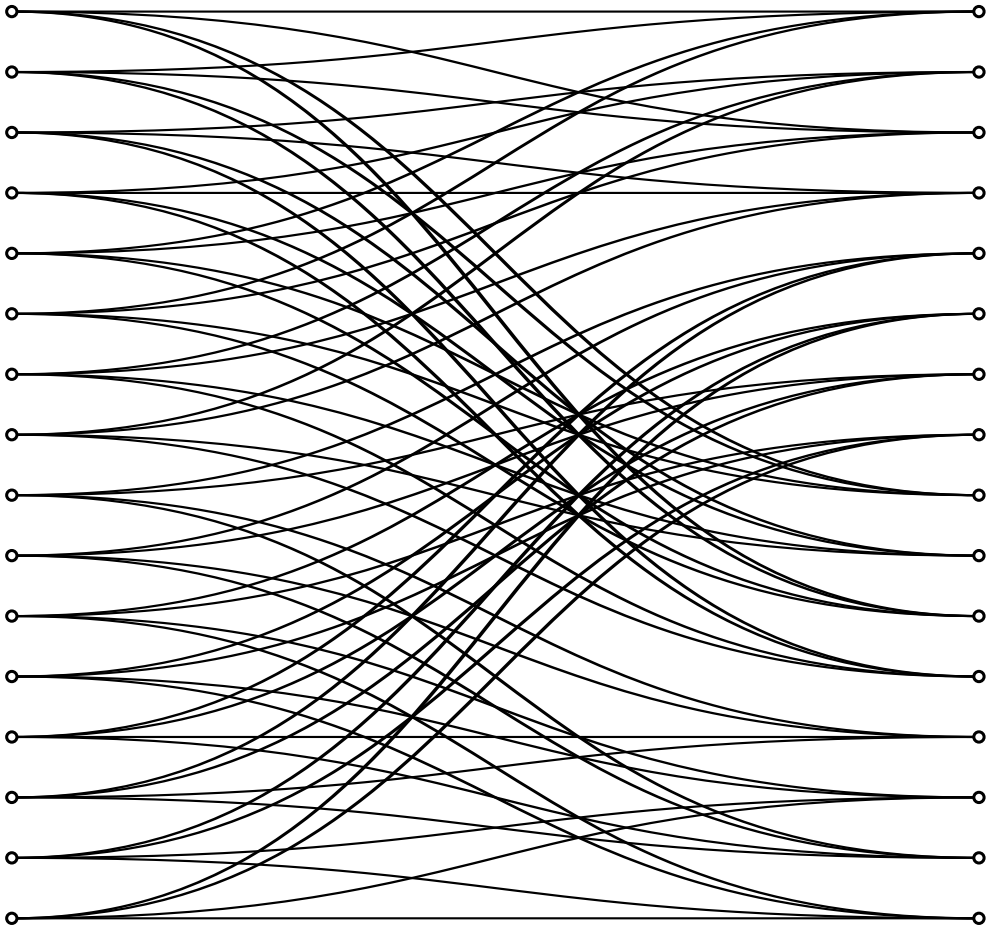
Samo implementacijo algoritma je tudi možno optimizirati. Univerzalnost algoritma, predvsem univerzalnost velikosti okolice, prispeva dosti režijskih stroškov. Če bi bil algoritem specializiran samo za quad ali GoL, bi se lahko izognil precej operacijam. Podobno velja za nabor stanj celice. Za binarne CA je možno optimizirati operacije, ki pretvarjajo 2D nize celic v cela števila in obratno. Binarne podatke je možno kompaktno pakirati v pomnilnik. Mogoče je tudi uporabiti logične operacije nad celo besedo celic, namesto procesiranja vsake celice posebej.

Procesiranje v algoritmu poteka v gnezdenih zankah in v znanem vrstnem redu. Le manjši del dostopov do pomnilnika je naključen. Zaporednost podatkov že sama po sebi omogoča dobro optimizacijo dostopa do predpomnilnika, tako da na tem področju ni dosti očitnih optimizacij.

Dodatek A

De Bruijnov diagram za GoL

Slika A.1 prikazuje del De Bruijnovega diagrama za GoL. Graf s 16 vozlišči in 64 povezavami je očitno preveč kompleksen, da bi bil primeren za razlago algoritma. Število vseh možnih okolic (ploskev v diagramu) pa je celo 512.



Slika A.1: Mreža ene celice za binarni CA z Moorovo okolico $M_x = M_y = 3$.

Dodatek B

Primer delovanja algoritma za CA z okolico quad

V naslednjem primeru je prikazano štetje in iskanje predslik za binarni CA (celica ima lahko 2 stanji) z okolico velikosti 2×2 in pravilom številka `0x725A`.

Majhna konfiguracija velikosti 2×2 je shranjena v datoteki `test_2x2.cas`. Uporabil bi večjo konfiguracijo, a na žalost ne poznam pravila, ki bi za večjo konfiguracijo dal majhno število predslik.

Uporabljen je program [26] z verzijo `v1`.

```
$ ./ca2d_preimages 2 2 2 0x725a 2 2 test_2x2.cas
CA parameters:
  sts  : 2
  ngb  : siz={2,2}, a=4, n=16
  ovl-y: siz={1,2}, a=2, n=4
  ovl-x: siz={2,1}, a=2, n=4
  rem-y: siz={1,2}, a=2, n=4
  rem-x: siz={2,1}, a=2, n=4
  ver  : siz={1,1}, a=1, n=2
  shf-y: siz={1,1}, a=1, n=2
  shf-x: siz={1,1}, a=1, n=2

RULE = 0x725a
  tab [0] = [[0,0],[0,0]] = 0
  tab [1] = [[1,0],[0,0]] = 1
  tab [2] = [[0,1],[0,0]] = 0
  tab [3] = [[1,1],[0,0]] = 1
  tab [4] = [[0,0],[1,0]] = 1
  tab [5] = [[1,0],[1,0]] = 0
  tab [6] = [[0,1],[1,0]] = 1
  tab [7] = [[1,1],[1,0]] = 0
  tab [8] = [[0,0],[0,1]] = 0
  tab [9] = [[1,0],[0,1]] = 1
  tab [A] = [[0,1],[0,1]] = 0
  tab [B] = [[1,1],[0,1]] = 0
  tab [C] = [[0,0],[1,1]] = 1
```

```
  tab [D] = [[1,0],[1,1]] = 1
  tab [E] = [[0,1],[1,1]] = 1
  tab [F] = [[1,1],[1,1]] = 0

CA configuration siz [y,x] = [2,2]:
 1 0
 0 1

NETWORK: edge weights from forward/backward direction,
         summed preimages
net [dy=0][y=0] = [1 1 1 1 1 1 1 1 ]
net [dy=0][y=1] = [2 2 0 2 3 3 0 1 ]
net [dy=0][y=2] = [0 3 5 0 0 2 7 2 ]  cnt [0] = 19
net [dy=1][y=0] = [0 4 4 0 0 4 5 2 ]  cnt [1] = 19
net [dy=1][y=1] = [2 0 3 4 2 0 1 1 ]
net [dy=1][y=2] = [1 1 1 1 1 1 1 1 ]

preimage i=0:  CA configuration siz [y,x] = [3,3]:
 1 0 0
 0 0 0
 0 1 0

preimage i=1:  CA configuration siz [y,x] = [3,3]:
 1 0 0
 0 0 0
 0 1 1

preimage i=2:  CA configuration siz [y,x] = [3,3]:
 1 0 0
 0 0 1
 0 1 0

preimage i=3:  CA configuration siz [y,x] = [3,3]:
 1 0 0
 0 0 1
 0 1 1

preimage i=4:  CA configuration siz [y,x] = [3,3]:
 0 1 0
 1 1 0
 1 0 0

preimage i=5:  CA configuration siz [y,x] = [3,3]:
 0 1 0
 1 1 0
 1 0 1

preimage i=6:  CA configuration siz [y,x] = [3,3]:
 0 1 0
 1 1 0
 0 1 1

preimage i=7:  CA configuration siz [y,x] = [3,3]:
 0 1 0
 1 1 0
 1 1 1

preimage i=8:  CA configuration siz [y,x] = [3,3]:
```

```
 1 0 1
 0 0 0
 0 1 0

preimage i=9:  CA configuration siz [y,x] = [3,3]:
 1 0 1
 0 0 0
 0 1 1

preimage i=10:  CA configuration siz [y,x] = [3,3]:
 1 0 1
 0 0 1
 0 1 0

preimage i=11:  CA configuration siz [y,x] = [3,3]:
 1 0 1
 0 0 1
 0 1 1

preimage i=12:  CA configuration siz [y,x] = [3,3]:
 0 1 1
 1 1 0
 1 0 0

preimage i=13:  CA configuration siz [y,x] = [3,3]:
 0 1 1
 1 1 0
 1 0 1

preimage i=14:  CA configuration siz [y,x] = [3,3]:
 0 1 1
 1 1 0
 0 1 1

preimage i=15:  CA configuration siz [y,x] = [3,3]:
 0 1 1
 1 1 0
 1 1 1

preimage i=16:  CA configuration siz [y,x] = [3,3]:
 0 1 1
 1 1 1
 1 0 0

preimage i=17:  CA configuration siz [y,x] = [3,3]:
 1 1 1
 0 0 1
 0 1 0

preimage i=18:  CA configuration siz [y,x] = [3,3]:
 1 1 1
 0 0 1
 0 1 1
```

