Contents

[Domáca úloha 1 2](#_Toc194847445)

[Príklad vstup/výstup: 4](#_Toc194847446)

[Domáca úloha 2 5](#_Toc194847447)

[Otázky a odpovede 7](#_Toc194847448)

[Otázky 7](#_Toc194847449)

[Odpovede 7](#_Toc194847450)

[Domáca Úloha 3 9](#_Toc194847451)

[Výsledná verzia algoritmu 11](#_Toc194847452)

[Hill climb 12](#_Toc194847453)

[Tabu Search 13](#_Toc194847454)

[Simulated Annealing 13](#_Toc194847455)

# Domáca úloha 1

O game of life som už počul, doteraz som ale nechápal ako je to spravené.  
Keď som bol malý a začínal s PC tak popri Powder Toy bolo Conway’s game of life – prišlo mi to ako zázrak, ako môže niekto simulovať život? Ale to bola naivná predstava 10 ročného ja.

Začal som s počiatočným kódom od LLM, pretože python som už dlho nepísal:

|  |
| --- |
| def step(row):  new\_row = [0] \* len(row) # Create a new row of the same size, initialized with 0s    for i in range(len(row)):  left = row[i - 1] if i > 0 else 0 # Left neighbor (0 if at the beginning)  right = row[i + 1] if i < len(row) - 1 else 0 # Right neighbor (0 if at the end)    # Rule: If a cell has exactly 1 neighbor alive, it becomes alive  if row[i] == 1 and (left == 1 or right == 1):  new\_row[i] = 1  elif row[i] == 0 and (left + right == 1):  new\_row[i] = 1  return new\_row  def printrow(row):  output = ''.join(['█' if cell == 1 else ' ' for cell in row]) # Use █ for alive, space for dead  print(output)  row = [0, 1, 0, 1, 1, 0, 0, 1]  printrow(row) # It will print a row of spaces and █ characters |

Spustil som program a dostal som tento výstup:  
█ ███ █ - toto sa mi nepáčilo, keďže mi to príde trocha neintuitívne, tak som to prepísal:

|  |
| --- |
| def step(row):  new\_row = [0] \* len(row) # Create a new row of the same size, initialized with 0s    for i in range(len(row)):  left = row[i - 1] if i > 0 else 0 # Left neighbor (0 if at the beginning)  right = row[i + 1] if i < len(row) - 1 else 0 # Right neighbor (0 if at the end)    # Rule: If a cell has exactly 1 neighbor alive, it becomes alive  if row[i] == 1 and (left == 1 or right == 1):  new\_row[i] = 1  elif row[i] == 0 and (left + right == 1):  new\_row[i] = 1  return new\_row |

Následne som si preštudoval syntax, a uvedomil že definícia left a right je vlastne skvelá, pretože nebudem musieť robiť mentálnu aritmetiku na poliach.

Pridal som pravidlo – ak má bunka dvoch živých susdeov tak zomrie (teraz už ja, nie LLM :d).

|  |
| --- |
| # pravidlo: ak ma bunka dvoch zivych susedov, zomrie  if row[i] == 1 and (left == 1 and right == 1):  new\_row[i] = 0 |

Následne som napísal prvotnú podobu funkcie printRow, ktorá vypísala živé bunky ako ‚Y‘ a mŕtve ako ‚x‘.

|  |
| --- |
| def printrow(row):  output = str(genNumber) + ' ' + ''.join(['Y' if cell == 1 else 'x' for cell in row])  row = [1, 0, 0, 1, 1, 0, 0, 0]  printrow(step(row)) # spusti algoritmus |

Potom som pridal iterovanie voči generáciám.

|  |
| --- |
| def printrow(row):  global generations  global genNumber  for \_ in range(generations): # opakuj dany pocet generacii  output = str(genNumber) + ' ' + ''.join(['Y' if cell == 1 else 'x' for cell in row])  print(output)    row = step(row) # vygeneruj dalsiu generaciu  genNumber += 1 # zvys pocet generacii  printrow(row) # spusti algoritmus |

Zároveň aj premennú genNumber aby som mal jednoducho a čítateľne prehľad o tom, ktorý riadok je ktorá generácia.

