**Projekt 2**

**Hľadanie pokladu (2b)**

**Laura Fulajtárová**

Fakulta informatiky a informačných technológií STU

xfulajtarova@stuba.sk

ID: 120782

UI – Utorok 14:00

31.10.2023

# Contents

[Contents 2](#_Toc149702946)

[1 Parametre môjho počítača 3](#_Toc149702947)

[2 Implementačné prostredie 3](#_Toc149702948)

[3 Riešený problém 3](#_Toc149702949)

[4 Evolučný algoritmus 3](#_Toc149702950)

[5 Reprezentácia údajov 3](#_Toc149702951)

[5.1 Populácia 3](#_Toc149702952)

[5.2 Chromozóm 4](#_Toc149702957)

[5.3 Gén 4](#_Toc149702958)

[6 Virtuálny stroj 4](#_Toc149702959)

[7 Tvorba populácie 5](#_Toc149702960)

[7.1 Prvá populácia 5](#_Toc149702961)

[7.2 Ostatné populácie 5](#_Toc149702962)

[7.3 Selekcia 6](#_Toc149702963)

[7.3.1 Ruleta 6](#_Toc149702964)

[7.3.2 Turnament 6](#_Toc149702965)

[7.4 Reprodukcia 7](#_Toc149702966)

[7.4.1 Crossover 7](#_Toc149702967)

[7.4.2 Mutácia 7](#_Toc149702968)

[8 Používateľské rozhranie 8](#_Toc149702969)

[8.1 Mriežka zo zadania 8](#_Toc149702970)

[8.2 Používateľom vytvorená mriežka 8](#_Toc149702971)

[8.3 Animácia 9](#_Toc149702972)

[9 Zhodnotenie výsledkov 9](#_Toc149702973)

[10 Možné vylepšenia 10](#_Toc149702974)

[11 Použitá literatúra 12](#_Toc149702975)

# Parametre môjho počítača

* Processor AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
* Installed RAM 16.0 GB (13.9 GB usable)
* System type 64-bit operating system, x64-based processor

# Implementačné prostredie

Rozhodla som sa použiť Python ako implementačné prostredie kvôli jeho jednoduchosti, prirodzenému čitateľnému kódu, rozsiahlemu ekosystému knižníc a funkcií, čo umožňuje rýchle a efektívne vývojové práce. Python je ideálny pre implementáciu rôznych algoritmov vrátane evolučných algoritmov, pričom minimalizuje komplikácie spojené s programovaním, takže môžem sústrediť svoju pozornosť na samotný algoritmus a jeho správne fungovanie.

# Riešený problém

Našou hlavnou úlohou bolo vytvoriť hru, v ktorej sme hľadali poklady na dvojrozmernej mriežke. Hľadač sa mohol pohybovať v štyroch základných smeroch: hore, dole, vpravo a vľavo. Naša úloha spočívala v tom, že sme mali obmedzený počet krokov, v ktorých sme sa snažili nájsť čo najviac pokladov. Túto úlohu sme riešili pomocou evolučného programovania nad virtuálnym strojom.

Virtuálny stroj mal za úlohu generovať postupnosť náhodných krokov, ktoré jednotlivec v populácii vykonal. Títo jedinci boli zoskupení do populácie, ktorá sa vyvíjala prostredníctvom selekcie, kríženia, mutácie a elitarizmu. Každého jedinca sme ohodnotili pomocou fitness hodnoty, ktorá nám indikuje, kvalitu jednotlivca.

Program má byť ukončený v prípade, že hľadač našiel všetky poklady alebo keď sme dosiahli maximálny počet generácií.

# Evolučný algoritmus

Evolučný algoritmus predstavuje fascinujúci prístup k riešeniu problémov, ktorý čerpá inšpiráciu z prírody, konkrétne z Darwinovej evolúcie a Mendelovej genetiky. Jeho cieľom je nájsť a zlepšiť najlepšie možné riešenie pre daný problém. Tento proces sa uskutočňuje v niekoľkých krokoch:

