Adam Miškove

Umelá Inteligencia

Zadanie 3: Zenová záhrada, Evolučný algoritmus
Cvičenie: Utorok 14:00-15:40, Kapustík
FIIT STU

Opis algoritmu, ktorý je použitý v programe:

Použil som algoritmus, ktorý bol načrtnutý v zadaní a na prednáške v šiestom týždni.
Toto zadanie som vypracoval v programovacom jazyku python, a IDE PyCharm.
Program začína tak, že sa spustí funkcia **initialize()**, ktorá má za úlohu inicializovať všetky potrebné premenné podľa parametrov z inputu od používateľa.

Možnosti nastavenia parametrov:

Počet riadkov a stĺpcov záhrady.
Počet génov v jednom chromozóme. Táto hodnota musí byť v intervale od 1 do (počet stĺpcov+počet riadkov + počet kameňov). V prípade príkladu zo zadania je táto maximálna hodnota 10+12+6=28.
Výška tejto hodnoty ovplyvňuje efektivitu hľadania riešenia, čo bude opísané nižšie.
Typ metódy výberu jedincov. Implementované sú dve metódy - turnaj a ruleta. Tento string musí byť presne “ruleta”, alebo “turnaj”.
Počet kameňov na záhrade.
Počet generácií.
Súradnice všetkých kameňov na záhrade.

```
for i in range(0, stoneCount):
    print("Zadajte riadkové súradnice {}. kameňa:".format(i + 1))
    stoneRow = int(input())
    print("Zadajte stĺpcové súradnice {}. kameňa:".format(i + 1))
    stoneColumn = int(input())
    stones.append([stoneRow,stoneColumn])
```

(príklad načítania súradníc kameňov)
Všetko načítanie vstupu je dosiahnuté klasicky pomocou input().

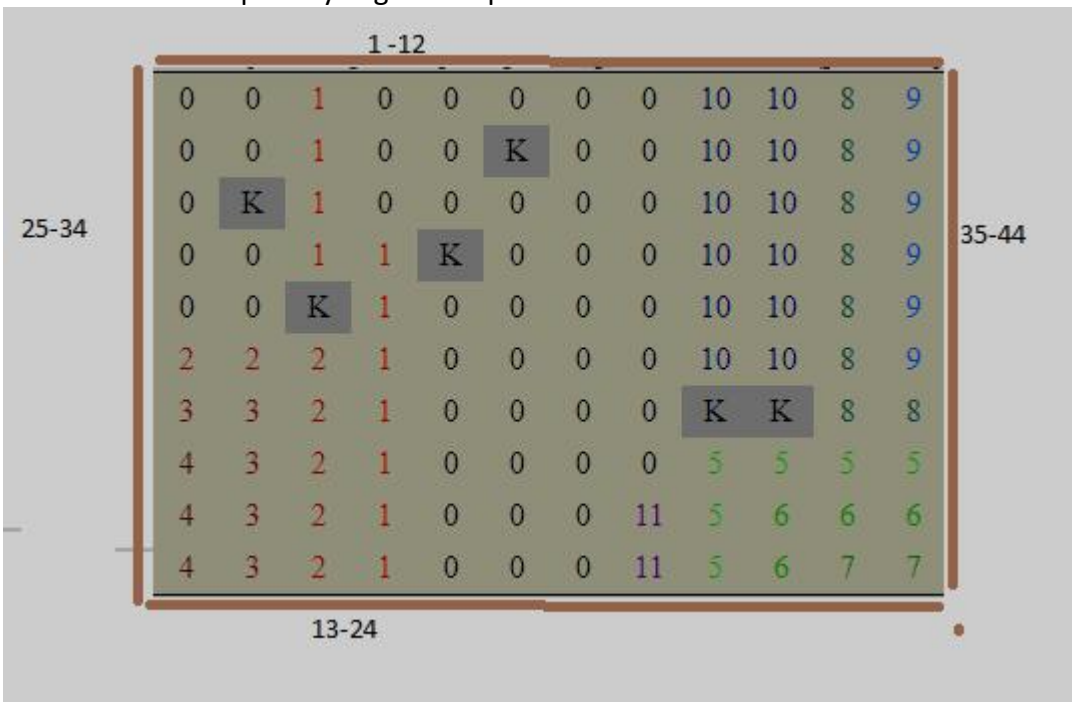
Pokračovanie opisu algoritmu:

Po úspešnom načítaní všetkých vstupných premenných sa spustí funkcia **mainLoop(),** ktorá riadi hlavný chod programu.
Aktuálne je v tejto funkcii nastavený **unitCount=100**, ktorý diktuje počet jedincov pre každú generáciu.
Pre prípadné menenie tejto hodnoty je optimálne zanechať hodnotu deliteľnú desiatimi.
Na začiatku sa vygeneruje prvá generácia, ktorú tvorí 100 kompletne náhodných chromozómov/jedincov. Stavba každého chromozómu, resp. Hodnoty každého génu sú opísané podrobnejšie v ich časti dokumentácie.
Po úspešnom vygenerovaní prvej generácie je potrebné generovať ďalšie generácie.
Tento algoritmus som rozdelil do troch fáz.
V prvej sa vyberie 10% jedincov s najlepšou hodnotou fitness.
V druhej fáze sa vyberú podľa zadanej metódy výberu jedincov dvaja rodičia pre kríženie, toto sa opakuje 70 krát pre generáciu so 100 jedincami, teda máme 70 “detí”. Pre tieto deti sa takisto vykoná mutácia s 10% šancou, ktorá zmení náhodný gén na náhodne vygenerovaný nový gén.

Zvyšných 20% jedincov ďalšej generácie je kompletne nanovo vygenerovaných náhodným spôsobom, takisto ako v prvej generácii, len v tej tak boli vygenerovaní všetci.
Nové generácie sa teda skladajú z 10% najlepších jedincov minulej generácie, 70% krížených jedincov, a 20% kompletne nových jedincov.
Tvorenie nových generácií sa opakuje pokiaľ nebude splnený maximálny počet generácií, alebo sa nenájde perfektné riešenie pre nášho mnícha.
V oboch prípadoch program končí so správou, a v prípade, ak sa našlo riešenie, ho takisto aj vypíše v podobe 2D pola.

Podrobný opis vlastností použitých génov

Pre demonštráciu použitých génov si požičiam obrázok zo zadania.



Každý jeden možný vstup do záhrady je očíslovaný, pri záhrade 10x12 je to 10*2+12*2=44 vstupov. V tomto prípade, prvých 12 je horná stena, druhých 12 je dolná stena, prvých 10 ľavá a druhých 10 pravá stena. V rámci týchto stien sú očíslované zľava do prava a zhora dole. 4 miesta v rohoch, pre ktoré existuje viac ako jeden gén sú rôzne v tom, že pre každý ich gén je príslušný iný smer mnícha, teda v ľavom hornom rohu (gén 1 a 25), pre gén s hodnotou 1 je smer dole, pričom pre gén s hodnotou 25 je smer vpravo.

Ako už bolo spomenuté, dodržiavam maximálny počet génov v chromozóme, ako bolo zadané, pričom som zvolil nepoužívať žiadne gény pre zmenu smeru mnícha. Zmena smeru mnícha funguje nasledovne: ak narazí na prekážku, najprv sa pokúsi otočiť vpravo, a ak to nie je možné, tak vľavo. Ak sú obe nemožné, a mních nie je na okraji záhrady, jeho pohyb končí. Pri náhodnej generácii génov nie je možné vygenerovať hodnoty génov, ktoré už sú v danom chromozóme použité.

```
def genChrom(unitCount, numPos, rows, columns, stoneCount, stones, geneCount):
    generation = []
    fitness = []
    for i in range(0,unitCount):
        chromosome = generateRandomChromosome(numPos, rows, columns, stoneCount, stones, geneCount)
        generation.append(chromosome)
        garden = createGarden(rows, columns, stoneCount, stones)
        generateGarden(chromosome, garden)
        singleFitness = getFitness(garden)
        fitness.append(singleFitness)
    return generation, fitness
```

