Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Umelá inteligencia

Zadanie 2 - Hľadanie pokladu

Petra Miková

Petra Miková
ID: 120852

Contents

Petra Miková
ID: 120852

Opis použitého algoritmu

Mojou úlohou bolo riešiť úlohu hľadania pokladov v dvojrozmernej mriežke pomocou evolučného algoritmu a za použitia virtuálneho stroja. Evolučný algoritmus je inšpirovaný princípmi evolúcie a dedičnosti v prírode a je možné ho použiť na riešenie optimalizačných problémov a vyhľadávanie najlepších riešení v priestore možností – v mojom prípade pre nájdenie najlepšej cesty s najviac vyzbieranými pokladmi a najmenej krokmi. Algoritmus pracuje s populáciou jedincov, ktorí sú reprezentovaní geneticky, a postupne vytvára nové generácie jedincov krížením, mutovaním a výberom na základe ich úspešnosti v riešení problému (fitnes funkcia). Týmto spôsobom sa algoritmus snaží konvergovať k optimálnemu riešeniu problému v priebehu viacerých generácií. Použitie virtuálneho stroja nám umožňuje pohyb po mriežke použitím 64 pamäťových buniek pre každého jedinca s využitím 4 inštrukcií. V nasledujucej časti opíšem použitie algoritmu v technickom zmysle a taktiež opíšem ďalšie dôležité časti kódu.

Opis riešenia

Trieda Individual

Pre reprezentáciu jedinca som zvolila triedu s nasledovnou štruktúrou:

```
class Individual:
    def __init__(self, path, memory_cells, treasures_found, fitness_func):
        self.path = path
        self.memory_cells = memory_cells
        self.treasures_found = treasures_found
        self.fitness_func = fitness_func
```

Obsahuje cestu, ktorú daný jedinec prešiel, jeho pamäťové bunky, nájdené poklady, a fitnes funkciu.

Vytvorenie prvej generácie

Vytvorenie prvej generácie sa líši od generovania tých ďalších.

```
def spawn_generation():
    for i in range(number_of_individuals):
        individuals.append(spawn_individual())
        individuals[i].memory_cells = []
        for j in range(mem_cells_count):
            if j <= first_gen_mem_cells_count:

individuals[i].memory_cells.append(numpy.uint8(random.randrange(256)))
            else:
                individuals[i].memory_cells.append(numpy.uint8(0))
```

V tejto funkcii si vytvoríme generáciu s vopred určeným počtom jedincov a nainicializujeme náhodnými hodnotami len polovicu pamäťových buniek pre každého z nich, aby evolúcia dokázala konvergovať.

Inicializácia jedinca

Na inicializáciu jedinca používam funkciu spawn_individual, kde sa jednoducho vytvorí objekt Individual a naplnia sa jeho atribúty základnými hodnotami.

```
def spawn_individual():
    path = []
    memory_cells = [numpy.uint8(random.randint(0, 255)) for _ in
range(mem_cells_count)]
    treasures_found = set()
    fitness_func = 0

    new_individual = Individual(path, memory_cells, treasures_found,
fitness_func)

    return new_individual
```

V prípade, že funkciu voláme pri vytváraní úplne prvej generácie, atribút memory_cells inicializujeme nanovo, keďže v prvej generácii chceme aby bola naplnená len ich polovica.

Vytvorenie cesty

Veľmi jednoduchá funkcia ktorá len vráti písmeno pohybu pre výpis cesty z bitov v pamäťovej bunke podľa zadania.

```
def return_move(bits):
    if bits == "00":
        return "H"
    elif bits == "01":
        return "D"
    elif bits == "10":
        return "P"
    elif bits == "11":
        return "L"
    return None
```