## Príklad vstup/výstup:

Konkrétne sa jednalo o 5 generácií:

|  |
| --- |
| Input |
| generations = 5 # pocet generacii na vykonanie  row = [1, 0, 0, 1, 1, 0, 0, 0] # vstupny array prvkov |

|  |
| --- |
| Output |
| 1 YxxYYxxx  2 xYYYYYxx  3 YYxxxYYx  4 YYYxYYYY  5 YxYxYxxY |

Je tam veľmi pekne vidieť, že ak sú tri ‚Y‘ vedľa seba, tak zanikne presne člen v strede, zároveň špeciálny prípad 5x Y vedľa seba, stredné členy vymreli.

|  |
| --- |
| Input |
| generations = 5 # pocet generacii na vykonanie row = [0, 0, 0, 0, 0, 0, 0, 0]  # vstupny array prvkov |
| Output |
| 1 xxxxxxxx 2 xxxxxxxx 3 xxxxxxxx 4 xxxxxxxx 5 xxxxxxxx |
| Input |
| generations = 5 # pocet generacii na vykonanie row = [1, 1, 1, 1, 1, 1, 1, 1]  # vstupny array prvkov |
| Output |
| 1 YYYYYYYY 2 YxxxxxxY 3 xYxxxxYx 4 YxYxxYxY 5 xxxYYxxx |

# Domáca úloha 2

Našiel som schelling.pdf cez fulltextsearch v bookkite učebnice, a prečítal som si ho.

Čiže potrebujem:

* Vytvoriť tabuľku/matrix, ktorá reprezentuje „UV mapu“ torusu
* Čiže, ak chcem aby *agent* bol *šťastný*, tak v jeho okruhu (v tomto prípade v tabuľke je to 3x3 s agentom v strede musí byť pomer agentov jeho farby väčší ako jeho tolerancia
* Mám pár otázok:
  + Ako chcem robiť toleranciu, na jedno pole alebo všade rovnaká?
    - Odpoveď: Skúsim najprv všade rovnakú
  + Akú veľkú tabuľku urobiť?
    - Ešte neviem ani kresliť v pythone, skúsim najprv 5x5

No o 15 minút neskôr som trochu zabojoval s grafom a mám toto:

A yellow and blue squares

AI-generated content may be incorrect.

Chcel som sa dozvedieť viac o Schellingovom modeli a narazil som na [toto video.](https://www.youtube.com/watch?v=JjfihtGefxk)

Vidím že mi treba oveľa viac buniek ako 5x5 aby som videl „segregáciu“, tak urobím 25x25 maticu. Mením deklaráciu z

|  |  |  |
| --- | --- | --- |
| rep = [  ['E', 'E', 'E', 'E', 'E'],  ['E', 'E', 'E', 'E', 'E'],  ['E', 'E', 'Y', 'E', 'E'],  ['E', 'E', 'E', 'B', 'E'],  ['E', 'E', 'E', 'E', 'E']  ] | na | matrix = [['E' for \_ in range(25)] for \_ in range(25)] |

Mal som problémy s jupyter notebook tak prechádzam na VSCode ako editor.

Hm, ale mám prázdne pole, skúsim nejako urobiť pseudo-náhodnú generáciu žltých a modrých členov.