Pre generáciu viacero chromozómov používam funkciu **genChrom()**, ktorá sa stará o generáciu potrebného počtu chromozómov a ich pridanie do listu, takisto ako aj o zistenie ich fitness pomocou funkcie **getFitness().**

Pre zistenie fitness jedinca vytváram jeho záhradu a posielam mnícha na prechádzku. **createGarden()** vytvorí základnú záhradu, a **generateGarden()** na nej vygeneruje konkrétne cesty nášho mnícha podľa poskytnutého chromozómu. Z takto vygenerovanej záhrady potom jednoducho **getFitness()** spočíta pohrabané políčka a vráti hodnotu fitness.
Funkcia, ktorá generuje každý náhodný chromozóm je **generateRandomChromosome()**, pričom jej hlavná úloha je nevkladať do chromozómu gény, ktoré sa v ňom už nachádzajú.

```
def generateRandomChromosome(numPos, rows, columns, stoneCount, stones, geneCount):
    chromosome = []
    for x in range(0,geneCount):
        chromosome.append(0)
    for i in range(0,geneCount):
        fine = False
        while(True):
            geneNum = random.randint(1, numPos)
            finecount = 0
            for y in range(0,i):
                if(chromosome[y]==geneNum):
                    finecount = 1
            if(finecount == 0):
                break
        chromosome[i] = geneNum
    return chromosome
```

Opis pohybu a rozhodovania mnícha

Pohyb mnícha začína vždy v prvom géne daného chromozómu. Tento pohyb sa koná vo funkcii **generateGarden().** Mních sa pozrie, či sa dá vstúpiť do záhrady z miesta, ktoré reprezentuje aktuálny gén. Ak sa dá, ide smerom vopred, ktorý určuje každý jeden typ génu osobitne. Ak nájde počas svojej cesty vopred nejakú prekážku(kameň reprezentovaný číslom -1, alebo pohrabaný piesok), bude sa musieť otočiť, pričom sa najprv pokúsi otočiť vpravo, a ak to je neúspešné, pokúsi sa otočiť vľavo. Ak sú oba možné smery otočenia mnícha vyplnené prekážkou, a mních nie je na okraji záhrady, jeho cesta končí. Ak prvé miesto vstupu mnícha do záhrady, ktoré je určené génom danej cesty je obsadené, mních sa presunie na ďalší gén, teda na ďalšie miesto na okraji záhrady.
Ukážka smeru chôdze nadol z funkcie **generateGarden()**:

```
if(direction == 1):
    if (row+1 >=0 and row+1<rows and garden[row+1][column] == 0):
        row+=1
        continue
    else:
        if(column-1 >=0 and column-1 < columns and garden[row][column-1] == 0):
            column-=1
            direction = 4
            continue
        elif(column+1 >=0 and column+1 < columns and garden[row][column+1] == 0):
            column+=1
            direction = 3
            continue
        else:
            brick = True
            able = False
```

Pre každý smer je naprogramované otočenie najprv vpravo, potom vľavo(v prvej **else** časti obrázku).
Smer = direction:
1=dole
2=hore
3=vpravo
4=vľavo