Virtuálny stroj

Funkcia virtuálneho stroja vykonáva inštrukcie reprezentované v pamäťových bunkách jedinca, ktoré si spracujeme na binárne stringy. Stroj pracuje v cykloch, pričom inkrementuje alebo dekrementuje hodnoty v pamäťových bunkách podľa inštrukcií. Skokové inštrukcie menia aktuálnu pozíciu v pamäti na základe adresy. Inštrukcia "Zápis cesty" pridáva pohybové kroky do cesty jedinca na základe informácie v inštrukcii. Celý proces pokračuje, kým sa nedosiahne maximálny počet krokov alebo neopustí rozsah pamäťových buniek.

```
def virtual_machine(individual):
    steps_count = 0
    cells_count = 0
    instructions = individual.memory_cells

    while steps_count <= max_steps and 0 <= cells_count < mem_cells_count:
        operation = format(instructions[cells_count], '08b')[0:2]
        address = int(format(instructions[cells_count], '08b')[2:], 2)

        if operation == "00":  # Inkrementácia
            individual.memory_cells[address] += 1
            individual.memory_cells[address] =
numpy.uint8(individual.memory_cells[address])

        elif operation == "01":  # Dekrementácia
            individual.memory_cells[address] -= 1
            individual.memory_cells[address] =
numpy.uint8(individual.memory_cells[address])
```

```
        elif operation == "10":  # Skok
            cells_count = address

        elif operation == "11":  # Zápis cesty
            move = format(instructions[cells_count], '08b')[6:]
            individual.path.append(return_move(move))

        steps_count += 1
        cells_count += 1
```

Operáciu reprezentujú prvé dva bity pamäťovej bunky a adresu tie zvyšné. Operácia pre zápis cesty využíva hore spomínanú funkciu, ktorá priraďuje jedincovi dané kroky.

Hľadanie pokladov

Táto funkcia vykonáva samotné pohybovanie jedinca po mriežke a hľadanie pokladov. Podľa toho o aký pohyb ide meníme pozíciu jedinca a zisťujeme, či sa na danom políčku nachádza poklad. Ak áno, pridáme ho jedincovi do zoznamu nájdených pokladov. Taktiež pozeráme na to, aby jedinec nevyšiel z mriežky, v takom prípade breakneme cyklus. Jedincovi priradíme prejdenú cestu a na konci zisťujeme, či náhodou nenašiel všetky poklady z mriežky. Ak áno, hľadanie môžeme ukončiť.

```
def treasure_hunt(individual):
    row, column = start_position
    updated_path = []

    for move in individual.path:
        if move == "H":
            row -= 1
        elif move == "D":
            row += 1
        elif move == "P":
            column += 1
        elif move == "L":
            column -= 1

        if row < 0 or row >= map_size[0] or column < 0 or column >=
map_size[1]:
            break

        current_position = (row, column)
        if current_position in all_treasures and current_position not in
individual.treasures_found:
            individual.treasures_found.add(current_position)

        updated_path.append(move)

    individual.path = updated_path

    if len(individual.treasures_found) == len(all_treasures):
        return
```

Petra Miková
ID: 120852

Tvorenie novej generácie

Proces tvorenia novej generácie je v mojom prípade taký, že si najprv zachovám pomocou percenta elitarizmu elitných (najlepších) jedincov. Ak sa uživateľ rozhodne nepoužiť elitarizmus, teda zadá nulu, tvoria sa nové jedince od začiatku a nezachovávame žiadne. Po tomto kroku si určíme koľko nových jedincov sa teda má generovať a vo forloope ich postupne inicializujeme pomocou už vyššie spomínanej funkcie spawn_individual. Ďalej vykonám metódu selekcie, znova závisí od vstupu uživateľa (ak si nevyberie, defaultne sa používa ruleta). Eventuálne sa vykoná na generácii mutácia a funkcia vráti novú vytvorenú generáciu.

```
def create_generation(sorted_gen, selection):
    new_gen = {}

    elite_end = int(elitism * number_of_individuals)

    if elite_end != 0:
        for j in range(elite_end):
            new_gen[j] = sorted_gen[j]

    start_fresh = int(elitism * number_of_individuals)
    end_fresh = int((elitism + new_individuals) * number_of_individuals)
    for k in range(start_fresh, end_fresh):
        new_gen[k] = spawn_individual()

    if selection == "ruleta":
        roulette_selection(sorted_gen, new_gen)
    else:
        tournament_selection(sorted_gen, new_gen)

    mutation(new_gen)
    new_gen = list(new_gen.values())

    return new_gen
```