1. Inicializácia: Začíname vytvorením prvej populácie náhodných jedincov. Každý jedinec v populácii má svoj chromozóm, ktorý je zložený z génov.
2. Evaluácia: Pre každého jedinca v populácii určíme jeho "fitness" alebo schopnosť riešiť daný problém. Táto hodnota ovplyvňuje, ktorí jedinci budú vybraní pre ďalší vývoj.
3. Možné ukončenie: Po určitom čase alebo po dosiahnutí požadovanej úrovne fitness posúdime, či sme spokojní s populáciou a jedincami. Ak áno, ukončíme algoritmus. V opačnom prípade pokračujeme v ďalšom vývoji populácie.
4. Selekcia: Najlepší jedinci s najvyššou "fitness" prežijú a budú základom pre budúcu generáciu.
5. Variácia: Na vytvorenie novej generácie použijeme týchto lepších jedincov a kombinujeme ich genetický materiál napríklad cez procesy ako crossover a mutácia, čo nám umožňuje objavovať nové možnosti.
6. Návrat k evaluácii: S novou populáciou, ktorá má potenciálne vyššiu "fitness", sa vrátime k evaluácii. Tento krok sa opakuje, kým nedosiahneme požadované výsledky alebo nedosiahneme stanovený maximálny počet generácií.

Evolučný algoritmus nám teda umožňuje postupne zlepšovať riešenia v rámci populácie, pričom sa inšpiruje prírodnými procesmi evolúcie a genetiky, a to všetko s cieľom nájsť čo najlepšie riešenie daného problému.

# Reprezentácia údajov

## Populácia

V rámci evolučného algoritmu je populácia reprezentovaná ako kolekcia jednotlivcov, ktorí sú v tomto kontexte známi ako chromozómy.



## Chromozóm

Chromozóm je jedinečným zástupcom v populácii, a preto mu venujem osobitný objekt, v ktorom uchovávam všetky relevantné informácie. To zahŕňa hodnoty, fitness, počet nájdených pokladov, počet krokov, ktoré vykonal, a informácie o jeho pohybe v rámci mriežky po vykonaní pohybov.

## Gén

Chromozóm je rozdelený na 64 bytov, pričom každých 8 bitov tvorí jeden gén. Tieto gény obsahujú informácie, ktoré určujú vlastnosti a správanie jednotlivca. Gény sú základnými stavebnými jednotkami chromozómov a ovplyvňujú vývoj a výsledky jednotlivcov v populácii.

# Virtuálny stroj

Náš virtuálny stroj disponuje 64 pamäťovými bunkami, pričom každá má veľkosť 1 byte, a rozumie štyrom základným inštrukciám: zvýšenie hodnoty bunky, zníženie hodnoty bunky, skok na konkrétnu adresu a výpis na základe hodnoty bunky. Inštrukcie majú tvar 00XXXXXX (increment), 01XXXXXX (decrement), 10XXXXXX (jump) a 11XXXXXX (write), kde XXXXXX reprezentuje adresu bunky. Výstup inštrukcií závisí od hodnoty bunky: P pre 1-2, H pre 3-4, D pre 5-6 a L pre 7-8. Program okrem výstupu poskytuje aj informácie o pohyboch, ktoré vykonal. Program sa ukončí, ak nájde všetky poklady, vyjde z mriežky alebo vykona 500 krokov. Vytvorí objekt pre daný chromozóm s údajmi ako je počet nájdených pokladov, gény, fitness, kroky a mriežka. Fitness funkcia závisí od počtu nájdených pokladov a vykonaných krokov, je reprezentovaná vzorcom: f=2/(počet krokov +1)+5\*počet nájdených pokladov. Program bol zjednodušený na pseudo program, pre jeho úplné znenie pozrite súbor *game.py*. Pre jednotlivé pohyby na mriežke pozrite súbor *moves.py*.

Obrázok, na ktorom je text, písmo, snímka obrazovky, rad

Automaticky generovaný popis

def virtual\_machine(individual, board, board\_size, treasure\_count):

    board\_copy = board.copy()

    moves\_list = []

    individual\_copy = individual.copy()

    out\_of\_bounds = False

    treasure\_found\_num = 0

    register\_index = 0

    for i in range(500):

        if register\_index <= 63:

            opcode = individual[register\_index][:2]

            register\_value = individual[register\_index][2:]

            if opcode == "00":

                increment

            elif opcode == "01":

                decrement

            elif opcode == "10":

                jump

            elif opcode == "11":

                new\_register\_index = int(register\_value, 2)

                new\_register\_value = individual[new\_register\_index]

                ones\_count = new\_register\_value.count("1")

                if ones\_count <= 2:

                    up

                elif ones\_count <= 4:

                    down

                elif ones\_count <= 6:

                    left

                else:

                   right

                if treasure\_found:

                    treasure\_found\_num += 1

                register\_index += 1

        else:

            register\_index = 0

        if out\_of\_bounds or treasure\_found\_num == treasure\_count:

            break

    fitness = 2 / (len(moves\_list) + 1) + 5 \* treasure\_found\_num

    individual\_object = Individual(

        individual\_copy, fitness, moves\_list, treasure\_found\_num, board\_copy)

    if treasure\_found\_num == treasure\_count:

        global solution\_individual

        solution\_individual = individual\_object

    return individual\_object

# Tvorba populácie

## Prvá populácia

Prvú populáciu vytvárame vytvorením náhodných génov pre určený počet jedincov v populácii. Každý gén je reprezentovaný 64 bytmi. Následne týchto jedincov vkladáme do virtuálneho stroja, ktorý vytvára objekty s relevantnými údajmi pre každého jedinca. Tieto objekty potom ukladáme do spoločného zoznamu, ktorý vytvorí danú populáciu.

def make\_first\_generation(

    board, individual\_count, random\_values\_for\_individuals, board\_size, treasure\_count

):

    generation\_list\_object = []

    global solution\_individual

    for i in range(individual\_count):

        if solution\_individual is None:

            board\_copy = board.copy()

            individual\_values = vm\_create\_random\_values(random\_values\_for\_individuals)

            individual\_object = virtual\_machine(

                individual\_values, board\_copy, board\_size, treasure\_count

            )

            generation\_list\_object.append(individual\_object)

    return generation\_list\_object

## Ostatné populácie

Začíname tým, že zoradíme jedincov v populácii na základe ich fitness hodnôt. Potom z tejto zoradenej populácie vyberáme elitných jedincov, ktorých automaticky prenesieme do novej populácie.

Pre vytvorenie ďalších jedincov využívame selekciu, či už pomocou metódy rulety alebo turnaja. Z týchto vybraných rodičov potom vytvárame nových potomkov pomocou procesu kríženia , kde kombinujeme genetický materiál od oboch rodičov, čím vytvárame nových jedincov.

Títo noví jedinci sú následne podrobení mutáciám s určitou pravdepodobnosťou, čo pridáva variabilitu do populácie. Ak je populácia plná, presunieme ju do ďalšieho kola, v ktorom opakujeme tento proces.

Celý tento cyklus pokračuje, kým nenájdeme optimálne riešenie, alebo nedosiahneme maximálny počet generácií. Týmto spôsobom evolučný algoritmus postupne zlepšuje populáciu a hľadá najlepšie možné riešenia pre daný problém.

while solution\_individual is None and generation\_num < max\_generations:

        generation\_num += 1

        generation\_list\_object.sort(key=lambda x: x.fitness, reverse=True)

        best\_fintness\_individuals.append(generation\_list\_object[0].fitness)

        elite\_individuals = []

        elite\_individuals = copy.deepcopy(

            generation\_list\_object[:elite\_individual\_count]

        )

        subelite\_individuals = []

        if selection\_type == 1:

            subelite\_individuals = roulette\_wheel(

                copy.deepcopy(generation\_list\_object),

                individual\_count,

                elite\_individual\_count,

            )

        else:

            subelite\_individuals = tournament(

                copy.deepcopy(generation\_list\_object),

                individual\_count,

                elite\_individual\_count,

            )

        mutation\_list = []

        mutation\_list = mutation(subelite\_individuals, mutation\_probability)

        subelite\_list\_objects = make\_other\_generations(

            mutation\_list, board\_copy, board\_size, treasure\_count

        )

        population\_list\_object = []

        population\_list\_object = copy.deepcopy(elite\_individuals)

        population\_list\_object.extend(subelite\_list\_objects)

        generation\_list\_object = population\_list\_object

## Selekcia

Selekcia je výber lepších jedincov na základe ich úspešnosti, čím zvyšuje ich šancu na prežitie a ďalšiu generáciu. Pomáha vylepšovať celkovú kvalitu populácie.