Dodatek C

Opis algoritma Dona Woodsa

Prilagam pisno izmenjavo z avtorjem [1] prvega algoritma za iskanje predslik GoL.

```
Don Woods <don@icynic.com>        Thu, Jul 28, 2016 at 11:45 PM
To: iztok.jeras@gmail.com
> Dear Mr. Woods,
>
> I am developing an algorithm for computing preimages of 2D CA. It is based
> on the latest research for computig preimages of 1D CA.
> I am using the 2x2 quad neighborhood in the examples instead of the 3x3
> neighborhood used in Life.
> https://github.com/jeras/fri-magisterij (work in progress and Slovene
> language)
> https://github.com/jeras/preimages-2D (not working yet)
>
> As part of my masters thesis I have to compare my algorithm to existing
> ones.
> You are known as the first person to prove a pattern to be a Garden of Eden
> orphan in the Game of Life.
> Could you send me some references to this proof? I was looking for an
> article to reference, but a generic description (preferably written by you)
> or source code would be as good.
>
> Regards,
> Iztok Jeras
>

Oof, it's been quite a while; let me see what I can remember.

The "proof" was programmatic, i.e. I wrote a program to compute
predecessors to a given position, and the program claimed there were no
predecessors to the particular position that someone had theorised was
a Garden of Eden.  I did not have a formal proof of correctness for the
program.  (I was a teenager when I wrote it!)  I don't think I still
have the source code anywhere, I'm afraid.  (I might have a paper
printout of it somewhere but don't have time to search right now.)
```

[1] Don Woods /urlhttp://www.icynic.com/ don/

The program worked by considering each cell in turn and looking at all
possible 3x3 predecessors that would produce the given state. It then
looked at the next cell to the right, looking for 3x3 predecessors that
were consistent with the 6 overlapping cells from the other, etc. (In
practice, the 3x3 neighborhoods were indexed so that I could quickly
find which of the 8 possible righthand extensions worked.) When it had
a 3x(N+2) predecessor for the top row of N, it continued at the left of
the next row, then across that row, etc. Obviously in those later rows
there was only one new predecessor cell being introduced for each new
current cell after the leftmost, so the combinatoric explosion was not
excessive, particularly since the GoE pattern by design did not have
many potential predecessors even for subsections. I also deliberately
oriented the target pattern with the short dimension going across, so
that I more quickly reached the point of extending the predecessor one
cell at a time in later rows.

My program normally included a 1-wide border around the area being
backtracked, to ensure there were no artifacts introduced, but for the
GoE proof I restricted it to the non-empty rectangular boundary, which
I think was 9 by 33? Thus, as I think was mentioned at the time, what
I actually demonstrated was a stronger result: there was no predecessor
that led to a state that included the GoE as a 9x33 subsection.

I hope that helps. Let me know if you have further questions.

-- Don Woods

Don Woods <don@icynic.com> Tue, Aug 16, 2016 at 6:42 PM
To: iztok.jeras@gmail.com
> I found the time to read your response just now.
>
> Your description of the algorithm confirms what I imagined based on third
> party descriptions.
> Your details regarding combinatorial explosion which affects memory
> consumption and processing time were extra helpful.
>
> May I cite this email in my master thesis, since there are no other direct
> sources I would know of?
>
> Regards,
> Iztok Jeras

Permission granted. Good luck with your thesis!

-- Don Woods

Dodatek D

Izvorna koda implementacije algoritma

Prilagam izvorno kodo, ki se sicer nahaja na naslovu:
`https://github.com/jeras/preimages-2D`

Zanimivi sta predvsem funkciji `ca1d_net` in `ca2d_net` ki se nahajata proti koncu datoteke:
`preimages-2D/ca2d_net.c`

preimages–2D/ca2d.h

```
#ifndef CA2D_H
#define CA2D_H

// math libraries
#include <gmp.h>

#define uintca_t long long int unsigned

typedef struct {
    // basic parameters
    int unsigned y;  // Y size
    int unsigned x;  // X size
    // calculated parameters
    int unsigned a;  // area (x*y)
    int unsigned n;  // number of states
} ca2d_size_t;

typedef struct {
    // basic parameters
    int unsigned sts;  // number of cell states
    ca2d_size_t  ngb;  // neighborhood size
    mpz_t        rule; // rule
    // calculated neighborhood/overlap parameters
    struct {
        ca2d_size_t y;
        ca2d_size_t x;
    } ovl;             // overlap size
    struct {
        ca2d_size_t y;
        ca2d_size_t x;
    } rem;             // remainder size (remainder and overlap add to a whole neighborhood)
    ca2d_size_t ver;   // vertice size (corner overlap)
    struct {
        ca2d_size_t y;
        ca2d_size_t x;
    } shf;             // shift size (shift and vertice add to a whole overlap)
    // rule table
    int unsigned *tab;
} ca2d_t;

int ca2d_update (ca2d_t *ca2d);
int ca2d_bprint (ca2d_t  ca2d);

#endif // CA2D_H
```

preimages–2D/ca2d.c

```
// user interface libraries
```

```c
#include <stdio.h>
#include <stdlib.h>

// math libraries
#include <stdint.h>
#include <math.h>
#include <gmp.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"
#include "ca2d_cfg.h"
#include "ca2d_fwd.h"
#include "ca2d_net.h"

////////////////////////////////////////////////////////////////////////////////
// main
////////////////////////////////////////////////////////////////////////////////

int main (int argc, char **argv) {
    // configuration
    ca2d_t  ca2d;
    ca2d_size_t  siz;
    char *filename;
    FILE  file;

    // read input arguments
    if (argc < 8) {
        fprintf (stderr, "Usage:\t%s␣STATES␣NEIGHBORHOOD_SIZE_Y␣NEIGHBORHOOD_SIZE_X␣RULE␣CA_SIZE_Y␣
    CA_SIZE_X␣ca_state_filename.cas\n", argv[0]);
        return (1);
    }
    ca2d.sts   = strtoul (argv[1], 0, 0);
    ca2d.ngb.y = strtoul (argv[2], 0, 0);
    ca2d.ngb.x = strtoul (argv[3], 0, 0);
    mpz_init_set_str (ca2d.rule, argv[4], 0);
    siz.y = strtoul (argv[5], 0, 0);
    siz.x = strtoul (argv[6], 0, 0);
    filename = argv[7];

    // update ca2d structure
    ca2d_update (&ca2d);
    printf ("\n");

    // read CA configuration file
    int unsigned cai [siz.y] [siz.x];
    int unsigned cao [siz.y] [siz.x];
    ca2d_read (filename, siz, cai);
    ca2d_print (siz, cai);
    printf ("\n");

    // calculate network
    mpz_t cnt [2];
    ca2d_size_t siz_pre = {siz.y+ca2d.ver.y, siz.x+ca2d.ver.x};
    siz_pre.a = siz_pre.y * siz_pre.x;
    int unsigned (* p_list) [] [siz_pre.y] [siz_pre.x];
    int unsigned (*   list)    [siz_pre.y] [siz_pre.x];
    ca2d_net (ca2d, siz, cai, cnt, &p_list);
    list = (void *)p_list;

    // calculate preimage from network
    int status;
    int unsigned preimage [siz_pre.y] [siz_pre.x];

    for (int unsigned d=0; d<2; d++) {
        gmp_printf ("cnt␣[%u]␣=␣%Zi\n", d, cnt [d]);
    }
    int unsigned error = 0;
    for (int unsigned i=0; i<mpz_get_ui(cnt[0]); i++) {
        printf ("preimage␣i=%u/%lu:␣␣", i, mpz_get_ui(cnt[0]));
        ca2d_print (siz_pre, list[i]);
        printf ("\n");
    }

    return (0);
}
```

