Evolučný algoritmus (zadanie 1b)

ZS 2024/25

Martin Kvietok

Obsah

Martin Kvietok ID: 127214

Opis zadania

Úlohou je navrhnúť algoritmus založený na evolučných princípoch, ktorého cieľom je optimalizovať správanie jedincov, aby našli všetkých päť pokladov umiestnených na 2D hracom poli s rozmermi 7x7. Každý jedinec je reprezentovaný sekvenciou inštrukcií s hodnotami v rozsahu 0–255, ktoré určujú jeho pohyb a správanie. Inštrukcie zahŕňajú bitové operácie, ako sú inkrementácia a dekrementácia, pohybové príkazy (hore, dole, vľavo, vpravo) a príkaz "JUMP", ktorý presúva vykonávanie inštrukcií na inú pamäťovú adresu. Tieto inštrukcie sú spracovávané virtuálnym strojom so 64-bajtovou pamäťou, kde každá pamäťová bunka obsahuje 8 bitov.

Algoritmus využíva evolučné princípy, pričom jedinci prechádzajú genetickými operáciami, ako sú kríženie a mutácia, aby sa postupne zlepšovali v priebehu generácií. Vývoj jedincov neprebieha cez tradičné logické výpočty, ale je riadený výlučne evolučnými operáciami.

Opis riešenia

Na riešenie problému som použil evolučný algoritmus, inšpirovaný prednáškami, a implementoval som ho v jazyku Python, ktorý je ideálnou voľbou pre projekty zamerané na umelú inteligenciu. Zvolil som objektovo orientovaný prístup, ktorý umožňuje lepšie štruktúrovanie riešenia.

Inicializácia počiatočnej generácie

Na začiatku programu sa načítajú vstupné parametre hracieho poľa (class **Environment**) zo vstupného súboru input.txt. Následne program vytvorí novú inštanciu prostredia, kde sa budú vykonávať evolučné procesy, vrátane vyhodnocovania fitness funkcií pre jednotlivých agentov.

Ďalej program vytvorí náhodných jedincov (class **Agent**), ktorí budú reprezentovať rôzne stratégie prechodu hracím poľom. Títo agenti sú inicializovaní náhodnými inštrukčnými sadami a vložení do úvodnej populácie (class **Population**). Táto inicializačná populácia slúži ako východiskový bod pre ďalšie evolučné operácie, ako je výber, kríženie, mutácia a hodnotenie výkonnosti agentov.

Optimalizoval som inicializačné nastavenie na **201 agentov** a **maximálne 2000 generácií**, čo poskytuje dostatočný priestor aj čas na efektívne nájdenie riešenia.

Fitness funkcia (Environment - def fitness_function(self, agent)):

V prvej fáze algoritmu sa testujú inštrukcie všetkých jedincov v populácii pomocou virtuálneho stroja (class **Machine**), pričom si každý jedinec uloží svoju výslednú trajektóriu pohybu do **agent.cmd_set[]**. Následne sa každému agentovi priradí hodnota fitness na základe hodnotiacej funkcie, ktorá spracováva výsledné trajektórie nasledovným spôsobom:

1. **Kladná fitness hodnota**: Agent získa fitness hodnotu, ktorá je súčtom počtu nájdených pokladov a bonusu za efektívnosť krokov, vyjadreného vzorcom:
   Fitness = Počet pokladov + 0,1 * (počet krokov / 100).

2. **Nulová fitness hodnota**: Ak agent nevykoná žiadne kroky, jeho fitness hodnota bude **0**.

3. **Negatívna fitness hodnota**: Agent môže dostať negatívnu hodnotu fitness za nasledujúce stavy:

   - **-1**: Ak vykoná viac ako 500 inštrukcií vo fáze spracovania virtuálnym strojom
   - **-(Počet pokladov + 0,1 * Kroky / 100)**: Ak agent vykoná krok mimo hracieho poľa, dostane penalizáciu, ktorá závisí od počtu nájdených pokladov a krokov.