Selekcia

Ruleta

Selekcia ruletou zo zostupne usporiadaného listu jedincov generácie vyberá jedincov na základe ich vhodnosti, ktorá je reprezentovaná váhami. Váhy sú určené na základe fitnes funkcie jedincov. Po určení počiatočného indexu pre elitných a nových jedincov sa náhodne vyberú dvaja rôzni rodičia, ktorých skrížim a vytvorím tak nového jedinca.

```
def roulette_selection(sorted_gen, new_generation):
    start_index = int((elitism + new_individuals) * number_of_individuals)
    weights = [ind.fitness_func for ind in sorted_gen]

    for i in range(start_index, number_of_individuals):
        parent1 = random.choices(sorted_gen, weights=weights)[0]
        parent2 = random.choices(sorted_gen, weights=weights)[0]

        # Ensure that parent1 and parent2 are different individuals
        while parent1 == parent2:
            parent2 = random.choices(sorted_gen, weights=weights)[0]

        # Create a new offspring Individual object and perform crossover on
memory_cells
        offspring_memory_cells = crossover(parent1, parent2)
```

```
        offspring = Individual([], offspring_memory_cells, set(), 0)
        new_generation[i] = offspring
```

Turnaj

Selekcia turnajom náhodne vyberie pevný počet jedincov z populácie na základe percenta turnaja a najvhodnejší jedinec z turnaja je vybraný ako rodič. Toto sa opakuje dvakrát pre získanie dvoch rodičov a týchto znova skrížime aby sme dostali ich potomka – nového jedinca.

```
def tournament_selection(sorted_gen, new_generation):
    start_index = int((elitism + new_individuals) * number_of_individuals)
    num_tournament = int(tournament * number_of_individuals)
    for i in range(start_index, number_of_individuals):
        parent1 = sorted(random.choices(sorted_gen, k=num_tournament),
reverse=True, key=lambda x: x.fitness_func)[0]
        parent2 = sorted(random.choices(sorted_gen, k=num_tournament),
reverse=True, key=lambda x: x.fitness_func)[0]

        while numpy.array_equal(parent2.memory_cells,
parent1.memory_cells):
            parent2 = sorted(random.choices(sorted_gen, k=num_tournament),
reverse=True, key=lambda x: x.fitness_func)[0]

        offspring_memory_cells = crossover(parent1, parent2)
        offspring = Individual([], offspring_memory_cells, set(), 0)
        new_generation[i] = offspring
```

Kríženie

V mojej implementácii vykonávam kríženie takým spôsobom, že si náhodne vyberiem bod, na ktorom bude dochádzať ku kríženiu, a po tento bod kopírujem bunky prvého rodiča, a od tohto bodu bunky druhého.

```
def crossover(parent1, parent2):
    crossover_point = random.randint(1, mem_cells_count - 1)
    child_memory_cells = parent1.memory_cells[:crossover_point] +
parent2.memory_cells[crossover_point:]
    return child_memory_cells
```

Mutácia

Mutáciu vykonávam spôsobom, že prechádzam všetkých jedincov a na každom vykonám zmutovanie invertovaním pamäťových buniek.

```
def mutation(new_generation):
    start_index = int((elitism + new_individuals) * number_of_individuals)
    for i in range(start_index, number_of_individuals):
        if i in new_generation:
            individual = new_generation[i]
            new_generation[i] = invert_cells(individual)
```

Invertovanie sa vykonáva tak, že pre každú bunku pamäte v jedincovi sa náhodne rozhodne, či sa vykoná mutácia, a ak sa má vykonať, vyberie sa náhodné miesto (bit) v danej bunke pamäte a zmení sa jeho hodnota.

```
def invert_cells(individual):
    mutated_memory_cells = list(individual.memory_cells)

    for index, value in enumerate(individual.memory_cells):
        if random.random() < mutation_rate:
            shift = random.randint(0, 7)
            mutated_memory_cells[index] = numpy.uint8(value ^ (1 << shift))

    mutated_individual = Individual(individual.path, mutated_memory_cells,
individual.treasures_found, individual.fitness_func)
    return mutated_individual
```