preimages–2D/ca2d_array.h

```c
#ifndef CA2D_ARRAY_H
#define CA2D_ARRAY_H

// math libraries
#include <gmp.h>

////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

int unsigned ca2d_array_print     (ca2d_size_t s, int unsigned array[s.y][s.x]);
int unsigned ca2d_array_to_ui     (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int
    unsigned *number);
int unsigned ca2d_array_from_ui   (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int
    unsigned  number);
int unsigned ca2d_array_to_mpz    (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x],
    mpz_t number);
int unsigned ca2d_array_from_mpz  (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x],
    mpz_t number);
```

```
int unsigned ca2d_array_slice     (ca2d_size_t is, ca2d_size_t ob, ca2d_size_t os, int unsigned ia[is.y
    ][is.x], int unsigned oa[os.y][os.x]);
int unsigned ca2d_array_fit       (ca2d_size_t is, ca2d_size_t ob, ca2d_size_t os, int unsigned ia[is.y
    ][is.x], int unsigned oa[os.y][os.x]);
int unsigned ca2d_array_combine_x (ca2d_size_t is0, ca2d_size_t is1, int unsigned ia0[is0.y][is0.x],
     int unsigned ia1[is1.y][is1.x], int unsigned oa[is0.y][is1.x+is0.x]);
int unsigned ca2d_array_combine_y (ca2d_size_t is0, ca2d_size_t is1, int unsigned ia0[is0.y][is0.x],
     int unsigned ia1[is1.y][is1.x], int unsigned oa[is1.y+is0.y][is0.x]);

#endif
```

preimages–2D/ca2d_array.c

```
////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

// user interface libraries
#include <stdio.h>

// math libraries
#include <gmp.h>

#include "ca2d.h"

int unsigned ca2d_array_print (ca2d_size_t s, int unsigned array[s.y][s.x]) {
    printf ("[");
    for (int unsigned y=0; y<s.y; y++) {
        printf ("%s[", y ? "," : "");
        for (int unsigned x=0; x<s.x; x++) {
            printf ("%s%u", x ? "," : "", array [y] [x]);
        }
        printf ("]");
    }
    printf ("]");
    return 0;
}

int unsigned ca2d_array_to_ui (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int
     unsigned *number) {
    *number = 0;
    int unsigned mul = 1;
    for (int unsigned y=0; y<s.y; y++) {
        for (int unsigned x=0; x<s.x; x++) {
            *number += array [y] [x] * mul;
            mul *= base;
        }
    }
    return 0;
}

int unsigned ca2d_array_from_ui (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int
     unsigned number) {
    for (int unsigned y=0; y<s.y; y++) {
        for (int unsigned x=0; x<s.x; x++) {
            array [y] [x] = number % base;
            number /= base;
        }
    }
    return 0;
}

int unsigned ca2d_array_to_mpz (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], mpz_t
     number) {
    mpz_t mul;
    mpz_init (mul);
    mpz_set_ui (mul, 1);
    mpz_set_ui (number, 0);
    for (int unsigned y=0; y<s.y; y++) {
        for (int unsigned x=0; x<s.x; x++) {
            mpz_addmul_ui (number, mul, array [y] [x]);
            mpz_mul_ui (mul, mul, base);
        }
    }
    return 0;
}

int unsigned ca2d_array_from_mpz (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], mpz_t
      number) {
    mpz_t num;
    mpz_init_set (num, number);
    for (int unsigned y=0; y<s.y; y++) {
        for (int unsigned x=0; x<s.x; x++) {
            array [y] [x] = mpz_tdiv_q_ui (num, num, base);
        }
    }
    return 0;
}

// get slice value
int unsigned ca2d_array_slice (
    ca2d_size_t is,
    ca2d_size_t ob, ca2d_size_t os,
    int unsigned ia[is.y][is.x], int unsigned oa[os.y][os.x])
{
    for (int unsigned y=0; y<os.y; y++) {
        for (int unsigned x=0; x<os.x; x++) {
```

```
            oa[y][x] = ia[ob.y+y][ob.x+x];
        }
    }
    return 0;
}

// set slice value
int unsigned ca2d_array_fit (
    ca2d_size_t is,
    ca2d_size_t ob, ca2d_size_t os,
    int unsigned ia[is.y][is.x], int unsigned oa[os.y][os.x])
{
    for (int unsigned y=0; y<os.y; y++) {
        for (int unsigned x=0; x<os.x; x++) {
            ia[ob.y+y][ob.x+x] = oa[y][x];
        }
    }
    return 0;
}

int unsigned ca2d_array_combine_x (
    ca2d_size_t is0, ca2d_size_t is1,
    int unsigned ia0[is0.y][is0.x], int unsigned ia1[is1.y][is1.x], int unsigned oa[is0.y][is1.x+is0.x
    ])
{
    for (int unsigned y=0; y<is0.y; y++) {
        for (int unsigned x=0; x<is0.x; x++) {
            oa[y][x      ] = ia0[y][x];
        }
        for (int unsigned x=0; x<is1.x; x++) {
            oa[y][x+is0.x] = ia1[y][x];
        }
    }
    return 0;
}

int unsigned ca2d_array_combine_y (
    ca2d_size_t is0, ca2d_size_t is1,
    int unsigned ia0[is0.y][is0.x], int unsigned ia1[is1.y][is1.x], int unsigned oa[is1.y+is0.y][is0.x
    ])
{
    for (int unsigned x=0; x<is0.x; x++) {
        for (int unsigned y=0; y<is0.y; y++) {
            oa[y      ][x] = ia0[y][x];
        }
        for (int unsigned y=0; y<is0.y; y++) {
            oa[y+is0.y][x] = ia1[y][x];
        }
    }
    return 0;
}
```

