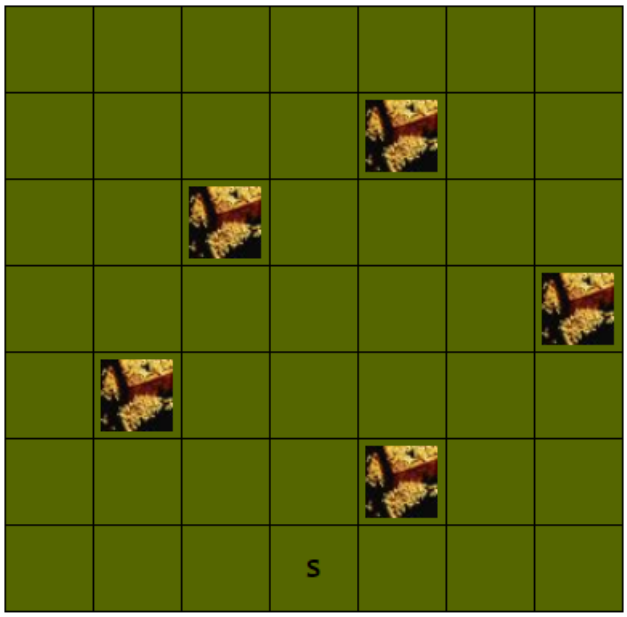
Umelá inteligencia

Zadanie č. 3 - Hľadanie pokladu

Dávid Šinko, FIIT STU, 3. ročník, ak. rok 2017/2018

Úloha

Majme hľadača pokladov, ktorý sa pohybuje vo svete definovanom dvojrozmernou mriežkou (viď. obrázok) a zbiera poklady, ktoré nájde po ceste. Začína na políčku označenom písmenom "S" a môže sa pohybovať štyrmi rôznymi smermi: "hore" **H**, "dolu" **D**, "doprava" **P** a "doľava" **L**. K dispozícii má konečný počet krokov. Jeho úlohou je nazbierať čo najviac pokladov. Za nájdenie pokladu sa považuje len pozícia, pri ktorej je hľadač aj poklad na tom istom políčku. Susedné políčka sa neberú do úvahy.



Zadanie

Horeuvedenú úlohu riešte prostredníctvom evolučného programovania nad virtuálnym strojom.

Tento špecifický spôsob evolučného programovania využíva spoločnú pamäť pre údaje a inštrukcie. Pamäť je na začiatku vynulovaná a naplnená od prvej bunky inštrukciami. Za programom alebo od určeného miesta sú uložené inicializačné údaje (ak sú nejaké potrebné). Po inicializácii sa začne vykonávať program od prvej pamäťovej bunky (Prvou je samozrejme bunka s adresou oooooo). Inštrukcie modifikujú pamäťové bunky, môžu realizovať vetvenie, programové skoky, čítať nejaké údaje zo vstupu a prípadne aj zapisovať na výstup. Program sa končí inštrukciou na zastavenie, po stanovenom počte krokov, pri chybnej inštrukcii, po úplnom alebo nesprávnom výstupe. Kvalita programu sa ohodnotí na základe vyprodukovaného výstupu alebo, keď program nezapisuje na výstup, podľa výsledného stavu určených pamäťových buniek.

Virtuálny stroj

Náš stroj bude mať **64 pamäťových buniek** o veľkosti 1 byte.

Bude poznať štyri inštrukcie: inkrementáciu hodnoty pamäťovej bunky, dekrementáciu hodnoty pamäťovej bunky, skok na adresu a výpis (**H**, **D**, **P** alebo **L**) podľa hodnoty pamäťovej bunky. Inštrukcie majú tvar podľa nasledovnej tabuľky:

inštrukcia	**tvar**
inkrementácia	00XXXXXX
dekrementácia	01XXXXXX
skok	10XXXXXX
výpis	11XXXXXX