Fitnes funkcia

Pre každého jedinca potrebujeme kalkulovať fitnes funkciu, ktorá určuje jeho vhodnosť pre riešenie problému v evolučnom algoritme. Počítam ju spôsobom, že čím viac nájdených pokladov, tým väčšiu hodnotu funkcie jedincovi priraďujem, a zároveň penalizujem množstvo krokov, aby sme vedeli vyselektovať riešenie s kratšou cestou, aj keď jedince našli rovnaký počet pokladov.

```
def calculate_fitness(individual):
    if not individual.path:
        return 0

    treasures_found = len(individual.treasures_found)
    total_steps = len(individual.path)

    treasures_weight = 10
    steps_weight = -0.001

    fitness = (treasures_weight * treasures_found) + (steps_weight *
total_steps)

    return fitness
```

Inicializácia evolučného algoritmu a main

V maine sa vykonáva hlavné spustenie celej implementácie tak, aby algoritmus pracoval korektne. Dokým uživateľ chce generovať riešenia, začne sa inicializovaním prvotnej generácie a vo forloope sa pokračuje s evolúciou na základe toho, koľko generácií chceme, aby vzniklo. Vnútri sa pre každého jedinca z generácie spustí virtuálny stroj, vykoná sa hľadanie pokladov, a vypočíta sa jeho fitnes funkcia. Týchto jedincov si potom zoradím podľa ich fitnes funkcie tak, že prvý v liste je najsilnejší jedinec. Po tomto spustím proces vytvárania ďalšej generácie, a taktiež si v každej iterácii udržujem alebo aktualizujem najlepšieho jedinca, ktorý nám zatiaľ ponúka najlepšie riešenie. Pre každú generáciu vypíšem zhodnotenie, a teda najlepšieho jedinca, jeho fitnes, a jeho cestu. Ak skončí evolúcia, teda uživateľ ďalej nechce už generovať, vypíše sa celkový najlepší jedinec a teda finálne nájdené riešenie.

```
if __name__ == "__main__":
    continue_generating = True
    counter = 1
    sorted_individuals = {}

    while continue_generating:
        spawn_generation()

        for generation in range(number_of_gens):
```

```
            for i in range(number_of_individuals-1):
                virtual_machine(individuals[i])
                treasure_hunt(individuals[i])
                individuals[i].fitness_func =
float(calculate_fitness(individuals[i]))

            sorted_individuals = sorted([ind for ind in individuals if
isinstance(ind, Individual)],
                                        key=lambda ind: ind.fitness_func,
reverse=True)

            new_generation = create_generation(sorted_individuals,
selection_choice)

            individuals = new_generation

            if best_individual is None or (
                    len(sorted_individuals) != 0 and
(sorted_individuals[0].fitness_func > best_individual.fitness_func)):
                best_individual = sorted_individuals[0]

            sorted_individuals = {}

            print("Najlepší jedinec pre", counter, ". generáciu:")
            print("Fitnes funkcia: ", individuals[0].fitness_func)
            print("Cesta:", individuals[0].path)

            counter += 1

        # Pýtame sa používateľa či chce pokračovať alebo nie
        user_input = input("Chcete pokračovať v generovaní? (ano/nie): ")
        if user_input.lower() != "ano":
            continue_generating = False
            print()
            print("Najlepší jedinec celkovo: ")
            print("Fitnes funkcia: ", best_individual.fitness_func)
            print("Cesta:", best_individual.path)
```