preimages–2D/ca2d_base.c

```
// math libraries
#include <stdio.h>
#include <math.h>
#include <gmp.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"
#include "ca2d_cfg.h"

int ca2d_update (ca2d_t *ca2d) {
    // overlay sizes
    ca2d->ovl.y   = (ca2d_size_t) {(ca2d->ngb.y-1) , (ca2d->ngb.x  )};
    ca2d->ovl.x   = (ca2d_size_t) {(ca2d->ngb.y  ) , (ca2d->ngb.x-1)};
    // remainder sizes
    ca2d->rem.y   = (ca2d_size_t) {(             1) , (ca2d->ngb.x  )};
    ca2d->rem.x   = (ca2d_size_t) {(ca2d->ngb.y  ) , (             1)};
    // vertice size
    ca2d->ver     = (ca2d_size_t) {(ca2d->ngb.y-1) , (ca2d->ngb.x-1)};
    // shift sizes
    ca2d->shf.y   = (ca2d_size_t) {(ca2d->ngb.y-1) , (             1)};
    ca2d->shf.x   = (ca2d_size_t) {(             1) , (ca2d->ngb.y-1)};
    // areas
    ca2d->ngb.a   = ca2d->ngb.y   * ca2d->ngb.x  ;
    ca2d->ovl.y.a = ca2d->ovl.y.y * ca2d->ovl.y.x;
    ca2d->ovl.x.a = ca2d->ovl.x.y * ca2d->ovl.x.x;
    ca2d->rem.y.a = ca2d->rem.y.y * ca2d->rem.y.x;
    ca2d->rem.x.a = ca2d->rem.x.y * ca2d->rem.x.x;
    ca2d->ver.a   = ca2d->ver.y   * ca2d->ver.x  ;
    ca2d->shf.y.a = ca2d->shf.y.y * ca2d->shf.y.x;
    ca2d->shf.x.a = ca2d->shf.x.y * ca2d->shf.x.x;
    // number of states
    ca2d->ngb.n   = (size_t) pow (ca2d->sts, ca2d->ngb.a  );
    ca2d->ovl.y.n = (size_t) pow (ca2d->sts, ca2d->ovl.y.a);
    ca2d->ovl.x.n = (size_t) pow (ca2d->sts, ca2d->ovl.x.a);
    ca2d->rem.y.n = (size_t) pow (ca2d->sts, ca2d->rem.y.a);
    ca2d->rem.x.n = (size_t) pow (ca2d->sts, ca2d->rem.x.a);
    ca2d->ver.n   = (size_t) pow (ca2d->sts, ca2d->ver.a  );
    ca2d->shf.y.n = (size_t) pow (ca2d->sts, ca2d->shf.y.a);
    ca2d->shf.x.n = (size_t) pow (ca2d->sts, ca2d->shf.x.a);
    // rule table
    ca2d_rule_table (ca2d);
```

```
//    ca2d_bprint (*ca2d);
    return (0);
}

int ca2d_bprint (ca2d_t ca2d) {
    printf("CA parameters:\n");
    printf("sts  : %u\n", ca2d.sts);
    printf("ngb  : siz={%u,%u}, a=%u, n=%u\n", ca2d.ngb.y  , ca2d.ngb.x  , ca2d.ngb.a  , ca2d.ngb.n  );
    printf("ovl-y: siz={%u,%u}, a=%u, n=%u\n", ca2d.ovl.y.y, ca2d.ovl.y.x, ca2d.ovl.y.a, ca2d.ovl.y.n);
    printf("ovl-x: siz={%u,%u}, a=%u, n=%u\n", ca2d.ovl.x.y, ca2d.ovl.x.x, ca2d.ovl.x.a, ca2d.ovl.x.n);
    printf("rem-y: siz={%u,%u}, a=%u, n=%u\n", ca2d.rem.y.y, ca2d.rem.y.x, ca2d.rem.y.a, ca2d.rem.y.n);
    printf("rem-x: siz={%u,%u}, a=%u, n=%u\n", ca2d.rem.x.y, ca2d.rem.x.x, ca2d.rem.x.a, ca2d.rem.x.n);
    printf("ver  : siz={%u,%u}, a=%u, n=%u\n", ca2d.ver.y  , ca2d.ver.x  , ca2d.ver.a  , ca2d.ver.n  );
    printf("shf-y: siz={%u,%u}, a=%u, n=%u\n", ca2d.shf.y.y, ca2d.shf.y.x, ca2d.shf.y.a, ca2d.shf.y.n);
    printf("shf-x: siz={%u,%u}, a=%u, n=%u\n", ca2d.shf.x.y, ca2d.shf.x.x, ca2d.shf.x.a, ca2d.shf.x.n);
    printf("CA rule table:\n");
    ca2d_rule_print (ca2d);
    return (0);
}
```

preimages–2D/ca2d_cfg.h

```
#ifndef CA2D_CONFIGURATION_H
#define CA2D_CONFIGURATION_H

////////////////////////////////////////////////////////////////////////////////
// CA configuration handling
////////////////////////////////////////////////////////////////////////////////

int ca2d_read (char *filename, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]);
int ca2d_print (ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]);
int ca2d_lattice_compare (ca2d_size_t siz, int unsigned ca0 [siz.y] [siz.x], int unsigned ca1 [siz.y] [
    siz.x]);

#endif
```

preimages–2D/ca2d_cfg.c

```
////////////////////////////////////////////////////////////////////////////////
// CA configuration handling
////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

#include "ca2d.h"

// read CA state from file
int ca2d_read (char *filename, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]) {
    FILE *fp;
    fp = fopen (filename, "r");
    if (!fp) {
        fprintf (stderr, "ERROR: file %s not found\n", filename);
        return (1);
    }
    for (int unsigned y=0; y<siz.y; y++) {
        for (int unsigned x=0; x<siz.x; x++) {
            fscanf (fp, "%u", &ca [y] [x]);
        }
    }
    fclose (fp);
    return (0);
}

// print CA state
int ca2d_print (ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]) {
    printf ("CA configuration siz[ y  , x  ]=[%u,%u]:\n", siz.y, siz.x);
    for (int unsigned y=0; y<siz.y; y++) {
        for (int unsigned x=0; x<siz.x; x++) {
            printf (" %u", ca [y] [x]);
        }
        printf ("\n");
    }
    return (0);
}

// compare CA states
int ca2d_lattice_compare (ca2d_size_t siz, int unsigned ca0 [siz.y] [siz.x], int unsigned ca1 [siz.y] [
    siz.x]) {
    for (int unsigned y=0; y<siz.y; y++) {
        for (int unsigned x=0; x<siz.x; x++) {
            if (ca0 [y] [x] != ca1 [y] [x]) {
                return (1);
            }
        }
    }
    return (0);
}
```

preimages–2D/ca2d_rule.h

```
#ifndef CA2D_RULE_H
#define CA2D_RULE_H
```

```
// math libraries
#include <gmp.h>

////////////////////////////////////////////////////////////////////////////////
// rule handling
////////////////////////////////////////////////////////////////////////////////

// function for transforming the Game of Life rule description into a number
int ca2d_rule_gol ();

int ca2d_rule_table (ca2d_t *ca2d);
int ca2d_rule_print (ca2d_t ca2d);

#endif
```

preimages–2D/ca2d_rule.c

```
////////////////////////////////////////////////////////////////////////////////
// rule handling
////////////////////////////////////////////////////////////////////////////////

// user interface libraries
#include <stdio.h>
#include <stdlib.h>

// math libraries
#include <math.h>
#include <gmp.h>

// CA libraries
#include "ca2d.h"
#include "ca2d_array.h"

// function for transforming the Game of Life rule description into a number
int ca2d_rule_gol () {
    int unsigned len = 1<<9;
    int unsigned sum, out;
    mpz_t rule;
    mpz_init (rule);
    // loop through the rule table
    for (int i=len-1; i>=0; i--) {
        // calculate neighborhood sum
        sum = 0;
        for (int unsigned b=0; b<9; b++) {
            sum += ((i & 0b111101111) >> b) & 0x1;
        }
        // calculate GoL transition function
        if      (sum == 3)  out = 1;
        else if (sum == 2)  out = (i & 0b000010000) >> 4;
        else                out = 0;
        // add rule table element to rule number
        mpz_mul_ui (rule, rule, 2);
        mpz_add_ui (rule, rule, out);
    }
    // return rule
    gmp_printf ("RULE GoL = 0x%ZX\n", rule);
    return (0);
}

int ca2d_rule_table (ca2d_t *ca2d) {
    // neighborhood area
    // check if it is within allowed values, for example less then 9==3*3
    if ((ca2d->ngb.x == 0) || (ca2d->ngb.y == 0) || ((ca2d->ngb.y * ca2d->ngb.x) > 9)) {
        printf ("ERROR: neighborhood area %u is outside range [1:9].\n", ca2d->ngb.a);
        return (1);
    }

    mpz_t range;
    mpz_init (range);
    mpz_ui_pow_ui (range, ca2d->sts, ca2d->ngb.n);
    if (mpz_cmp (ca2d->rule, range) > 0) {
        gmp_printf ("ERROR: rule is outside of range = %Zi\n", range);
        return (1);
    }
    mpz_clear (range);

    // allocate rule table memory
    ca2d->tab = malloc (ca2d->ngb.n * sizeof(int unsigned));

    // rule table (conversion to base sts)
    mpz_t rule_q;
    mpz_init_set (rule_q, ca2d->rule);
    for (int unsigned i=0; i<ca2d->ngb.n; i++) {
        // populate transition function
        ca2d->tab[i] = mpz_tdiv_q_ui (rule_q, rule_q, ca2d->sts);
    }
    mpz_clear (rule_q);

    return (0);
}

int ca2d_rule_print (ca2d_t ca2d) {
    int unsigned a [ca2d.ngb.y] [ca2d.ngb.x];
    for (int unsigned n=0; n<ca2d.ngb.n; n++) {
        // convert table index into neighborhood status 2D array
        ca2d_array_from_ui (ca2d.sts, ca2d.ngb, a, n);
        // print rule table
```

```
        printf ("  tab[%X] = [", n);
        for (int unsigned y=0; y<ca2d.ngb.y; y++) {
            printf ("%s[", y ? "," : "");
            for (int unsigned x=0; x<ca2d.ngb.x; x++) {
                printf ("%s%u", x ? "," : "", a[y][x]);
            }
            printf ("]");
        }
        printf ("] = %u\n", ca2d.tab[n]);
    }
    return (0);
}
```

