Dokumentácia druhého zadania z UI

Adam Štuller

April 7, 2019

1 Riešený problém – Zadanie

Zadaním bolo použiť evolučný algoritmus nad virtuálnym strojom a tak nájsť cestu, po ktorej keď prejdem tak nájdem všetky poklady na mape.

1.1 Mapa

Na vstupe do môjho programu bola mapa zo zadania, ale skúšal som algoritmus (a aj fungoval) aj na iných mapách. Menších aj väčších.

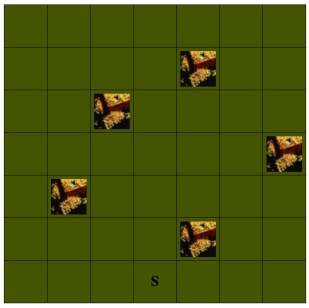

Figure 1: Mapa zo zadania

Túto a aj iné mapy som reprezentoval pomocou textového súboru, ktorý som vždy na začiatku načítal a spracovával ako dvojrozmerné pole. Písmeno X predstavovalo obycajné políčko, P poklad a S predstavoval štart.

2 Použitý Algoritmus

Nasleduje podrobný opis algoritmu, ktorý som použil. Program bol rozdelený na dve časti, prvou bol virtuálny stroj, do ktorého sa dal vložiť "program". Program bol listom "príkazov" o veľkosti maximálne 64. Príkazy boli reprezentované ako á bitové číslo, prvé dva bity boli konkrétna inštrukcia a zvyšných 6 pamäťový aj adresovaci priestor. Z pohľadu druhej časti programu - evolučného algoritmu, boli programy jedince a príkazy jednotlivé gény.

Figure 2: Reprezentácia mapy

2.1 Virtuálny stroj

Samotný virtuálny stroj bol triedou a jednotlivé operácie, ktoré vedel vykonávať boli metódy tejto triedy. Hlavnou metódou je run_program. Táto metóda berie ako parameter program, ktorý má vykonať a sekvenčne vykonáva príkazy. Program sa skončí keď virtuálny stroj narazí na jeho koniec alebo keď prebehne 500 inštrukcií. Run program je implementovaný ako generátor, ktorý po jednom vracia písmenko H, D, P, L, ktoré reprezentujú pohyb po mape. Z vonku sa dá teda program ukončiť napríklad keď nás postupnosť inštrukcií vyvedie mimo mapy.

Štyri inštrukcie, ktoré vie virtuálny stroj vykonať sú:

2.1.1 Write

Inštrukcia na výpis nejakého pohybu. V prípade že počas vykonávania programu príde virtuálny stroj na túto inštrukciu tak podľa posledných dvoch bitov príkazu určí aké písmenko sa má vypísať a to sa aj yieldne.

2.1.2 Increment

Táto inštrukcia inkrementuje hodnotu celého príkazu o 1. V prípade ze príkaz je 11111111 teda 255 tak po inkrementácií je hodnota 0.

2.1.3 Decrement

Táto inštrukcia naopak decrementuje hodnotu celého príkazu. V prípade, že máme decrementovať nulu tak výsledok je 255.

2.1.4 Jump

Operácia jump skočí na príkaz, ktorého adresa sa nachádza na posledných 6 bitoch príkazu. Potom pokračuje vo vykonávaní programu od daného príkazu.

2.2 Geneticky algoritmus

Genetický algoritmus som použil na to aby som našiel jedinca, v našom prípade program, ktorý nájde uspokojivé riešenie. Na začiatku algoritmu si vygenerujem počiatočnú generáciu jedincov- programov. Každému jednému nameriam fitness a potom sa celý algoritmus točí v cykle, kde najprv pomocou elitarizmu (dá sa aj vypnút) vyberiem n najlepších členov generácie. Potom pomocou selekcie postupne vyberám dvoch jedincov, ktorý sa stanú rodičmi nového jedinca. Tých dvoch skrížim a potom potom sa ho pokúsim zmutovať (záleží od pravdepodobnosti mutácie). Novovytvoreného jedinca najprv otestujem, priradím mu hodnotu fitness a ak je dostatočná, ukončím program a vŕatim tohto jedinca. Ak nie, priradím ho do novej generácie. Potom pokračujem v takejto tvorbe jedincov až pokiaľ nemá nová generácia rovnaký poćet jedincov ako stara. S nou generáciou opakujem proces kríženia a skúšam či sa nenájde dostatočné riešenie.