Snažil som sa vytvoriť funkciu pomocou času, ale kód sa spracúva príliš rýchlo:

Trvalo mi to cca 15 pokusov, dokým som dostal toto:

|  |  |
| --- | --- |
| def whatColor():      cur\_time = time.time()      ms = int(cur\_time \* 1000) % 10      if ms < 3:          return 'E' #biela      elif ms < 6:          return 'Y' #zlta      else:          return 'B' #modra |  |

Back to the drawing board:

rep = [[random.choice(['E','Y','B']) for \_ in range(25)] for \_ in range(25)]

Tak tentokrát to už vyzerá dobre, teraz idem zaimplementovať schellingov model.A blue and yellow squares

AI-generated content may be incorrect.

Okej, trvalo mi to asi 1,5h ale nakoniec som vymyslel checkovanie satisfaction tak, aby kontrolovalo vsetkych 9 clenov, ale ak je out of bounds tak proste tam doplni E ako prazdne.

Ok, tam kde je prazdne je satisfakcia 1

Zaujimavy vysledok pocas debugovania:

Počas behov som narazil na to, že ak mám funkcie step(rep) v jednej funkcii tak je výpočet do 10 sekúnd a ak ich rozdelím na funkciu, ktorá berie tieto údaje z step(rep) a iteruje cez nich tak 10+ minut potom som to vzdal.A yellow and blue square

AI-generated content may be incorrect.

Zistil som že som mal zle poskladané funkcie a prepočítavalo matice spokojnosti viackrát, aj keď nemuselo. Optimalizoval som kód dokým beh na matici 25x25 netrval menej ako minútu, nechce sa mi dlho čakať. Taktiež som nahradil inkrementovanie premien funkciami sum() na miestach, kde sa to dalo.

Celé som to prepísal, boli tam funkcie, ktoré som vôbec nepoužíval ako kontrola rohov.

## Otázky a odpovede

### Otázky

1. Aký je vplyv parametru **tol?**
2. Vývoj N° spokojných jednicov v čase?
3. U jakého **tol** nastáva segregácia?
4. Vplyv veľkosti okolia?
5. Možné rozšírenia modelu?

### Odpovede

1: Parameter **tol** ovplyvňuje zgrupovanie agentov, t.j. ako veľmi im bude vadiť, že majú suseda inej farby. Čím väčší **tol**, tým viac im to vadí.   
V prípade môjho algoritmu je to pomer:

Kde sa porovnáva **tol** voči **satisfaction.**

2: Vývoj počtu spokjných jedincov pri tolerancii 0.5 to vyzerá približne ako funkcia

A screenshot of a graph

AI-generated content may be incorrect.

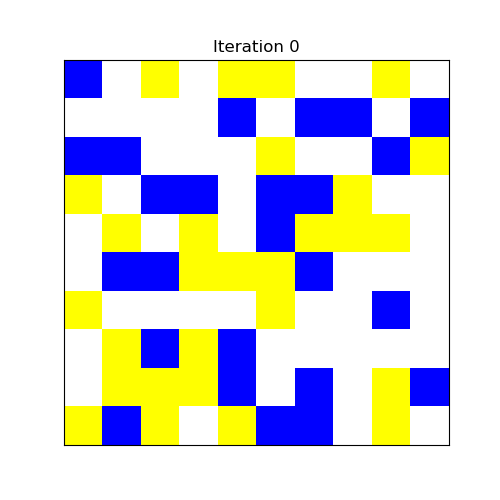
Keď som dal model nad toleranciu 0.5 tak mi nenašlo výsledok (infinite loop)

3: Segregácia u **tol** nastáva už okolo 0.3, kedy vidieť že sa bunky dávajú do skupín.

A graph of a graph

AI-generated content may be incorrect.

4: Vplyv väčšieho prostredia: Pri 2x viac prázdneho miesta sa mi dialo toto: (gif v pdf nefunguje, je priložený v zipku)



Následne ak som dal 3x väčšie prostredie a väčší počiatočný matrix tak všetky bunky boli „šťastné“.