preimages–2D/ca2d_fwd.h

```
#ifndef CA2D_FORWARD_H
#define CA2D_FORWARD_H

////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

int unsigned ca2d_forward (ca2d_t ca2d, ca2d_size_t siz,
    int unsigned ai[siz.y][siz.x],
    int unsigned ao[siz.y-(ca2d.ngb.y-1)][siz.x-(ca2d.ngb.x-1)]
);

#endif
```

preimages–2D/ca2d_fwd.c

```
////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

// math libraries
#include <math.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"

int unsigned ca2d_forward (ca2d_t ca2d, ca2d_size_t siz,
    int unsigned ai[siz.y][siz.x],
    int unsigned ao[siz.y-(ca2d.ngb.y-1)][siz.x-(ca2d.ngb.x-1)]
) {
    int unsigned at [ca2d.ngb.y] [ca2d.ngb.x];
    int unsigned nt;
    for (int unsigned y=0; y<siz.y-(ca2d.ngb.y-1); y++) {
        for (int unsigned x=0; x<siz.x-(ca2d.ngb.x-1); x++) {
            ca2d_array_slice (siz, (ca2d_size_t) {y, x}, ca2d.ngb, ai, at);
            ca2d_array_to_ui (ca2d.sts, ca2d.ngb, at, &nt);
            ao [y] [x] = ca2d.tab [nt];
        }
    }
    return 0;
}
```

preimages–2D/ca2d_net.h

```
#ifndef CA2D_NETWORK_H
#define CA2D_NETWORK_H

////////////////////////////////////////////////////////////////////////////////
// CA preimage network
////////////////////////////////////////////////////////////////////////////////

int ca2d_net_print (ca2d_t ca2d, ca2d_size_t siz, int unsigned res [siz.y] [siz.x] [ca2d.ngb.n]);
int ca2d_net       (ca2d_t ca2d, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x], mpz_t cnt [2], int
     unsigned (** p_list) [] [siz.y+ca2d.ver.y] [siz.x+ca2d.ver.x]);

#endif
```