Tvorba Novej Generácie (Population - def create_new_generation(self, distractor)):

Pri tvorbe novej generácie sa populácia agentov triedi podľa ich fitness hodnôt (uložených v **agent.fit_index**). Po tomto triedení sa vytvorí nová generácia, pričom sa zachová rovnaký počet agentov ako v predošlej generácii s rovnakým počtom inštrukcií.

Vytváranie novej generácie môže byť ovplyvnené parametrom **distractor**, ktorý je zadaný ako príznak:

- **Distractor = 0**: Hlavné kríženie.
- **Distractor = 1**: Aktivuje alternatívnu sadu kríženia, ktorá je navrhnutá na rozbitie lokálnych extrémov, čím zabráni tomu, aby algoritmus uviazol na nevhodnom riešení. Táto technika bola implementovaná preto, že program nedokázal nájsť všetkých 5 pokladov do 2000 generácií, pretože sa populácia "zasekla" v extréme.

Elitizmus

Po vyhodnotení agentov sa populácia usporiada podľa fitness hodnôt vzostupne. V rámci tejto fázy som implementoval **elitizmus**, kde najlepšie hodnotený jedinec zostáva zachovaný pre ďalšiu generáciu. Tento prístup zabezpečuje, že aspoň jeden vysoko výkonný jedinec sa prenesie do ďalšej generácie, čím sa chráni dobrý genetický materiál. Zistil som, že kopírovanie iba jedného jedinca je najefektívnejšie, pretože bráni tomu, aby sa celá populácia uzavrela do lokálnych maxím, a zároveň podporuje ďalší vývoj a diverzitu v nasledujúcich generáciách.

Predvýber

a) Pri hlavnom výbere som vyberal štyroch náhodných jedincov a rozdelil ich do dvoch párov. Títo dvaja jedinci sa zúčastnili turnaja, aby sa určil ten s najvyššou fitness hodnotou. -> Efektívnejšie riešenie

b) V druhej variante som vyberal úplne náhodného rodiča (**parent1**) a najlepšieho z troch náhodne vybraných jedincov, čo zabezpečilo väčšiu mieru náhodnosti. -> Náhodnejšie riešenie

Turnajový výber (Population - def tournament(self, parent1, parent2))

Turnajový výber sa osvedčil ako efektívna metóda, pretože zabezpečil, že do kríženia sa dostane vždy najlepší jedinec. Ak sa však fitness hodnoty rodičov rovnali, rozhodoval som o víťazovi náhodne.

Počas hľadania najlepšieho riešenia som testoval aj ruletový výber, avšak ten nedokázal dosiahnuť lepšie výsledky.

Kríženie (Population - def crossover(self, parent1, parent2))

Kríženie bolo pôvodne navrhnuté na základe štandardného prístupu prezentovaného na prednáške, avšak prešlo významnou modifikáciou. Pomocou náhodnej funkcie som vybral náhodný cutpoint, čím som každého jedinca rozdelil na dve časti (začiatok a koniec). Pri krížení som uprednostňoval gény rodiča s vyššou fitness hodnotou, pričom jeho gény sa vždy umiestnili na začiatok inštrukčného reťazca.

Každý pár rodičov generoval dvoch potomkov. Prvý potomok bol efektívnejší, pretože vznikol kombináciou začiatkov oboch rodičov, čo umožnilo rýchlejšie spracovanie inštrukcií a znížilo pravdepodobnosť výskytu génu JUMP na začiatku reťazca, ktorý mohol spôsobovať, že sa jedinec vôbec nepohol. Tento prístup viedol k výraznému a rýchlemu rastu, pričom často dochádzalo k exponenciálnemu zlepšeniu.

V priemere sa tým dosahovali minimálne tri nájdené poklady počas 200 generácií. Druhý potomok bol vytvorený kombináciou začiatku jedného rodiča a konca druhého, čím sa zabezpečila väčšia genetická diverzita.