**Čím väčšie okolie, resp. čím viac voľného miesta, tak tým „šťastnejšie“ bunky.**

5: Možné rozšírenie modelu by som si predstavoval, najmä nad **tol** 0.5, že bunky budú dohľadávať väčšie skupiny, a pripojovať sa k ním,  
alebo jednoduchšie nechodiť znova na prázdne miesto, kde už bunka bola

# Domáca Úloha 3

Počiatočne som začal s kódom, ktorý som našiel na internete, aby som mal predstavu.

Algoritmus bežal okolo 80min a našiel ofarbenie grafu [dsjc125.9](http://cedric.cnam.fr/~porumbed/graphs/dsjc125.9.col) na 50 farieb, s čím som nebol spokojný.

Vlastný algoritmus: moja prvá idea bola, že náhodne ofarbím všetky polia a potom postupne budem meniť farby, dokým graf nebude správne ofarbený (t.j. všetko random walk) ale po znovu-zamyslení sa som si uvedomil že to je nereálne (všetko cez random = extrémne dlhý čas na výpočet + nemusí vždy fungovať).

Môj nový nápad pozostáva z toho, že budem využívať **náhodné ofarbenie,** následne **hill climb** a ak potom uviazne tak **random walk**.

Pre prácu s grafom využívam **networkx**, na zistenie ako funguje syntax pre prácu s knižnicou som si vypísal toto:

|  |
| --- |
| # zobrazenie bodov a hran  print("Nodes:", list(G.nodes())) # body  print("Edges:", list(G.edges())) # hrany  # struktura grafu  print("\nGraph as adjacency list:")  for node in G.nodes():  neighbors = list(G.neighbors(node))  print(f"{node}: {neighbors}") |

No, cez ten hill climb to bolo dosť dlho čakať na výsledok, a vždy mi to nenašlo ten počet farieb aký chcem. Skúšal som to s trochu „ľahším“ grafom: [dsjc125.1.col](https://cedric.cnam.fr/~porumbed/graphs/dsjc125.1.col)  
aby som nemusel dlho čakať. Tento graf má mať 5 farieb, ale našlo mi to najlepšie 8.

Vymyslel som to následovne:

* Hill climb ako main vec
  + Kontrola či sa zmenší počet konfliktov
    - Ak áno, super, našli sme novú farbu
    - Ak nie, vráť originálnu
* Random walk každých 10 steps
  + Vyber random node a meň mu nahodne k-krat farbu
    - Každý pokus kontrola či sa zmenší počet konfliktov
      * Ak áno, super
      * Ak nie, vráť originálnu

Ak algo prejde 10k steps, tak mi navýši počet pokusov a farbu. Počet max pokusov som dal na 10. Spustil som to na grafe zo zadania, som zvedavý ako to dopadne (idem sa prejsť :D).

Po 856 minútach som beh grafu vypol, nechcem si brať algoritmus so sebou do hrobu.

Po súperení s mojou naivitou a hlúposťou som sa úplne rozhodol vypustiť chatbotov a robiť to sám, keďže aj mozog potrebuje niekedy námahu a nechcem skončiť tak, že mi AIčko bude hovoriť všetko čo mám robiť.

Po chvíľke rozmýšľania som vymyslel jednoduchý algoritmus, ktorý mi pomohol sa dostať k celkom zaujímavému výsledku, ale má pár problémov, ktoré chcem vyriešiť cez hill climb algoritmus.

Tento algoritmus spočíva v tom, že:

* Navštívim prvý bod, ktorý skúmam
* Nastavím mu farbu X
* Všetkým jeho susedom nastavím novú farbu Y
* Pokračujem do ďalšieho skúmaného bodu
* Nastavím mu farbu Z
* Všetkým jeho susedom...

Nafarbilo mi to graf dsjc125.9 na 55 farieb, čo je celkom zaujímavé a je extrémne rýchly.