preimages–2D/ca2d_net.c

```
// user interface libraries
#include <stdio.h>
#include <stdlib.h>

// math libraries
//#include <stdint.h>
#include <math.h>
#include <gmp.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"
#include "ca2d_cfg.h"

//     // print edge counters
//     printf ("NETWORK: edge weights from forward/backward direction,\n");
```

```
//     printf ("         summed preimages\n");
//     for (int d=0; d<2; d++) {
//         for (int y=0; y<=siz.y; y++) {
//             printf ("net [dy=%u][y=%u] = [", d, y);
//             for (int unsigned edg=0; edg<edg_x; edg++) {
//                 gmp_printf ("%Zi ", net[d][y][edg]);
//             }
//             printf ("]");
//             if ((y-(1-d)*(siz.y))==0) {
//                 gmp_printf ("  cnt [%u] = %Zi", d, cnt [d]);
//             }
//             printf ("\n");
//         }
//     }
//     printf ("\n");

// tables for geting overlap values from neighborhood values
static int ca2d_net_table_v2o (ca2d_t ca2d, int unsigned v2o_y [2] [ca2d.ovl.y.n] [ca2d.shf.y.n],
                                            int unsigned v2o_x [2] [ca2d.ovl.x.n] [ca2d.shf.x.n]) {
    int unsigned ovl_ay [ca2d.ovl.y.y] [ca2d.ovl.y.x];
    int unsigned ovl_ax [ca2d.ovl.x.y] [ca2d.ovl.x.x];
    int unsigned ovl_ayo[ca2d.ovl.y.y] [ca2d.ovl.y.x];
    int unsigned ovl_axo[ca2d.ovl.x.y] [ca2d.ovl.x.x];
    int unsigned shf_ay [ca2d.shf.y.y] [ca2d.shf.y.x];
    int unsigned shf_ax [ca2d.shf.x.y] [ca2d.shf.x.x];
    int unsigned ver_a  [ca2d.ver.y  ] [ca2d.ver.x  ];

    for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
        ca2d_array_from_ui (ca2d.sts, ca2d.ovl.y, ovl_ay, ovl);
        for (int unsigned shf=0; shf<ca2d.shf.y.n; shf++) {
            ca2d_array_from_ui (ca2d.sts, ca2d.shf.y, shf_ay, shf);
            for (int unsigned d=0; d<2; d++) {
                ca2d_array_slice (ca2d.ovl.y, (ca2d_size_t) {0, d}, ca2d.ver, ovl_ay, ver_a);
                if (!d)  ca2d_array_combine_x (ca2d.shf.y, ca2d.ver, shf_ay, ver_a, ovl_ayo);  // shift
  added to the left  of overlap
                else     ca2d_array_combine_x (ca2d.ver, ca2d.shf.y, ver_a, shf_ay, ovl_ayo);  // shift
  added to the right of overlap
                ca2d_array_to_ui (ca2d.sts, ca2d.ovl.y, ovl_ayo, &v2o_y [d] [ovl] [shf]);
            }
        }
    }
    for (int unsigned ovl=0; ovl<ca2d.ovl.x.n; ovl++) {
        ca2d_array_from_ui (ca2d.sts, ca2d.ovl.x, ovl_ax, ovl);
        for (int unsigned shf=0; shf<ca2d.shf.x.n; shf++) {
            ca2d_array_from_ui (ca2d.sts, ca2d.shf.x, shf_ax, shf);
            for (int unsigned d=0; d<2; d++) {
                ca2d_array_slice (ca2d.ovl.x, (ca2d_size_t) {d, 0}, ca2d.ver, ovl_ax, ver_a);
                if (!d)  ca2d_array_combine_y (ca2d.shf.x, ca2d.ver, shf_ax, ver_a, ovl_axo);  // shift
  added to the left  of overlap
                else     ca2d_array_combine_y (ca2d.ver, ca2d.shf.x, ver_a, shf_ax, ovl_axo);  // shift
  added to the right of overlap
                ca2d_array_to_ui (ca2d.sts, ca2d.ovl.x, ovl_axo, &v2o_x [d] [ovl] [shf]);
            }
        }
    }
    return (0);
}

// tables for geting overlap values from neighborhood values
static int ca2d_net_table_n2o (ca2d_t ca2d, int unsigned n2o_y [2] [ca2d.ngb.n],
                                            int unsigned n2o_x [2] [ca2d.ngb.n]) {
    int unsigned ovl_ay [ca2d.ovl.y.y] [ca2d.ovl.y.x];
    int unsigned ovl_ax [ca2d.ovl.x.y] [ca2d.ovl.x.x];
    int unsigned ngb_a  [ca2d.ngb.y  ] [ca2d.ngb.x  ];

    for (int unsigned ngb=0; ngb<ca2d.ngb.n; ngb++) {
        // neighborhood integer is converted into array
        ca2d_array_from_ui (ca2d.sts, ca2d.ngb, ngb_a, ngb);
        for (int unsigned d=0; d<2; d++) {
            ca2d_array_slice (ca2d.ngb, (ca2d_size_t) {d, 0}, ca2d.ovl.y, ngb_a, ovl_ay);
            ca2d_array_to_ui (ca2d.sts, ca2d.ovl.y, ovl_ay, &n2o_y [d] [ngb]);
        }
        for (int unsigned d=0; d<2; d++) {
            ca2d_array_slice (ca2d.ngb, (ca2d_size_t) {0, d}, ca2d.ovl.x, ngb_a, ovl_ax);
            ca2d_array_to_ui (ca2d.sts, ca2d.ovl.x, ovl_ax, &n2o_x [d] [ngb]);
        }
    }
    return (0);
}

// tables for geting neighborhood values from overlap values and rest
int ca2d_net_table_o2n (ca2d_t ca2d, int unsigned o2n_y [2] [ca2d.ovl.y.n] [ca2d.rem.x.n],
                                     int unsigned o2n_x [2] [ca2d.ovl.x.n] [ca2d.rem.y.n]) {
    int unsigned ovl_ay [ca2d.ovl.y.y] [ca2d.ovl.y.x];
    int unsigned ovl_ax [ca2d.ovl.x.y] [ca2d.ovl.x.x];
    int unsigned rem_ay [ca2d.rem.y.y] [ca2d.rem.y.x];
    int unsigned rem_ax [ca2d.rem.x.y] [ca2d.rem.x.x];
    int unsigned ngb_a  [ca2d.ngb.y  ] [ca2d.ngb.x  ];
    int unsigned ovl_y;
    int unsigned ovl_x;
    int unsigned rem_y;
    int unsigned rem_x;

    // for every neighborhood integer
    for (int unsigned ngb=0; ngb<ca2d.ngb.n; ngb++) {
        // neighborhood integer is converted into array
        ca2d_array_from_ui (ca2d.sts, ca2d.ngb, ngb_a, ngb);
        for (int unsigned d=0; d<2; d++) {
            // neighborhood array is split into overlay and reminder arrays
```

```
            if (!d) {
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0           , 0}, ca2d.ovl.y, ngb_a, ovl_ay);
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {ca2d.ovl.y.y, 0}, ca2d.rem.y, ngb_a, rem_ay);
            } else {
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0           , 0}, ca2d.rem.y, ngb_a, rem_ay);
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {ca2d.rem.y.y, 0}, ca2d.ovl.y, ngb_a, ovl_ay);
            }
            // overlay and reminder arrays are converted into integers
            ca2d_array_to_ui (ca2d.sts, ca2d.ovl.y, ovl_ay, &ovl_y);
            ca2d_array_to_ui (ca2d.sts, ca2d.rem.y, rem_ay, &rem_y);
            // table is pupulated
            o2n_y [d] [ovl_y] [rem_y] = ngb;
        }
        for (int unsigned d=0; d<2; d++) {
            // neighborhood array is split into overlay and reminder arrays
            if (!d) {
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, 0           }, ca2d.ovl.x, ngb_a, ovl_ax);
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, ca2d.ovl.x.x}, ca2d.rem.x, ngb_a, rem_ax);
            } else {
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, 0           }, ca2d.rem.x, ngb_a, rem_ax);
                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, ca2d.rem.x.x}, ca2d.ovl.x, ngb_a, ovl_ax);
            }
            ca2d_array_to_ui (ca2d.sts, ca2d.ovl.x, ovl_ax, &ovl_x);
            ca2d_array_to_ui (ca2d.sts, ca2d.rem.x, rem_ax, &rem_x);
            o2n_x [d] [ovl_x] [rem_x] = ngb;
        }
    }
    return (0);
}

// function for geting array of overlap integers from edge integer in X dimension
int ca2d_net_ex2o (ca2d_t ca2d, size_t siz, int unsigned e, int unsigned o [siz]) {
    ca2d_size_t se = {ca2d.ngb.y-1, ca2d.ngb.x-1 + siz};
    ca2d_size_t so = {ca2d.ngb.y-1, ca2d.ngb.x  };
    int unsigned ae [se.y] [se.x];
    int unsigned ao [so.y] [so.x];
    ca2d_array_from_ui (ca2d.sts, se, ae, e);
    for (int unsigned x=0; x<siz; x++) {
        ca2d_array_slice (se, (ca2d_size_t) {0, x}, so, ae, ao);
        ca2d_array_to_ui (ca2d.sts, so, ao, &o[x]);
    }
    return (0);
}

// function for geting edge integer in X dimension from array of overlap integers
int ca2d_net_o2ex (ca2d_t ca2d, size_t siz, int unsigned *e, int unsigned o [siz]) {
    ca2d_size_t se = {ca2d.ngb.y-1, ca2d.ngb.x-1 + siz};
    ca2d_size_t so = {ca2d.ngb.y-1, ca2d.ngb.x  };
    ca2d_size_t sr = {ca2d.ngb.y-1,            1};
    int unsigned ae [se.y] [se.x];
    int unsigned ao [so.y] [so.x];
    int unsigned ar [sr.y] [sr.x];
    // set first overlap
    ca2d_array_from_ui (ca2d.sts, so, ao, o[0]);
    ca2d_array_fit (se, (ca2d_size_t) {0, 0}, so, ae, ao);
    // apprend remaining remainders
    for (int unsigned x=0; x<siz; x++) {
        ca2d_array_from_ui (ca2d.sts, so, ao, o[x]);
        ca2d_array_slice (so, (ca2d_size_t) {0,   ca2d.ngb.x-1}, sr, ao, ar);
        ca2d_array_fit   (se, (ca2d_size_t) {0, x+ca2d.ngb.x-1}, sr, ae, ar);
    }
    ca2d_array_to_ui (ca2d.sts, se, ae, &(*e));
    return (0);