Spôsob tvorby novej generácie:

```
for i in range(0,generationCount):
    newGen, newGenFitness = firstPhase(unitCount, generation, fitness)
    secondPhaseGen, secondPhaseFitness = secondPhase(unitCount, generation, fitness, selection, geneCount, rows, columns, stoneCou
    for j in range(0, len(secondPhaseGen)):
        newGen.append(secondPhaseGen[j])
        newGenFitness.append(secondPhaseFitness[j])
    thirdPhaseGen, thirdPhaseFitness = genChrom(int(unitCount/100*20), numPos, rows, columns, stoneCount, stones, geneCount)
    for k in range(0, len(thirdPhaseGen)):
        newGen.append(secondPhaseGen[k])
        newGenFitness.append(secondPhaseFitness[k])
    generation = newGen
    fitness = newGenFitness
    if(i> 0 and i%100==0):
        print("max fitness {}. generacie: {}".format(i, max(fitness)))
    if(max(fitness) == (rows*columns)-stoneCount):
        print("našlo sa riešenie v {}. generacii:".format(i))
        garden = createGarden(rows, columns, stoneCount, stones)
        generateGarden(generation[fitness.index(max(fitness))], garden)
        printGarden(garden)
        return
print("Nepodarilo sa nájsť kompletnú cestu.\nV poslednej generácii bola najvyššia hodnota fitness {}.".format(max(fitness)))
```

Tento proces je rozdelený na 3 časti:
Prvá fáza: Výber 10% najlepších jedincov z aktuálnej generácie.

```
def firstPhase(unitCount, generation, fitness):
    bestCount = int(unitCount/100*10)
    newGen = []
    newGenFitness = []
    for i in range(0, bestCount):
        idx = fitness.index(max(fitness))
        newGen.append(generation[idx])
        newGenFitness.append(fitness[idx])
        #fitness.pop(idx)
        #generation.pop(idx)
    return newGen, newGenFitness
```

Druhá fáza: výber podľa turnaja/rulety.
V tejto fáze sa vytvorí 70% jedincov novej generácie.
V oboch prípadoch prebehne s 10% šancou po vytvorení nového jedinca krížením aj mutácia, ktorá zmení jeden náhodný gén na náhodnú hodnotu z možných hodnôt génov pre daného jedinca.

Pri výbere podľa rulety:

```
if(selection == "ruleta"):
    fullRoulette = 0
    for a in range(0,len(fitness)):
        fullRoulette+=fitness[a]
    for i in range(0, secondCount):
        firstParent = rouletteSelection(fullRoulette, fitness, generation)
        secondParent = rouletteSelection(fullRoulette, fitness, generation)
        roll = random.randint(0,geneCount)
        newChrom = []
        for x in range(0,roll):
            newChrom.append(firstParent[x])
        for y in range(roll,geneCount):
            newChrom.append(secondParent[y])
        if(mutationRoll<=chanceOfMutation):
            newChrom = mutation(newChrom, geneCount, numPos)
        garden = createGarden(rows, columns, stoneCount, stones)
        generateGarden(newChrom, garden)
        singleFitness = getFitness(garden)
        newGen.append(newChrom)
        newGenFitness.append(singleFitness)
    return newGen,newGenFitness
```

Náhodne sa vyberie hodnota **roll**, ktorá je z intervalu <0,početGénov>, pričom do nového potomka sa gény presunú z rodičov nasledujúcim spôsobom:
Z prvého rodiča sa presunú gény od prvého až po **roll**-tý.
Z druhého rodiča sa presunú ostatné.
Tým pádom je potomok vlastne kombináciou jeho rodičov.

Samotný výber ruletou funguje tak, že ešte pred zavolaním **rouletteSelection()** sa spočíta fitness všetkých jedincov v generácii (fullRoulette), z ktorej sa bude vyberať, a v **rouletteSelection**() sa vyberie náhodné číslo **select** od 0 po fullRoulette, pričom keď prechádzame list fitness všetkých jedincov v generácii, postupne odčítavame hodnotu fitness daného jedinca od hodnoty **select**, a v momente, kde nájdeme hodnotu fitness väčšiu/rovnú aktuálnej hodnote **select**, sme našli jedinca z rulety. Navyše je implementovaná iba 1/3 šanca, že jedinec s hodnotou fitness menšou ako medián všetkých z generácie, bude vybraný ako rodič. Dôvodom je zväčšenie rozdielu šance výberu medzi nízkou hodnotou a vyššou hodnotou, keďže môže byť veľký rozdiel v použiteľnosti medzi hodnotami 100 a 110, pričom 110 by inak mala iba o 10% väčšiu šancu na výber ako 100.

```
def rouletteSelection(fullRoulette, fitness, generation):
    done = 0
    med = statistics.median(fitness)
    while(done == 0):
        select = random.randint(0, fullRoulette)
        for i in range(0,len(fitness)):
            if(select<=fitness[i]):
                roll = random.randint(1,3)
                if(fitness[i] < med and roll == 1):
                    done = 1
                    return generation[i]
                elif(fitness[i]>=med):
                    done = 1
                    return generation[i]
            else:
                select -= fitness[i]
    #print("error 3")
    return None
```