Zistil som, že generovanie iba jedného potomka viedlo k rýchlemu uviaznutiu v lokálnych extrémoch a graf diverzity populácie sa začal zmenšovať, čo výrazne obmedzovalo vývoj populácie. Pretože sa začala tvoriť generácia identických jedincov.

Príklad s uprednostnením:

- Rodič 1: XXXYYY
- Rodič 2: ZZZAAA
- Výsledok: Dieťa 1 - XXXZZZ a Dieťa 2 - XXXAAA

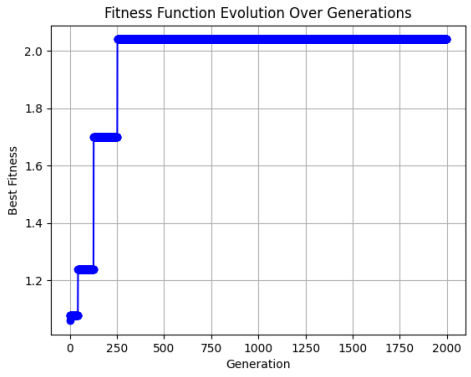
Náhodné kríženie (Population - def random_crossover(self, parent1, parent2))

Výhodou tejto metódy bolo získanie maximalizáciu náhodnosti, pričom najefektívnejším prístupom sa ukázal model z prednášky. Zároveň som testoval aj multipoint crossover, avšak tento prístup sa neosvedčil ako efektívny.

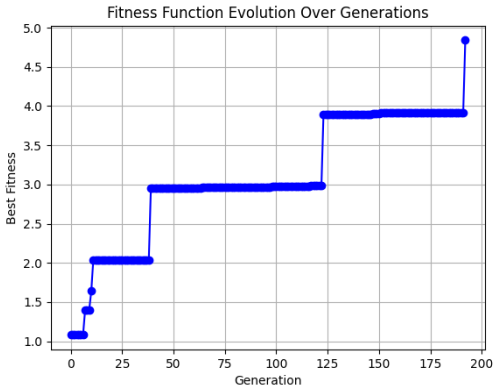
Náhodné kríženie z prednášky:

- Rodič 1: XXXYYY
- Rodič 2: ZZZAAA
- Výsledok: Dieťa 1 – ZZZYYY a Dieťa 2 - XXXAAA

Týmto spôsobom som zabezpečil, že generované potomstvo má tendenciu zachovať výhodné vlastnosti z rodičov, čím sa zvyšuje potenciál pre evolučný pokrok a využil som to ďalej pri vylepšovaní algoritmu.



Použitie náhodného kríženia



Použitie kríženia s uprednostnením

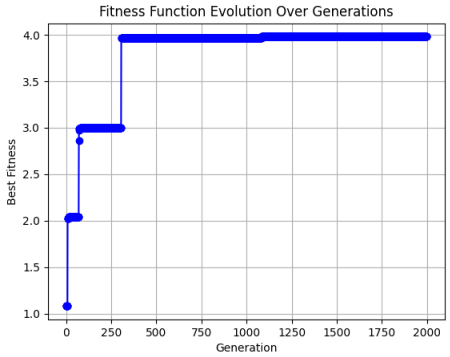
Mutácia (Agent - def mutate(self, mutation_rate))

Mutácia prebieha v dvoch fázach:

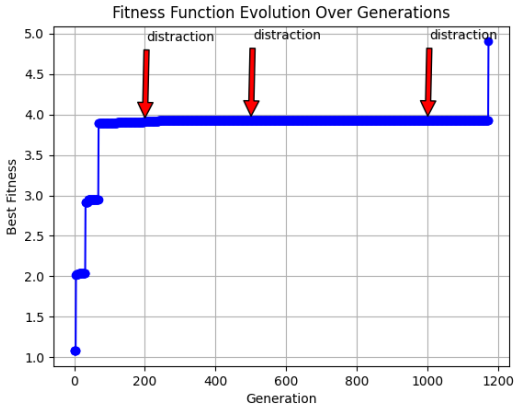
1. **Výber počtu iterácií mutácie:** Pri nižších hodnotách mutačného pomeru (do 0.5) som vybral počet iterácií mutácií v rozmedzí od 0 do 4, zatiaľ čo pri vyšších hodnotách sa tento rozsah zvýšil na 3 až 9.
2. **Výber formy mutácie:** Mutácie môžu prebiehať rôznymi spôsobmi, a to buď inkrementovaním alebo dekrementovaním náhodného čísla, alebo swapom dvoch náhodne vybraných čísel v agent.inst_set.