Hodnota XXXXXX predstavuje 6bitovú adresu pamäťovej bunky s ktorou inštrukcia pracuje (adresovať je teda možné každú). Prvé tri inštrukcie by mali byť jasné, pri poslednej je potrebné si dodefinovať, čo sa vypíše pri akej hodnote bunky. Napríklad ak bude obsahovať maximálne dve jednotky, tak to bude **H**, pre tri a štyri to bude **D**, pre päť a šesť to bude **P** a pre sedem a osem jednotiek v pamäťovej bunke to bude **L**. Ako ukážku si uvedieme jednoduchý príklad:

Adresa:	000000	000001	000010	000011	000100	000101	000110	**...**
Hodnota:	00000000	00011111	00010000	01010000	00000101	11000000	10000100	**...**

Výstup tohoto programu bude postupnosť: P H H H D H ... Ďalšie hodnoty budú závisieť od hodnôt nasledujúcich pamäťových buniek. (Dúfam, že je jasné, že uvedenú postupnosť vypisuje inštrukcia v pamäťovej bunke s adresou 5. A program je samomodifikujúci sa, takže vypísané hodnoty nezodpovedajú hodnotám na začiatku, ale počas behu programu.) **Sú možné aj lepšie reprezentácie hodnôt pre H D P a L, napríklad podľa posledných dvoch bitov**.

Program sa zastaví, akonáhle bude splnená niektorá z nasledovných podmienok:

1. program našiel všetky poklady
2. postupnosť, generovaná programom, vybočila zo stanovenej mriežky
3. program vykonal 500 krokov (inštrukcií)

Či sa program zastaví, keď príde na poslednú bunku alebo pokračuje znovu od začiatku, si môžete zvoliť sami. (Môžete to nechať aj na voľbu používateľovi.)

Je možné navrhnúť aj komplikovanejší virtuálny stroj s rozšírenými inštrukciami a veľkosťou pamäťovej bunky viac ako osem bitov, ale musí sa dodržať podmienka maximálneho počtu pamäťových buniek 64 a limit 500 krokov programu. Rozšírenie inštrukcií

sa využíva nielen na zadefinovanie nových typov inštrukcií, ale hlavne na vytvorenie inštrukcií s podmieneným vykonávaním.

Zaujímavou (a fungujúcou) obmenou je aj použitie nepriamej adresácie. Vtedy predstavuje hodnota XXXXXX adresu bunky, kde je (v posledných šiestich bitoch) uložená adresa bunky ktorá sa má modifikovať alebo ktorá sa číta. Pri inštrukcii skoku je to adresa bunky kde je uložená adresa skoku. Nepriama adresácia sa dá využiť na oddelenie programu a údajov (napríklad keď z nejakého dôvodu nechceme vytvoriť samomodifikujúce sa programy). To je však pri tomto spôsobe tvorby programov len málokedy výhodné.

Použitý evolučný algoritmus

Hľadanie riešenia prebieha podľa nasledovného postupu:

1. Počiatočná populácia jedincov (najmenej 20) sa nainicializuje náhodnými hodnotami v stanovenom rozsahu 64 buniek.
2. Každému jedincovi sa určí fitness - počet nájdených pokladov a spravených pohybov na mapke do zastavenia jeho programu.
3. Ak je nájdený jedinec, ktorý našiel všetky poklady, riešenie končí s úspechom a vypísaním programu a postupnosti, ktorú vygeneroval.
4. Jednou z metód selekcie (ruleta alebo turnaj) sa určia rodičia a krížením a mutovaním vytvoria nových potomkov.
5. Jedince s istou pravdepodobnosťou mutujú a vstupujú do novej populácie.
6. Prejde sa na vykonávanie kroku 3.

Hodnota fitness závisí v prvom rade od počtu nájdených pokladov, ale za každý spravený pohyb na mapke sa jedincovi z fitness odpočítava.