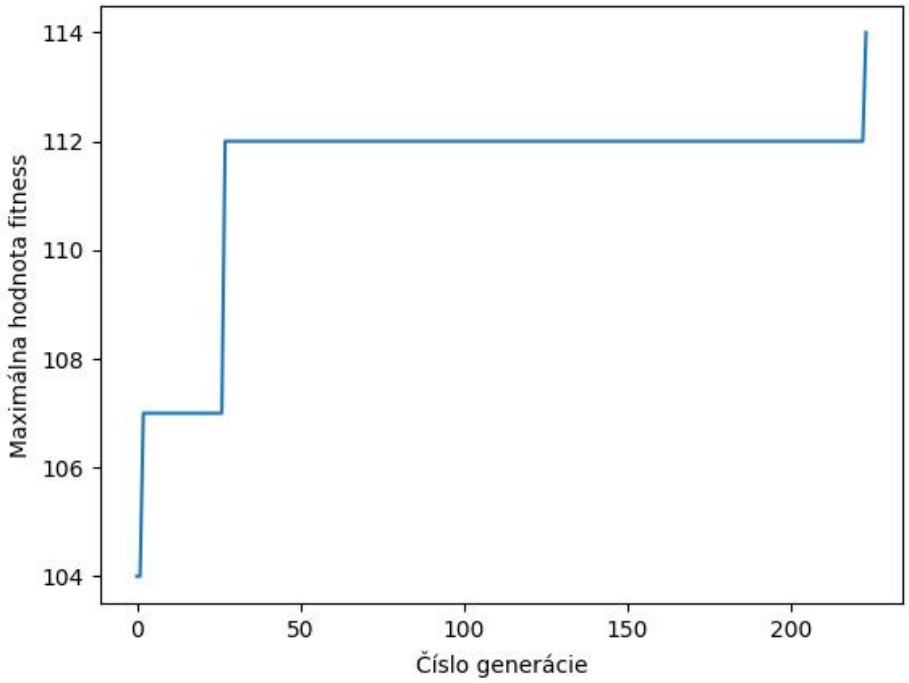
Pri výbere podľa turnaja:
Výber turnajom je omnoho jednoduchší.
Náhodne vyberiem troch jedincov, porovnám ich fitness, vyberiem jedinca s najvyššou hodnotou fitness, a mám rodiča. Toto opakujem aj pre druhého rodiča a môžem pomaly začať krížiť. Kríženie v tejto metóde výberu funguje úplne rovnako ako v predošlej, teda sa potomok podľa náhodne vybratého bodu v dĺžke chromozómu spojí z prvého a druhého rodiča. Nápodobne prebieha aj mutácia s 10% šancou.

Tým pádom nám už zostáva iba posledná, tretia časť.
V tejto časti vygenerujeme nových náhodných jedincov rovnakým spôsobom ako na začiatku prvej generácie. Z tejto časti bude pochádzať 20% jedincov nových generácií.