Tento prístup podporil diverzitu v populácii a prispel k ďalšiemu evolučnému pokroku v nasledujúcich generáciách a bol aplikovaný pri tvorbe každého jedinca.

Aby som predišiel ďalšej monotónnosti v evolučnom vývoji, implementoval som aj tzv. **"Budiča,"** ktorý po 200–300 generáciách randomizoval inštrukčné sety pomocou implementovania alternatívnej cesty a random_crossoveru spomínaného vyššie. Tento mechanizmus bol navrhnutý na základe experimentu, ktorý preukázal, že doteraz vyvíjaný spôsob má účinnosť do 200 inštrukcií. Budič umožnil "zlomiť" stagnujúci vývoj.



Bez Budiča



Po implementácií Budiča

Týmto postupom sa mi podarilo dosiahnuť 80+ % efektivitu, pričom do 2000 generácií s 201 agentmi som zabezpečil rast a diverzitu v evolučnom procese.

Martin Kvietok ID: 127214

Finálne testovanie:

Celkovo sa ukazuje, že úspešné generácie, ktoré dosiahli maximálny počet nájdených pokladov (5), vykazujú variabilitu v efektivite, pokiaľ ide o počet potrebných krokov. Najefektívnejšie generácie sa pohybujú medzi 19 a 30 krokmi, pričom rozdiely sú zjavné v dôsledku vysokej náhodnosti použitých stratégií.

Počet generácií	Počet nájdených pokladov	Počet potrebných krokov
139	5	19
1128	5	26
163	5	20
2000	4	15
109	5	27
184	5	30
320	5	23
74	5	23
160	5	22
239	5	24
252	5	27
1999	4	13
1077	5	20
355	5	20
164	5	30
603	5	29
1645	5	20
301	5	22
1999	4	16

User manual

Program je odovzdaný ako riešenie v prostredí Visual Studio 2022 a obsahuje Python environment s pridanými knižnicami (random a matplotlib pre vykresľovanie grafov). Spúšťacím súborom programu je setup.py, preto je potrebné skontrolovať, či je tento súbor nastavený ako Startup File. Po skompilovaní sa program spustí, začne hľadať a vypisovať generácie, ktorými prechádza. Po dokončení vypíše počet nájdených pokladov, počet krokov a počet generácií, a na záver zobrazí graf evolúcie najlepšej fitness funkcie.

Záver

Navrhnutý evolučný algoritmus úspešne rieši zadaný problém, pričom jeho výkon závisí od kombinácie vstupných parametrov a inicializácie agentov. Je možné dosiahnuť rýchle exponenciálne výsledky, ale aj dlhé monotónne vyhľadávania. Hlavnou výhodou algoritmu je schopnosť nájsť uspokojivé množstvo pokladov za relatívne krátku dobu, často v rozmedzí 3 až 4 poklady do 200 generácií. Existuje však priestor na zlepšenie, najmä pokiaľ ide o skĺzavanie do extrémov, ktoré sa preukázalo ako významná prekážka pri optimalizácii výkonu. To naznačuje, že ďalší výskum a implementácia nových stratégií by mohli posilniť variabilitu a adaptabilitu agentov, čím by sa zlepšila celková efektivita algoritmu. (napr. hodnotenie samotných génov (inštrukcií) jedincov a následné skladanie). Taktiež by sa program dal vylepšiť o hľadanie najlepšej cesty k pokladu.