}

static int ca1d_net (
    // CA properties
    ca2d_t ca2d,
    size_t siz,
    // configuration
    int unsigned ca [siz],
    // direction in Y dimmension
    int unsigned dy,
    int unsigned edg_i,
    mpz_t edg_w,
    int unsigned *edg_n,
    mpz_t edg_o [(size_t) pow (ca2d.sts, (ca2d.ngb.y-1)*((ca2d.ngb.x-1)+siz))]
) {
    // tables
    int unsigned n2o_y [2] [ca2d.ngb.n];
    int unsigned n2o_x [2] [ca2d.ngb.n];
    int unsigned o2n_y [2] [ca2d.ovl.y.n] [ca2d.rem.y.n];
    int unsigned o2n_x [2] [ca2d.ovl.x.n] [ca2d.rem.x.n];
    ca2d_net_table_n2o (ca2d, n2o_y, n2o_x);
    ca2d_net_table_o2n (ca2d, o2n_y, o2n_x);

    // memory allocation for preimage network
    int unsigned net [siz] [ca2d.ovl.y.n];

    // convert input edge into array of overlaps
    int unsigned ovl_i [siz];
    ca2d_net_ex2o (ca2d, siz, edg_i, ovl_i);

    int unsigned ngb_a  [ca2d.ngb.y  ] [ca2d.ngb.x  ];
    // create unprocessed 1D preimage network for the current edge
    for (int unsigned x=0; x<siz; x++) {
        // initialize network weights to zero
        for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
            net [x] [ovl] = 0;
```

```
        }
        // get segment x from current edge
        int unsigned o = ovl_i [x];
        // for all neighborhoods with the curent edge check them against the table
        for (int unsigned rem=0; rem<ca2d.rem.x.n; rem++) {
            // combine overlap and remainder into neighborhood
            int unsigned ngb = o2n_y [dy] [o] [rem];
            // check neighborhood against the rule
            if (ca[x] == ca2d.tab[ngb]) {
                // get end edge overlap from pointer table
                int unsigned ovl = n2o_y [1-dy] [ngb];
                // set weight to overlap
                net [x] [ovl] = 1;
            }
        }
    }

    // count 1D network preimages
    int unsigned v2o_y [2] [ca2d.ovl.y.n] [ca2d.shf.y.n];
    int unsigned v2o_x [2] [ca2d.ovl.x.n] [ca2d.shf.x.n];

    ca2d_net_table_v2o (ca2d, v2o_y, v2o_x);

    for (int x=1; x<siz; x++) {
        for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
            // first check if the path is available
            if (net [x] [ovl]) {
                int unsigned sum = 0;
                for (int unsigned shf=0; shf<ca2d.shf.y.n; shf++) {
                    sum += net [x-1] [v2o_y[0][ovl][shf]];
                }
                net [x] [ovl] = sum;
            }
        }
    }

    // calculate preimage number
    *edg_n = 0;
    for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
        *edg_n += net [siz-1] [ovl];
    }
    // allocate memory for preimage list
    int unsigned lst [*edg_n] [siz];

    // initialize list of 1D network preimages
    int unsigned p = 0;
    for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
        for (int unsigned i=0; i<net[siz-1][ovl]; i++) {
            lst [p] [siz-1] = ovl;
            p++;
        }
    }
    // list 1D network preimages
    for (int x=siz-2; x>=0; x--) {
        int unsigned p = 0;
        while (p < *edg_n) {
            int unsigned ovl = lst [p] [x+1];
            for (int unsigned shf=0; shf<ca2d.shf.y.n; shf++) {
                int unsigned o = v2o_y[0][ovl][shf];
                for (int unsigned i=0; i<net[x][o]; i++) {
                    lst [p] [x] = o;
                    p++;
                }
            }
        }
    }

    // put preimages into edge list
    for (int unsigned i=0; i<*edg_n; i++) {
        int unsigned edg;
        ca2d_net_o2ex (ca2d, siz, &edg, lst [i]);
        mpz_add (edg_o [edg], edg_o [edg], edg_w);
    }

    return (0);
}

int ca2d_net (ca2d_t ca2d, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x], mpz_t cnt [2], int
    unsigned (** p_list) [] [siz.y+ca2d.ver.y] [siz.x+ca2d.ver.x]) {
    // edge size
    const int unsigned edg_x  = pow (ca2d.sts,  (ca2d.ngb.y-1)       *((ca2d.ngb.x-1)+siz.x));
    const int unsigned edg_y  = pow (ca2d.sts, ((ca2d.ngb.y-1)+siz.y)* (ca2d.ngb.x-1)       );

    // compact list of edges
    int unsigned edg_n;

    // memory allocation for preimage network
    mpz_t net [2] [siz.y+1] [edg_x];

    // initialize array variable
    for (int d=0; d<2; d++) {
        for (int y=0; y<=siz.y; y++) {
            for (int unsigned edg=0; edg<edg_x; edg++) {
                mpz_init (net [d] [y] [edg]);
            }
        }
    }
    // initialize starting edge to unit (open edge)
    for (int unsigned edg=0; edg<edg_x; edg++) {
```

```
        mpz_set_ui (net [0] [0    ] [edg], 1);
        mpz_set_ui (net [1] [siz.y] [edg], 1);
    }
    // compute network weights in both directions
    for (int y=0; y<siz.y; y++) {
        // loop over all edges
        for (int unsigned edg=0; edg<edg_x; edg++) {
            // only process edge if it's weight is not zero
            if (mpz_sgn (net [0] [      y] [edg]) > 0) {
                ca1d_net(ca2d, siz.x, ca [        y], 0, edg, net [0] [      y] [edg], &edg_n, net [0]
[      (y+1)]);
            }
            if (mpz_sgn (net [1] [siz.y-y] [edg]) > 0) {
                ca1d_net(ca2d, siz.x, ca [siz.y-1-y], 1, edg, net [1] [siz.y-y] [edg], &edg_n, net [1]
[siz.y-(y+1)]);
            }
        }
    }
    // count all preimages
    for (int unsigned d=0; d<2; d++) {
        mpz_init   (cnt [d]);
        for (int unsigned edg=0; edg<edg_x; edg++) {
            mpz_add (cnt [d], cnt [d], net [d] [d ? 0 : siz.y] [edg]);
        }
    }

    // allocate memory for preimages described by edges
    mpz_t weight;
    mpz_init (weight);
    int unsigned lst [mpz_get_ui(cnt[0])] [siz.y+1];

    // initialize list of 2D network preimages
    int unsigned p = 0;
    for (int unsigned edg=0; edg<edg_x; edg++) {
        int unsigned max;
        max = mpz_get_ui(net[1][0][edg]);
        for (int unsigned i=0; i<max; i++) {
            lst [p] [0] = edg;
            p++;
        }
    }
    // list 2D network preimages
    // memory allocation for preimage network
    mpz_t edges [edg_x];
    // initialize array variable
    for (int unsigned edg=0; edg<edg_x; edg++) {
        mpz_init (edges [edg]);
    }
    for (int y=1; y<=siz.y; y++) {
        int unsigned p = 0;
        while (p < mpz_get_ui(cnt[0])) {
            // gain process 1D preimage for current edge (lst [p] [y])
            mpz_set_ui (weight, 1);
            // re initialize edges
            for (int unsigned edg=0; edg<edg_x; edg++) {
                mpz_set_ui (edges [edg], 0);
            }
            ca1d_net (ca2d, siz.x, ca [y-1], 0, lst [p] [y-1], weight, &edg_n, edges);
            for (int unsigned edg=0; edg<edg_x; edg++) {
                mpz_set (weight, edges[edg]);
                if ((mpz_sgn (weight) > 0) && (mpz_sgn (net[1][y][edg]) > 0)) {
                    for (int unsigned i=0; i<mpz_get_ui(net[1][y][edg]); i++) {
                        lst [p] [y] = edg;
                        p++;
                    }
                }
            }
        }
    }

    // allocate memory for preimages
    ca2d_size_t siz_pre = {siz.y+ca2d.ver.y, siz.x+ca2d.ver.x};
    siz_pre.a = siz_pre.y * siz_pre.x;
    *p_list = (int unsigned (*) [] [siz_pre.y] [siz_pre.x]) malloc (sizeof(int unsigned) * siz_pre.a *
    mpz_get_ui(cnt[0]));
    // convert edge list into actual preimage list
    for (int unsigned i=0; i<mpz_get_ui(cnt[0]); i++) {
        ca2d_size_t siz_lin0 = (ca2d_size_t) {ca2d.ver.y, siz.x+ca2d.ver.x};
        int unsigned line0 [siz_lin0.y] [siz_lin0.x];
        ca2d_array_from_ui (ca2d.sts, siz_lin0, line0, lst[i][0]);
        ca2d_array_fit (siz_pre, (ca2d_size_t) {0, 0}, siz_lin0, (**p_list)[i], line0);
        for (int unsigned y=0; y<siz.y; y++) {
            ca2d_size_t siz_lin = (ca2d_size_t) {1, siz.x+ca2d.ver.x};
            int unsigned line [siz_lin.y] [siz_lin.x];
            ca2d_array_from_ui (ca2d.sts, siz_lin0, line0, lst[i][y+1]);
            ca2d_array_slice   (siz_lin0, (ca2d_size_t) {ca2d.ver.y-1, 0}, siz_lin, line0, line);
            ca2d_array_fit (siz_pre, (ca2d_size_t) {y+ca2d.ver.y, 0}, siz_lin, (**p_list)[i], line);
        }
    }

    printf ("DEBUG: end of network\n");
    return (0);
}
```