Hlavný problém je taký, že graf nie je správne ofarbený, skúšal som na to urobiť funkciu ale po 15 minútach behu som to vzdal. (Problém bol v implementácií checkovania správnosti ofarbenia grafu, kvôli posunutým farbám od 1 nie od 0)

Vytváram hill climb algoritmus, ktorý:

* Zavolá funkciu simpleAdjacencyColoring
* Priradí z nej col, k
* Načitá target\_k
* Vyberie random číslo / farbu
* Bude na týchto miestach skúšať farby z existujúcich farieb bez predošlej vybranej farby, dokým nebude graf správne zafarbený

Vzdal som tento postup, nefungovalo to: Hill climb sa snaží zmenšiť počet konfliktov, ale môj algoritmus na simple adjacency coloring nemá žiadne konflikty.   
Namiesto toho som zlepšil svoj hill climb a zaimplementoval po 10x rovnakom výsledku stagnation counter, ktorý triggeruje beh random walk, ktorý pomáha sa dostať von z lokálneho „loopu“. Myslím že problém je v tom, že cykluje medzi rovnakými bodmi, ktorých zmena farby nepomáha sa dostať k skutočnému výsledku. Mám tam 20% šancu na ten random walk ale podľa toho čo vidím tak budem to musieť dať trochu vyššie. Prosím, tolerujte môj profesionálny debug message na beh random walk ...

A screen shot of a computer

AI-generated content may be incorrect.1 stagnácia = 10x rovnaký výsledok za sebou, čiže som mal ~31x 10x rovnaký výsledok za sebou. Neresetujem tento inkrement, preto je veľký.

Taktiež zaujímavá vec, keďže to je na pseudo-náhode, tak niekedy resp. niektorý beh algoritmu je lepší ako ten predošlý. Na screenshote to bol celkom dobrý beh, bežalo to 15min a potom som to vypol.

Najprv som skúšal každých 10 stagnacii šancu na random walk, zmenil som to na ak sa 3x opakuje rovnaké číslo 100% že urobí random walk a vyzerá to, že to funguje.

Po 30min behu som sa nedokázal dostať pod 7 konfliktov, preto sa pokúsim s niekým porozprávať a skúsiť niečo ďalšie.

Všimol som si tabu search, pokúsim sa to zaimplementovať, znie to sľubne.

Do existujúceho algoritmu som nedokázal „vopchať“ tabu search, tak som sa rozhodol program znova prepísať.

## Výsledná verzia algoritmu

**Load web graph** je funkcia, ktorá načíta getovacím requestom algoritmus.

**Count conflicts** je funkcia, ktorá spočíta konflikty v grafe. Konflikt = 2 nodes s rovnakou farbou vedľa seba.

**Get\_conflicting nodes** rovnaká ako count\_conflicts ale pridáva do setu nodes, ktoré majú rovnakú farbu a vráti ich.

**Random walk** vykoná random walk s prioritou na conflicting nodes, ak nie sú konflikty ide na náhodný node. Vyberá možné farby, ktoré nepoužívajú susedia a mení node z dostupných farieb.

**Is stagnating** je jednoduchá funkcia ktorá vráti **True** alebo **False** na základe posledných 20 výsledkov a threshold: ak je viac rovnakých výsledkov ako threshold, vráti **True.**

**Hill climbing with walks** je hill climb rozšírený o **stagnation detection** a prioritou konfliktujúcich nodes, taktiež periodicky (80% šanca) mení medzi hill climb a random walk.

**Tabu search** je funkcia, ktorá zaznamenáva predošlé výsledky v tzv. „taboo“ tabuľke a skúša nové možnosti. Riešenie je taboo práve vtedy, ak sa už vyskytuje v tabuľke, vtedy sa preskakuje. Riešenie sa prijíma práve vtedy, ak nie je taboo alebo je lepšie ako riešenie doteraz.

**Simulated annealing** je funkcia, ktorá obsahuje teplotu a cooling rate. Skúša náhodné možnosti, v tomto prípade s 90% šancou na vybratie farby, ktoré nekonfliktujú so susedom, ináč vyberá zo všetkých farieb. Rozhodnutie je prijaté na základe „teploty“.