### Ruleta

Metóda rulety vyberá jedincov do podpopulácie (subelite). Rozpočítame fitness jedincov, vytvoríme pravdepodobnosti výberu, a na základe nich vyberáme rodičov a tvoríme potomkov. Potomkovia sú pridaní do subpopulácie.

def roulette\_wheel(generation\_list\_object, individual\_count, elite\_individual\_count):

    subelite\_individuals = []

    total\_fitness = sum(individual.fitness for individual in generation\_list\_object)

    selection\_probabilities = [

        individual.fitness / total\_fitness for individual in generation\_list\_object

    ]

    while len(subelite\_individuals) != (individual\_count - elite\_individual\_count):

        parent1 = random.choices(generation\_list\_object, selection\_probabilities)[0]

        parent2 = random.choices(generation\_list\_object, selection\_probabilities)[0]

        first\_child, second\_child = crossover(parent1.value, parent2.value)

        if first\_child not in subelite\_individuals and len(

            subelite\_individuals

        ) + 1 <= (individual\_count - elite\_individual\_count):

            subelite\_individuals.append(first\_child)

        if second\_child not in subelite\_individuals and len(

            subelite\_individuals

        ) + 1 <= (individual\_count - elite\_individual\_count):

            subelite\_individuals.append(second\_child)

    return subelite\_individuals

### Turnament

V prípade funkcie tournament\_winner vytvárame náhodný turnajný zoznam a vyberáme víťaza na základe fitness.

Funkcia tournament vytvára náhodne veľké turnaje, kým nedosiahneme požadovaný počet jedincov pre subpopuláciu. V každom turnaji vyberáme rodičov na základe ich fitness pomocou funkcie tournament\_winner, a títo rodičia sú následne využití na vytvorenie potomkov. Potomkovia sú pridaní do subelite podľa stanovených kritérií.

def tournament\_winner(generation\_list\_object, size\_of\_tournament):

    tournament\_list = []

    for \_ in range(size\_of\_tournament):

        tournament\_list.append(random.choice(generation\_list\_object))

    tournament\_list.sort(key=lambda x: x.fitness, reverse=True)

    winner = tournament\_list[0]

    return winner

def tournament(generation\_list\_object, individual\_count, elite\_individual\_count):

    subelite\_individuals = []

    size\_of\_tournament = random.randint(2, 5)

    while len(subelite\_individuals) != (individual\_count - elite\_individual\_count):

        parent\_1 = tournament\_winner(generation\_list\_object, size\_of\_tournament)

        parent\_2 = tournament\_winner(generation\_list\_object, size\_of\_tournament)

        while parent\_1 == parent\_2:

            parent\_2 = tournament\_winner(generation\_list\_object, size\_of\_tournament)

        parent\_1\_values = parent\_1.value

        parent\_2\_values = parent\_2.value

        first\_child, second\_child = crossover(parent\_1\_values, parent\_2\_values)

        if first\_child not in subelite\_individuals and len(

            subelite\_individuals

        ) + 1 <= (individual\_count - elite\_individual\_count):

            subelite\_individuals.append(first\_child)

        if second\_child not in subelite\_individuals and len(

            subelite\_individuals

        ) + 1 <= (individual\_count - elite\_individual\_count):

            subelite\_individuals.append(second\_child)

    return subelite\_individuals

## Reprodukcia

Reprodukcia v evolučných algoritmoch slúži na vytvorenie nových jedincov v populácii pomocou kríženia (crossover) a mutácie existujúcich jedincov. Týmto spôsobom sa kombinujú genetické vlastnosti a vytvárajú sa potomkovia, čím sa postupne zlepšuje celková kvalita populácie a hľadá najlepšie riešenia pre daný problém.

### Crossover

Táto funkcia dostáva dvoch rodičov ako vstupné parametre a vykonáva kríženie tým spôsobom, že náhodne vyberie index, kde rozdelí genetický materiál oboch rodičov, a potom spojí tieto časti, čím vytvorí nových potomkov.

def crossover(parent1, parent2):

    r\_num = random.randint(1, 64)

    first\_child = parent1[:r\_num] + parent2[r\_num:]

    second\_child = parent2[:r\_num] + parent1[r\_num:]

    return first\_child, second\_child

### Mutácia

Mutácia prebieha s pravdepodobnosťou, ktorú určuje používateľ, a ovplyvňuje zmenu genov a ich počet. Začíname výberom náhodných indexov génov na mutáciu podľa stanovenej pravdepodobnosti. Potom vybrané gény mutujeme pomocou XOR operácie na 3 bitoch. Mutovaných jedincov postupne pridávame do zoznamu a následne ho vrátime.