Slike

Literatura

[1] Conway's Reverse Game of Life. https://www.kaggle.com/c/conway-s-reverse-game-of-life, Marec 2013.

[2] LifeWiki: Flower of Eden. http://www.conwaylife.com/wiki/Flower_of_Eden, Avgust 2016.

[3] The On-Line Encyclopedia of Integer Sequences: A196447. https://oeis.org/A196447, Avgust 2016.

[4] Wikipedia: Backtracking. https://en.wikipedia.org/wiki/Backtracking, Avgust 2016.

[5] Wikipedia: Boolean satisfiability problem. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem, Avgust 2016.

[6] Wikipedia: Brute-force search. https://en.wikipedia.org/wiki/Brute-force_search, Avgust 2016.

[7] Wikipedia: Conway's Game of Life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life, Avgust 2016.

[8] Wikipedia: De Bruijn graph. https://en.wikipedia.org/wiki/De_Bruijn_graph, Avgust 2016.

[9] Wikipedia: Depth-first search. https://en.wikipedia.org/wiki/Depth-first_search, Avgust 2016.

[10] Wikipedia: Garden of Eden (cellular automaton). https://en.wikipedia.org/wiki/Garden_of_Eden_(cellular_automaton), Avgust 2016.

[11] Wikipedia: Langton's loops. https://en.wikipedia.org/wiki/Langton%27s_loops, Avgust 2016.

[12] Wikipedia: Rule 110. https://en.wikipedia.org/wiki/Rule_110, Avgust 2016.

[13] Wikipedia: Von Neumann universal constructor. https://en.wikipedia.org/wiki/Von_Neumann_universal_constructor, Avgust 2016.

[14] Vladimir Batagelj. Efficient Algorithms for Citation Network Analysis. *CoRR*, cs.DL/0309023, 2003. Dostopno na http://arxiv.org/abs/cs.DL/0309023.

[15] Neil Bickford. Reversing the Game of Life for Fun and Profit. https://nbickford.wordpress.com/2012/04/15/reversing-the-game-of-life-for-fun-and-profit/, April 2012.

[16] Peter Borah. Atabot. https://github.com/PeterBorah/atabot, 2013.

[17] Bryan Duxbury. Cellular Chronometer. https://github.com/bryanduxbury/cellular_chronometer/tree/master/rb, September 2013. Članek: https://bryanduxbury.com/2013/09/02/cellular-chronometer-part-1-reversing-the-game-of-life/.

[18] Yossi Elran. Retrolife and The Pawns Neighbors. *The College Mathematics Journal*, 43(2):147–151, 2012. Dostopno na http://www.jstor.org/stable/10.4169/college.math.j.43.2.147.

[19] Achim Flammenkamp. Garden of Eden / Orphan. http://wwwhomes.uni-bielefeld.de/achim/orphan.html, Avgust 2016.

[20] Dave Greene. New Technology from the Replicator Project. http://b3s23life.blogspot.si/2013/11/new-technology-from-replicator-project.html, November 2013. Dostop: 2-avg-2016.

[21] Jean Hardouin-Duparc. À la recherche du paradis perdu. *Publ. Math. Univ. Bordeaux Année*, 4:51–89, 1972-1973.

[22] Jean Hardouin-Duparc. Paradis terrestre dans l'automate cellulaire de Conway. *Rev. Française Automat. Informat. Recherche Operationnelle Ser. Rouge*, 8:64–71, 1974. Dostopno na http://archive.numdam.org/ARCHIVE/ITA/ITA_1974__8_3/ITA_1974__8_3_63_0/ITA_1974__8_3_63_0.pdf.

[23] Christiaan Hartman, Marijn J. H. Heule, Kees Kwekkeboom, in Alain Noels. Symmetry in Gardens of Eden. *Electronic Journal of Combinatorics*, 20, 2013.

[24] ionreq. calculate previous generations for Conway's Game of Life. https://github.com/ionreq/GoLreverse, 2013. Video prispevek: https://www.youtube.com/watch?v=ZOuMnaarI5k.

[25] Iztok Jeras. Solving cellular automata problems with SAGE/Python. Objavljeno v Andrew Adamatzky, Ramón Alonso-Sanz, Anna T. Lawniczak, Genaro Juárez Martínez, Kenichi Morita, in Thomas Worsch, editors, *Automata 2008: Theory and Applications of Cellular Automata, Bristol, UK, June 12-14, 2008*, strani 417–424. Luniver Press, Frome, UK, 2008. Dostopno na http://uncomp.uwe.ac.uk/free-books/automata2008reducedsize.pdf.

[26] Iztok Jeras. 2D cellular automata preimages count&list algorithm. https://github.com/jeras/preimages-2D, 2016.

[27] Iztok Jeras. Cellular Automata SAGE Toolkit. https://github.com/jeras/cellular-automata-sage-toolkit, 2016.

[28] Iztok Jeras. WebGL QUAD CA sumulator. https://github.com/jeras/webgl-quad-ca, 2016.

[29] Iztok Jeras in Andrej Dobnikar. Cellular Automata Preimages: Count and List Algorithm. Objavljeno v Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, in Jack Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part III*, del 3993 of *Lecture Notes in Computer Science*, strani 345–352. Springer, 2006. Dostopno na `http://dx.doi.org/10.1007/11758532_47`.

[30] Iztok Jeras in Andrej Dobnikar. Algorithms for computing preimages of cellular automata configurations. *Physica D Nonlinear Phenomena*, 233:95–111, September 2007.

[31] Jarkko Kari. Reversibility of 2D cellular automata is undecidable. *Physica D: Nonlinear Phenomena*, 45:379–385, 1990.

[32] Paulina A. Léon in Genaro J. Martínez. Describing Complex Dynamics in Life-Like Rules with de Bruijn Diagrams on Complex and Chaotic Cellular Automata. *Journal of Cellular Automata*, 11(1):91–112, 2016.

[33] Harold V. McIntosh. Linear Cellular Automata via de Bruijn Diagrams. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.8763&rep=rep1&type=pdf`, Avgust 1991.

[34] E. F. Moore. Machine models of self-reproduction. Objavljeno v *Mathematical Problems in Biological Sciences (Proceedings of Symposia in Applied Mathematics)*, del 14, strani 17–33. American Mathematical Society, 1962.

[35] John Myhill. The converse of Moore's Garden-of-Eden theorem. Objavljeno v *Proceedings of the American Mathematical Society*, del 14, strani 685–686. American Mathematical Society, 1963.

[36] Florian Pigorsch. A SAT-based forward/backwards solver for Conway's "Game of Life". `https://github.com/flopp/gol-sat`, Oktober 2015.

[37] Edward Powley. QUAD Prize Submission: Simulating Elementary CAs with Trid CAs. *Journal of Cellular Automata*, 5:415–417, 2010. Dostopno na `http://uncomp.uwe.ac.uk/automata2008/files/quadprize_powley.pdf`.

[38] Paul Rendell. A Turing Machine In Conway's Game Life. `https://www.ics.uci.edu/~welling/teaching/271fall09/Turing-Machine-Life.pdf`, Avgust 2001. Dostop: 2-avg-2016.

[39] Chris Salzberg in Hiroki Sayama. Complex genetic evolution of artificial self-replicators in cellular automata. *Complexity*, 10:33–39, 2004. Dostopno na `http://www3.interscience.wiley.com/journal/109860047/abstract`.

[40] José Manuel Gómez Soto. Computation of Explicit Preimages in One-Dimensional Cellular Automata Applying the De Bruijn Diagram. *Journal of Cellular Automata*, 3:219–230, 2008.

[41] Tommaso Toffoli. Background for the Quad Prize. `http://uncomp.uwe.ac.uk/automata2008/files/quad.pdf`, Februar 2008.

[42] Erik P. Verlinde. On the origin of gravity and the laws of Newton. 2010.

[43] Robert Wainwright. Lifeline newsletter - volume 3. http://www.conwaylife.com/w/index.php?title=Lifeline_Volume_3, September 1971.

[44] Robert Wainwright. Lifeline newsletter - volume 4. http://www.conwaylife.com/w/index.php?title=Lifeline_Volume_4, December 1971.

[45] Christopher Wellons. WebGL Game of Life. https://github.com/jeras/webgl-quad-ca, 2014.

[46] Andrew Wuensche. Discrete Dynamics Lab. https://www.ddlab.com, 2016.

[47] Andrew Wuensche in Mike Lesser. *The Global Dynamics of Cellular Automata*. Santa Fe Institute, 1992. Dostopno na http://uncomp.uwe.ac.uk/wuensche/gdca.html.