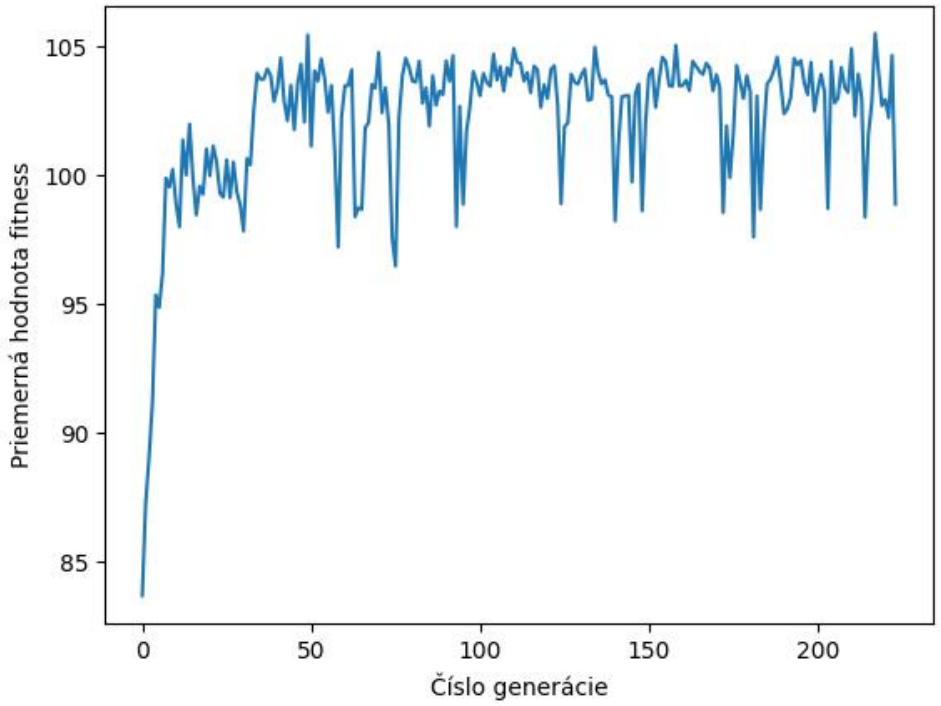
Teda prvá časť:10%
Druhá časť: 70%
Tretia časť: 20%

Vývoj fitness:

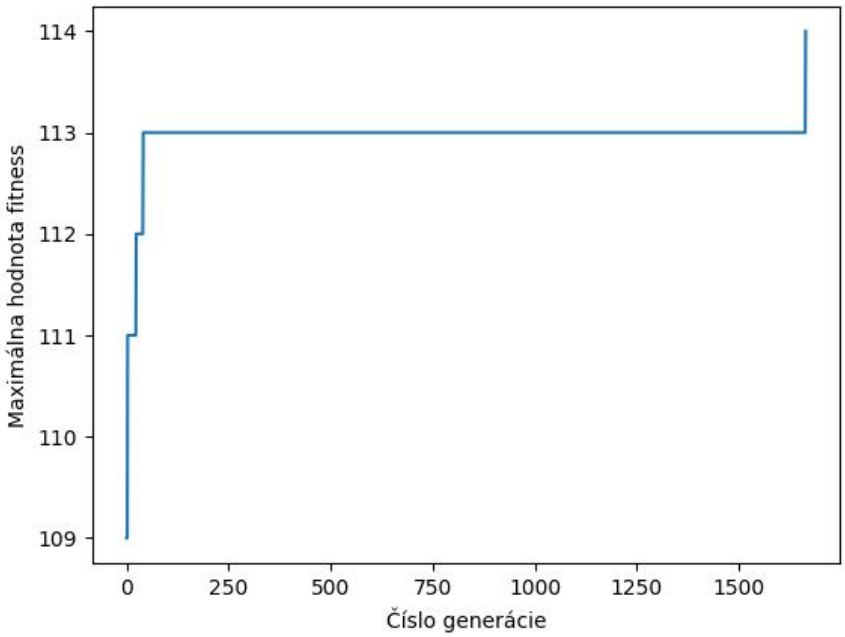
Pre štandardný vstup zo zadania (10x12 záhrada, 6 kameňov s danými súradnicami) je príklad vývoja fitness(v týchto príkladoch je použitý výber turnajom) nasledovný:
Maximálna hodnota fitness pre generáciu:



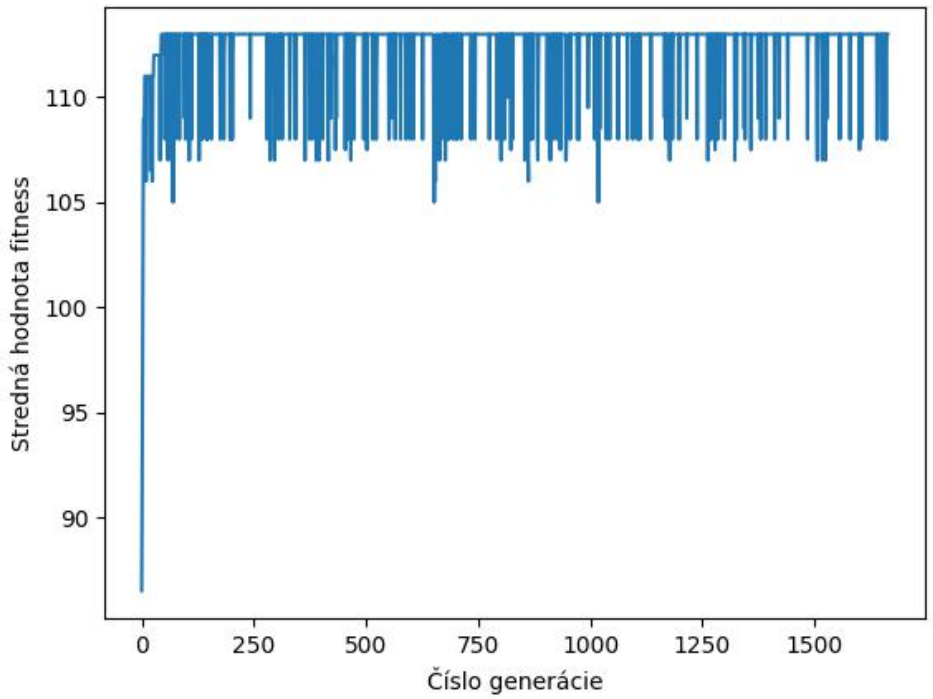
Priemerná hodnota fitness pre generáciu:



Pre rovnaký vstup v ďalšej iterácii:
Maximálna hodnota fitness:

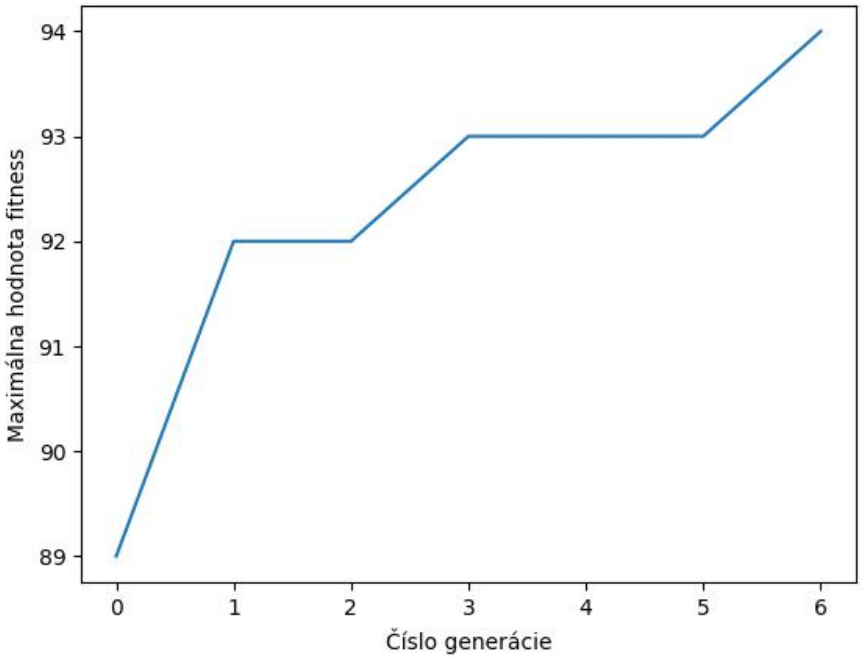


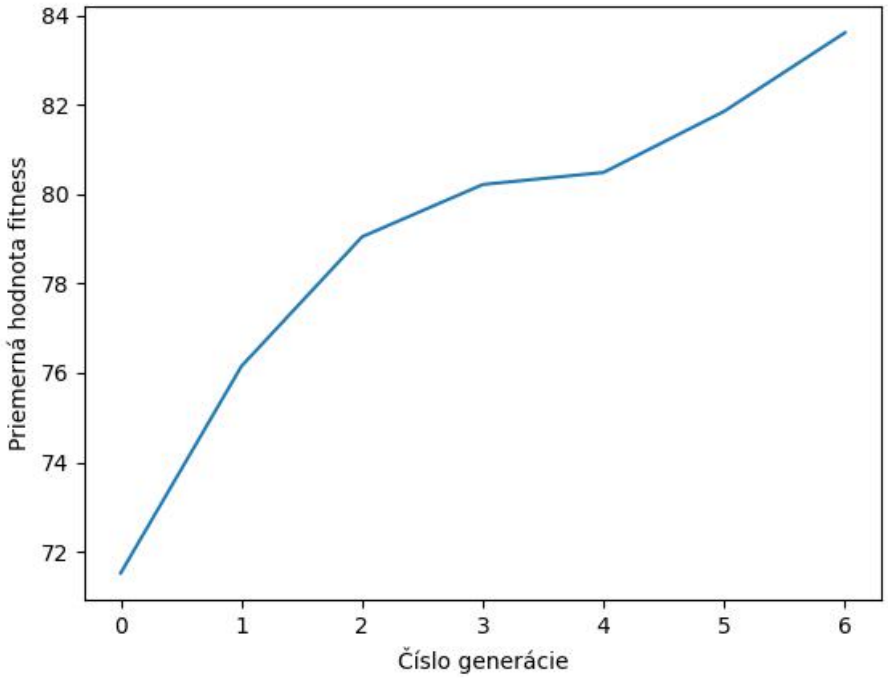
Stredná hodnota fitness:



Tu si môžeme povšimnúť, ako obrovský môže byť rozdiel medzi jednotlivými iteráciami nášho algoritmu.

Pre rovnaký vstup ako v zadaní, lenže záhrada je 10x10 sa našlo nasledovné riešenie:





Zhodnotenie vlastností vytvoreného systému:

Treba myslieť na fakt, že generácia génov a všetkého ostatného je v podstate kompletne náhodná, takže môžu byť(aj sú) obrovské rozdiely medzi jednotlivými iteráciami tohto evolučného algoritmu, čo ovplyvňuje presnosť štatistických meraní.
Často sa stáva, že sa kompletná odpoveď nájde hned v prvej stovke generácií, no niekedy sa nájde až v dvetisícej, alebo aj neskôr. Vzhľadom na túto skutočnosť nevidím pointu porovnávať rôzne výsledky pre rôzne parametre.
Napriek tomu, verím že optimálny počet génov je niekde v intervale medzi súčtom riadkov a stĺpcov, a maximálnym počtom génov v chromozóme (teda pre 10x12 záhradu s 6 kameňmi v intervale od 20 do 28).
Maximálny počet generácií je dobré nastaviť na číslo vyššie ako očakávaný priemer generácií na nájdenie riešenia, takže aspon niekoľko tisíc, pričom prirodzene vysokú časť riešení nájde oveľa oveľa rýchlejšie.

Záver:

V priebehu pracovania na tomto zadaní som sa naučil mnoho vecí o evolučných algoritmoch a podobných princípoch.
Zadanie som vypracoval v programovacom jazyku Python, v IDE PyCharm od JetBrains.