def mutation(other\_generation\_list, mutation\_probability):

    mutation\_list = []

    for individual in other\_generation\_list:

        for j in range(mutation\_probability):

            cell\_index = random.randint(0, 63)

            cell = individual[cell\_index]

            dec\_byte = int(cell, 2)

            for k in range(3):

                mask = 1 << random.randint(0, 7)

                cell = dec\_byte ^ mask

                dec\_byte = cell

            individual[cell\_index] = bin(cell)[2:].zfill(8)

        mutation\_list.append(individual)

    return mutation\_list

# Používateľské rozhranie

Upravila som používateľské rozhranie vytvorením menu v termináli. Používateľ má teraz možnosť vybrať si, či chce použiť vopred definovanú mriežku zo zadania, vytvoriť vlastnú mriežku alebo ukončiť program. V prípade, že neukončí program, používateľ môže zadať rôzne údaje, ako je počet jedincov v populácii, počet náhodne inicializovaných génov pre jednotlivca, typ selekcie, elitarizmus, mutáciu, maximálny počet generácií a aj to, či chce zobraziť výstupy a jednoduchú animáciu simulácie pohybu hľadača na mriežke. Po vykonaní nám program taktiež zobrazí graf, ktorý opisuje fitness hodnotu najlepšieho jedinca v jednotlivej populácii. Týmto spôsobom sme pridali viac interakcie a prispôsobili program podľa potrieb používateľa.

## Mriežka zo zadania

|  |  |
| --- | --- |
| Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn  Automaticky generovaný popis |  |
| Obrázok, na ktorom je text, displej, snímka obrazovky, štvorec  Automaticky generovaný popis | |

## Používateľom vytvorená mriežka

|  |  |
| --- | --- |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis |
| Obrázok, na ktorom je text, displej, snímka obrazovky, štvorec  Automaticky generovaný popis | |

## Animácia

Obrázok, na ktorom je snímka obrazovky, text, štvorec, pestrofarebnosť

Automaticky generovaný popis Obrázok, na ktorom je štvorec, text, snímka obrazovky

Automaticky generovaný popis

# Zhodnotenie výsledkov

Vykonala som selekciu v populácii dvoma spôsobmi: pomocou metódy rulety a turnaja. Pre každý z týchto dvoch typov selekcie som spustila program 50-krát a získala priemerné výsledky. Zistila som, že tieto dva typy selekcie sa podobajú, a výsledné rozdiely sú skoro zanedbateľné.

Pokiaľ ide o počet nenájdených riešení, oba typy boli na tom rovnako. Avšak, ak sa pozrieme na nájdené riešenia a v akej populácii boli nájdené, vidíme, že turnajová selekcia dosahuje lepšie výsledky. Čo sa týka časovej náročnosti, ruleta trvala o 10 sekúnd dlhšie.

Na grafe sú na osi X uvedené poradové čísla jednotlivých testov a na osi Y je zobrazené, v ktorej generácii bolo riešenie nájdené alebo nenájdené. Nenájdené riešenia sú označené červenou farbou a nájdené zelenou. Pri výpočte priemeru som zahrnula všetky riešenia.

|  |
| --- |
| Obrázok, na ktorom je text, snímka obrazovky, displej, vývoj  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, vývoj, displej  Automaticky generovaný popis |
| Obrázok, na ktorom je text, písmo, snímka obrazovky  Automaticky generovaný popis  s |
| s |

# Možné vylepšenia

V snahe zlepšiť môj kód a optimalizovať ho, začnem identifikovaním miest, kde môžem zvýšiť jeho efektivitu. To znamená preskúmanie existujúceho algoritmu a použitie efektívnejších dátových štruktúr alebo algoritmických postupov. Zároveň zredukujem nepotrebný kód a odstránim zbytočné časti, ktoré neprinášajú hodnotu.

Rovnako dôležité je zvýšenie priestorovej efektivity kódu. Preskúmam, ako sa využíva pamäť, a pokúsim sa znížiť pamäťovú náročnosť tam, kde to bude možné.

Na záver, možnosti rozšírenia programu zahrňujú pridanie ďalších typov selekcií a mutácií. Môžem implementovať rôzne stratégie selekcie, a experimentovať s rôznymi spôsobmi mutácií genetického algoritmu. Týmto spôsobom by sme mohli dosiahnuť lepšie výsledky pri riešení problémov pomocou genetických algoritmov.

# Použitá literatúra

1. <https://www.youtube.com/watch?v=L--IxUH4fac&t=112s&ab_channel=Dr.ShahinRostami>