2.2.1 Populácia

Prvú populáciu tvorím pomocou metódy generate population. Táto metóda berie ako parameter počet jedincov (ten sa počas behu nemení) a minimálny počet nenulových príkazov v jedincovi- programe. V nej generujem zadaný počet programov a zaplním ich do určitej časti, čo je náhodné číslo od minimálneho počtu nenulových príkazov do 63. Potom vrátim list všetkých jedincov.

2.2.2 Selekcia

Používam dva druhy selekcie, turnaj a ruleta. Ruleta znamená, že pravdepodobnosť s ktorou vyberiem jedinca na množenie je priamo úmerná veľkosti jeho fitness. Teda každému priradím pravdepodobnosť fitness deleno suma fitness od všetkých jedincov.

Turnaj je odlišný. Najlepšiemu jedincovi priradí pravdepodobnosť p a pokúsi sa ho vybrať. Ak sa mu to nepodarí, pokračuje na druhého najlepšieho ale už iba s pravdepodobnosťou p*(1 - p)n̂ pričom n je poradie jedinca od najlepšieho po najhorší. Ak sa mu nepodarí vybrať ani jedného, vyberie najlepšieho.s

2.2.3 Kríženie

Jedincov krížim dvoma spôsobmi. Buď vygenerujem náhodné číslo a z jedného rodiča zoberiem všetky gény po to číslo a od druhého od toho čísla. Druhou možnosťou je že si pri každom géne hodím mincou a podľa toho vyberiem gén buď z jedného alebo druhého.

2.2.4 Mutácia

Používam dva druhy mutácie. Prvou je zmena jedného génu. Ten náhodne vyberiem a namiesto neho dám na jeho miesto úplne náhodný iný gén. Druhým druhom je výmena dvoch génov. Čo sa pravdepodobností mutácií týka, začínam pri celkom veľkej pravdepodobnosti a postupne sa zmenšuje.

2.2.5 Fitness funkcia

Fitness funkcia je veľmi dôležitou častou algoritmu. V mojom prípade je to 1 + počet nájdených pokladov - (počet prejdených krokov * 0.001). Takýmto spôsobom rátam aj s prejdenými pokladmi ale aj s počtom prejdených krokov a vyhnem sa riešeniu s veľkým počtom krokov.

3 Optimalizácia riešenia

Po tom ako som dokončil celý program, som strávil veľmi veľa času konfiguráciou jednotlivých vyššie menovaných parametrov. Nakoniec sa mi ako optimálna veľkosť populácie javí rozmedzie 100 - 150 jedincov. Na selekciu je určite lepšia v tomto prípade ruleta. Dôvod na to je, že rozdiely medzi fiitness funkciami nie sú také veľké, teda jeden jedinec nikdy nebude dominovať. Týmto si zaistíme potrebnú diverzitu pri krížení. Elitarizmus je na úrovni 10-15 jedincov, teda 10 percent z populácie.

Pomocou tejto konfigurácie nájdem väčšinou dobré riešenie vo veľmi krátkom čase. Priemerne to je 100 až 300 generácií. Niekedy sa stane, že algoritmus uviazne na lokálnom maxime štyroch najdených pokladov a nevie sa cez to dostať a vtedy mu to trvá niekoľko tisíc generácií. To je ale iba malá časť zo všetkých behov programu.

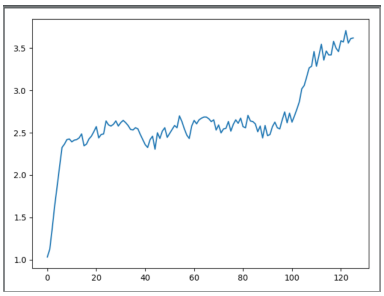

Figure 3: Graf priemerného riešenia y- priemerná fitness, x- počet generácií