GUI a vstup od uživateľa

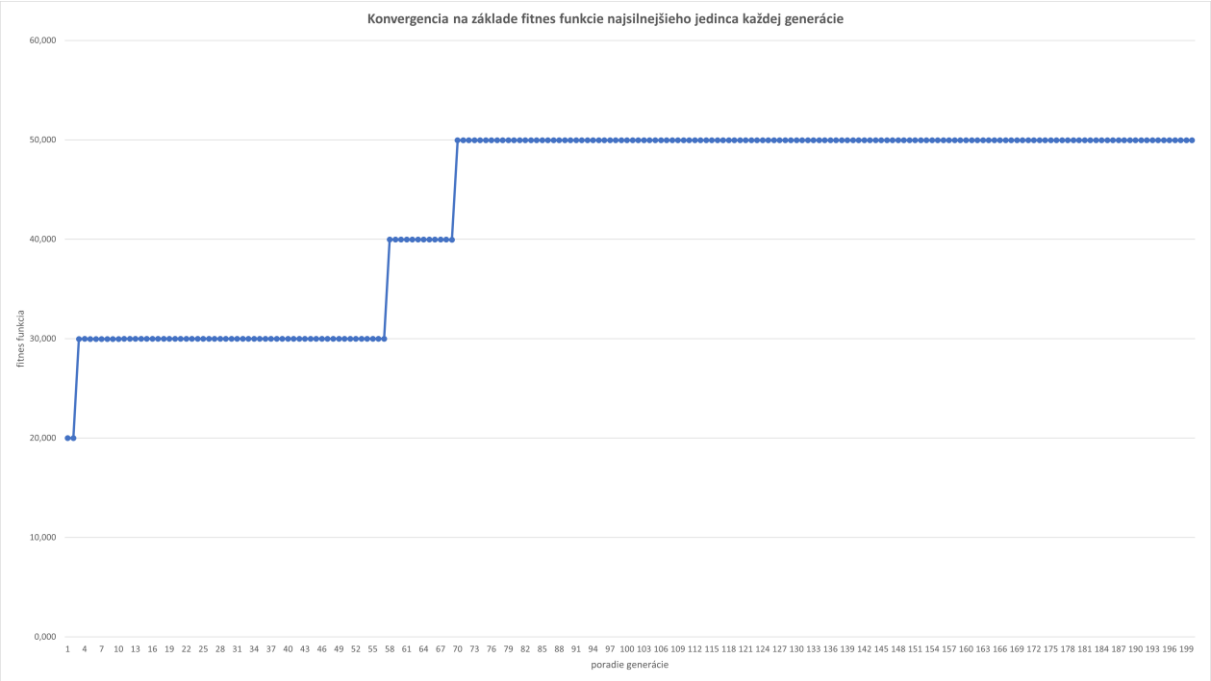
Uživateľ sa stretne s názvom programu, a následne je vyzvaný zadať hodnoty pre algoritmus, alebo vie ponechať tie defaulte. Taktiež si vie vybrať metódu selekcie a po každej iterácií pre daný počet generácií je vyzvaný, či chce pokračovať alebo ukončiť hľadanie riešenia. Po ukončení je vypísané najlepšie riešenie spolu s cestou a fitnes funkciou.

```
*------------------------------------------------------------------------*
|              Hľadanie pokladu - evolučný algoritmus                    |
|                        Autor: Petra Miková                             |
|                          UI, ZS 2023/2024                              |
*------------------------------------------------------------------------*

Nasleduje input hodnôt pre algoritmus, ak chcete ponechať defaultnu hodnotu, stlačte enter:
Zadajte počet pamäťových buniek pre virtuálny stroj (default hodnota: 64):
Zadajte počet pamäťových buniek ktoré sa inicializujú v prvej generácii (default hodnota: 32):
Zadajte počet krokov, po koľkých sa zastaví hľadanie (default hodnota: 500):
Zadajte percento nových jedincov vytvorených elitarizmom (default hodnota: 0.02):
Zadajte percento nových jedincov vytvorených v novej generácii (default hodnota: 0.15):
Zadajte pravdepodobnosť mutácie (default hodnota: 0.05):
Zadajte percento jedincov v turnaji (default hodnota: 0.2):
Zadajte počet generácií (default hodnota: 200):
Zadajte počet jedincov v jednej generácii (default hodnota: 80):
Zadajte metódu selekcie (ruleta(default) / turnaj) (pre ponechanie default stlačte enter):
```

```
Najlepší jedinec pre 398 . generáciu:
Fitnes funkcia:  39.981
Cesta: ['P', 'H', 'L', 'L', 'H', 'L', 'P', 'P', 'P', 'H', 'L', 'L', 'H', 'P', 'P', 'H', 'H', 'P', 'P']
Najlepší jedinec pre 399 . generáciu:
Fitnes funkcia:  39.981
Cesta: ['P', 'H', 'L', 'L', 'H', 'L', 'P', 'P', 'P', 'H', 'L', 'L', 'H', 'P', 'P', 'H', 'H', 'P', 'P']
Najlepší jedinec pre 400 . generáciu:
Fitnes funkcia:  39.981
Cesta: ['P', 'H', 'L', 'L', 'H', 'L', 'P', 'P', 'P', 'H', 'L', 'L', 'H', 'P', 'P', 'H', 'H', 'P', 'P']
Chcete pokračovať v generovaní? (ano/nie): nie

Najlepší jedinec celkovo:
Fitnes funkcia:  39.981
Cesta: ['P', 'H', 'L', 'L', 'H', 'L', 'P', 'P', 'P', 'H', 'L', 'L', 'H', 'P', 'P', 'H', 'H', 'P', 'P']
```