Po štarte programu sa do konzoly zadá veľkosť populácie s ktorou chceme pracovať, ďalej typ selekcie (na výber je selekcia pomocou rulety alebo pomocou turnaju). Taktiež zadáme počet jedincov, ktorí sa budú do novej generácie vytvárať krížením rodičov alebo mutovaním. Päť jedincov sa vždy zachová z každej generácie a k tomu sa do veľkosti populácie ak sa tento počet ešte nedosiahol vygeneruje zvyšok nových jedincov. Nakoniec čo zadávame do konzoly je, či chceme upraviť počet vykonaných inštrukcií pre jedinca (prednastavená hodnota je 500 krokov) alebo nie. Ak chceme, tak následne zadáme požadovaný počet.

Potom program pokračuje. Pre každého jedinca z populácie vypočítava fitness podľa toho ako sa pohybujú po mapke. Ak jedinec narazí na poklad, do fitness sa mu priráta hodnota 1000, ak vykoná nejaký pohyb, z fitness sa mu uberie hodnota 0,3. Keď sa skončí ohodnotenie všetkých jedincov v populácií, začne sa kríženie a vytváranie nových jedincov podľa vopred

nastaveného typu selekcie (ruleta alebo turnaj). Pre každý typ selekcie je konkrétny algoritmus.

Algoritmy

Algoritmus pre selekciu ruletou

```
public static List<jedinec> ruleta(double suma, List<jedinec> jedinci, int kolkoVytvor)
```

Vstupnými argumentami sú suma, t.j. súčet všetkých fitness jedincov, zoznam týchto jedincov a argument pre počet, koľko sa má vytvoriť nových. Najprv si vytvorím pomocný zoznam, do ktorého si ukladám nových jedincov. Na začiatku si sem vložím päť najlepších jedincov z pôvodných. Potom nasleduje algoritmus, ktorý určí pre každého jedinca, aká je jeho pravdepodobnosť, teda aká veľká časť mu prislúcha a podľa toho má šancu byť zvolený ako rodič. Následne sa vyberú dvaja rôzni rodičia, ktorí sa krížia a vytvoria nového jedinca. Nový jedinec vzniká krížením a invertovaním bitov. Výstupom je zoznam nových jedincov.

Algoritmus pre selekciu turnajom

```
public static List<jedinec> turnaj(List<jedinec> jedinci, int kolkoVytvor)
```

Vstupné argumenty sú zoznam jedincov a premenná pre počet jedincov, koľkí sa majú vytvoriť krížením alebo mutáciou. Tak ako v rulete, aj tu si päť najlepších jedincov z predchádzajúcej generácie prekopírujem do novej a ďalej pokračujem samotným realizovaním výberu rodičov. Náhodne vygenerujem číslo, ktoré určuje počet jedincov, koľkí pôjdu do turnaju a z nich sa vyberie ten z najvyšším fitness, ktorý sa stane rodičom. Toto opakujem ešte raz, aby som mal dvoch rodičov. Keď mám, tak vytváram nových potomkov krížením a invertovaním bitov tak ako v rulete. Nakoniec funkcia vracia zoznam nových jedincov.

Keď sú vybratí rodičia, tak v každom type sa vytvárajú potomkovia tromi spôsobmi. Prvý je, že bunky rodičov krížia podľa náhodne zvoleného bodu a tiež sa zmení jeden náhodný bit. Druhý spôsob je, že sa v náhodne zvolenej bunke jedinca zmení náhodný bit a tretí spôsob je dosť podobný, ale zmení sa v náhodnej bunke jedinca posledný bit tejto bunky. Pre všetky mutácie sú konkrétne algoritmy v triede *jedinec*.

Kríženia a mutácie

Kríženie

```
public jedinec krizenie(jedinec rodic1, jedinec rodic2)
```

Vstupom sú dvaja rodičia, ktorý sa budú krížiť. Vo funkcií si náhodne rozdelím bunky novovytváraného jedinca na dve časti, kde do prvej prekopírujem bunky od jedného rodiča a do druhej od druhého. Následne tiež invertujem jeden náhodný bit v bode rozdelenia. Výstupom funkcie je nový jedinec.