Beh na algo dsjc125.9 s 44 farbami:

A screen shot of a computer

AI-generated content may be incorrect.

## Failed to find valid coloring. Best solution has 2 conflicts. [FAILURE]: Could not find valid coloring with 43 colors after 4804.22 seconds. Stopping search. Minimum colors found: 44

**Beh trval 144m 48.2s**

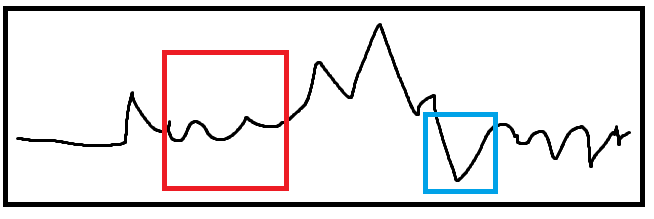
## Hill climb

Hill climb je algoritmus, ktorý sa snaží optimalizovať riešenie tým, že vyberá náhodné body a ich susedov, mení ich a ak sa riešenie (v tomto prípade zníženie počtu konfliktov alebo kolízií) stane lepším, tak ho nechá, ak nie tak skúša ďalej.

Výhody: Celkom ok rýchlosť, pri jednoduchých riešeniach funguje super

Nevýhody: Závislosť na náhode, všimol som si že ak spúšťam algoritmus na domácej úlohe, tak sa vie „zaseknúť“ a nepokračuje už ďalej. Pri zdĺhavých riešeniach trvá exponenciálne dlhšie.

Do svojej úlohy som zaimplementoval detekciu stagnácie, a následne vykonanie random walk, ktoré pomáha dostať sa z tzv. lokálneho minima. Znamená to, že ak mám napr. hľadanie najnižšej hodnoty, môže sa stať že hill climb sa bude pohybovať len v červenom štvorci, ale „nedostane sa odtamadiaľ“ lebo kvôli implementácií si myslí že ďalej sú už len vyššie hodnoty.



## Tabu Search

Tabu search je taký vylepšený hill climb - robí to, že si pamätá posledné riešenia a pridáva ich do „taboo“ tabuľky. V praxi to znamená to, že ak nájde riešenie, ktoré je už v taboo tabuľke, tak ho preskočí. Zároveň porovnáva medzi riešeniami v tejto tabuľke a vyberie najlepšie z nich (aj keď je celkovo horšie), čo pomáha dostať sa von z lokálneho minima.

## Simulated Annealing

Zdroje:

<https://en.wikipedia.org/wiki/Simulated_annealing>

<https://www.machinelearningplus.com/machine-learning/simulated-annealing-algorithm-explained-from-scratch-python/>

Tzv. simulované kalenie:

* Optimalizačný algoritmus
* Vyberie náhodný stav
* Rozhodne či vybrať výsledok na základe „teploty“ a hodnoty funkcie

Ako mám chápať túto teplotu, a funkčnú hodnotu? Môžem to jednoducho vysvetliť na svojom algoritme (aj keď ho neuvidíte).

Problém je nasledovný: Počas hľadania najlepšieho riešenia mám určitý počet konfliktov, a snažím sa ho znížiť, ale neviem ísť pod určité číslo (keďže výber je pseudonáhodný). Annealing pomáha sa dostať k výsledku tým, že pomocou teploty vyberá niekedy aj horšie riešenie tzn. escapuje local minimum.

Ak je funkčná hodnota vyššia a teplota nižšia, je väčšia šanca vybrať toto riešenie.

# delta  
delta = new\_conflicts - old\_conflicts  
# vypocet sance prijatie riesenia  
 delta <= 0 or random.random() < math.exp(-delta / max(0.01, temperature)):  
 # prijat a prepocitat konflikty  
current\_conflicts = count\_conflicts(G, current\_col)