Graf konvergencie (vývoj fitnes funkcie)

Evolučný algoritmus nám po čase začne konvergovať a ustáli sa na jednej finálnej hodnote. Otestovala som svoju implementáciu na 200 generáciach s tým že do grafu som za každú generáciu zadala fitnes funkciu najsilnejšieho jedinca. Tu je výsledok:



Môžeme teda vidieť, že algoritmus eventuálne skončí pri jednom riešení na ktorom konverguje, ktoré považujeme za finálne.

Porovnanie selekcií

Súčasťou zadania bolo aj porovnať dva spôsoby tvorby novej generácie alebo rôzne spôsoby selekcie. Ja som sa teda rozhodla pre dve rôzne selekcie, z ktorých si uživateľ vie vybrať, a to turnaj a ruleta. Spustila som teda program s rovnakým počtom generácií a počtom jedincov v jednej generácii, ako aj rovnakými vstupnými hodnotami.

Petra Miková
ID: 120852

Štatistika pre selekciu turnajom:

Najlepší jedinec celkovo:
Fitnes funkcia: 39.985
Cesta: ['P', 'H', 'L', 'L', 'L', 'H', 'H', 'P', 'H', 'P', 'H', 'P', 'H', 'P', 'P']

Štatistika pre selekciu ruletou:

Najlepší jedinec celkovo:
Fitnes funkcia: 49.976
Cesta: ['P', 'H', 'H', 'H', 'H', 'H', 'L', 'L', 'D', 'D', 'L', 'L', 'P', 'D', 'H', 'P', 'P', 'D', 'H', 'P', 'P', 'D', 'H', 'P']

Môžeme teda vidieť, že ruleta pre daný počet generácií dokázala nájsť oveľa lepšie riešenie.

Zhodnotenie

Mojou úlohou bolo implementovať evolučný algoritmus pre riešenie hľadača pokladov v mriežke za použitia virtuálneho stroja, ktorý simuluje pohyby jedinca. Program úspešne vytvára generácie na základe vstupných hodnôt, využíva elitarizmus, jedince vyberá pomocou selekcie turnajom a ruletou, kríži ich, a mutuje ich pamäťové bunky. Eventálne nám nájde pre daný počet generácií finálne riešenie a môžeme buď dalej generovať, alebo skončiť.

Riešenie som testovala priamo na mriežke zo zadania, a to spôsobom, že som si svoj výstup odkrokovala ručne. Výsledok spĺňa podmienky zadania, teda nevyjde z mriežky, a taktiež nevykoná viac ako 500 krokov.

Zadanie by sa určite dalo rozšíriť ešte napríklad o iné spôsoby mutácie či kríženia, no ja som sa rozhodla v tomto zadaní porovnávať dva spôsoby selekcie. Z môjho pozorovania vzišlo, že úspešnejšia bola ruleta.

Riešenie som implementovala v jazyku Python a celkovo by som zadanie zhrnula ako veľmi zaujímavé, keďže som sa doteraz ešte nestretla s prepojením informatiky s prírodou takýmto spôsobom.