Invertuj náhodný bit

```
public jedinec invertujNahodnyBit(jedinec jedinec)
```

Vstupom je jeden jedinec, ktorému sa v náhodnej bunke invertuje jeden náhodný bit. Čiže ak na zvolenom mieste mal 1, zmení sa na 0 a ak 0, tak opačne na 1. Výstupom je nový jedinec.

Invertuj posledný bit

```
public jedinec invertujPoslednyBit(jedinec jedinec)
```

Vstupom je tiež jeden jedinec. Funkcionalita je podobná ako v predošlej, ale s tým rozdielom, že tu sa vyberie náhodná bunka a invertuje sa v nej posledný bit. Výstupom je taktiež nový jedinec.

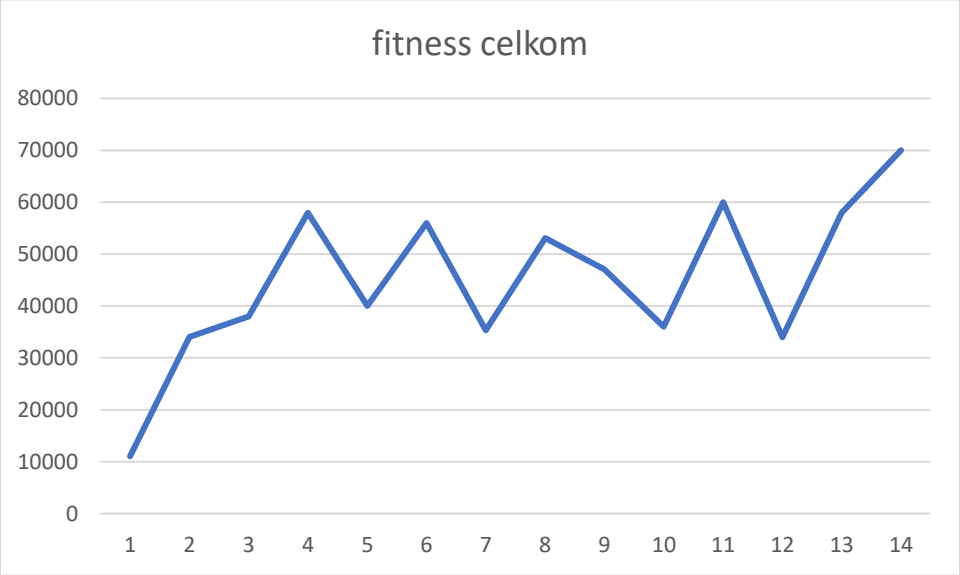
Po skončení vytvárania nových jedincov do novej generácie sa náhodne dogenerujú noví, ak treba do naplnenia počtu generácie. Inak máme vytvorenú novú generáciu jedincov a program sa ďalej vykonáva v cykle, čiže vypočítavajú sa fitness pre jedincov tejto novej generácie a znovu sa kríža tak ako predtým.

Jedince

Jedince sa skladajú z buniek, ktorých je 64 a každá bunka obsahuje 8 bitovú informáciu o tom, ako sa program bude vykonávať. Taktiež každý jedinec má informáciu o svojej fitness a pravdepodobnosť, ktorá sa vypočíta a je potrebná pre výber ruletou.

Vizualizácia

Obidva typy selekcií dávajú približne rovnaké výsledky. Medzi generáciami je rozostup 200. Fitness celkom značí, aký bol súčet všetkých fitness funkcií jedincov.



Obr. 1: *Graf priebehu programu*

Riešenie nie je najoptimálnejšie, lebo ako aj na grafe vidno, funkcia sa často mení hore a dole, nemá tendenciu sa len zvyšovať. Program je však napísaný dobre, no pozorujem nedostatok keď sa kopírujú zoznamy jedincov medzi a prechádza sa do ďalšej generácie. Program mi náhodne mení údaje v zoznamoch a preto sú výsledky niekedy horšie.