# SLOVENSKÁ TECHNICKÁ UNIVERZITA

**Fakulta informatiky a informačných technológií**

**v Bratislave**


## Umelá inteligencia – zadanie č.3


## Jakub Taraba


**Prednášajúci:** Ing. Lukáš Kohútka, PhD.

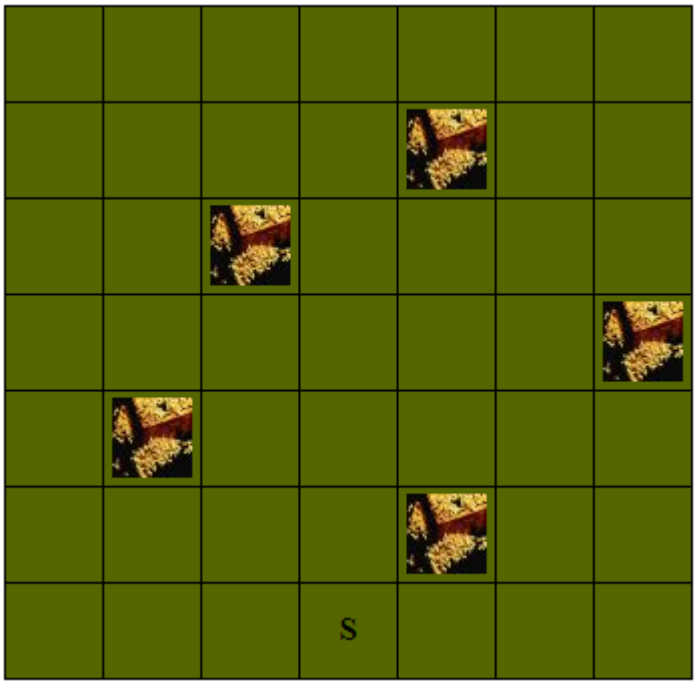
**Cvičiaci:** Ing. Ivan Kapustík

**Čas cvičení:** Štvrtok 14:00

Jakub Taraba
AIS ID: 103159

# Zadanie

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, dole 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.




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 000000.) 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.

Jakub Taraba
AIS ID: 103159

# Reprezentácia hľadača

Hľadač je definovaný triedou Seeker(), ktorá obsahuje genóm (64 bitová pamäťová bunka), kroky, fitness a počet pokladov. Trieda obsahuje aj metódu pre porovnávanie hľadačov.

```
class Seeker():
    def __init__(self, genome, moves, fitness, treasures):
        self.moves = moves
        self.genome = []
        self.genome.extend(genome)
        self.fitness = fitness
        self.treasures = treasures

    def __eq__(self, other):
        if not isinstance(other, Seeker):
            return NotImplemented

        return self.moves == other.moves and self.genome == other.genome and self.
        fitness == other.fitness and self.treasures == other.treasures
```

# Inicializácia prvej generácie

Prvá generácia je inicializovaná tak, že sa vytvorí pole objektov triedy Seeker() a každému objektu sa náhodne vygenerujú hodnoty 0-255 pre každú pamäťovú bunku (gén). Následne sa pomocou virtuálneho stroja určia kroky. Potom sa vypočíta fitness pre každého hľadača a porovnáva sa, či sa našli všetky poklady.

```
def generate_first_population(n_population):
    generation = []

    for i in range(0, n_population):
        genome = create_genome()
        seeker = Seeker(genome, virtual_machine(genome[:]), 0, 0)

        print(seeker.genome, seeker.moves)

        generation.append(seeker)

    return generation
```

Jakub Taraba
AIS ID: 103159

# Tvorba novej generácie:

Nová generácia je vytvorená metódou selekcie turnaj. Následne vzniknú noví hľadači vďaka kríženiu predošlých hľadačov. Niektorí hľadači môžu mutovať, teda zmení sa im genóm a to tak, že sa vygenerujú gény náhodne odznova. Potom sa títo 2 noví jedinci pridajú do novej generácie. Toto sa opakuje dovtedy dokým nie je naplnená nová generácia.

Pokiaľ je zapnuté elitárstvo, do generácie sa vždy dostane N najlepších hľadačov a zvyšok generácie tvoria hľadači, ktorí sú generovaní spôsobom uvedeným vyššie.

```
if ELITISM_COUNT > 0:
    while len(new_generation) != N_POPULATION:
        if len(new_generation) < ELITISM_COUNT:
            elites = elitism(generation, ELITISM_COUNT)
            for elite in elites:
                new_generation.append(elite)
        else:
            seekers = tournament_start(generation, 3)
            new_seekers = crossover(seekers[0], seekers[1])
            mutate(new_seekers[0])
            mutate(new_seekers[1])

            if len(new_generation) != N_POPULATION-1:
                new_generation.append(new_seekers[0])
                new_generation.append(new_seekers[1])
            else:
                new_generation.append(new_seekers[0])
else:
    seekers = tournament_start(generation, 3)
    new_children = crossover(seekers[0], seekers[1])
    mutate(new_children[0])
    mutate(new_children[1])
    new_generation.append(new_children[0])
    new_generation.append(new_children[1])

generation.clear()
generation.extend(new_generation)
```

# Virtuálny stroj

Virtuálny stroj hľadačovi pokladu určí kroky na základe jeho genómu. Opakuje sa 500 inštrukcií, ktoré môžu inkrementovať gén/pamäťovú bunku, dekrementovať gén/pamäťovú bunku, skok na gén alebo pridať nový krok.

Kroky sa určujú podľa posledných 2 bitov. Pokiaľ je hodnota posledných 2 bitov 0 krok je H, 1 krok je D, 2 krok je L, 3 krok je R.

Jakub Taraba
AIS ID: 103159

# Spôsob selekcie

Program obsahuje 1 spôsob selekcie a to konkrétne **turnaj**, ktorý funguje spôsobom, že sa náhodne vybere **k** hľadačov z generácie. Z týchto **k** hľadačov sa vyberie taký, ktorý má najväčšiu hodnotu fitness. Ošetril som aj prípad, aby sa nestalo, že vyberiem 2 rovnákých rodičov.

```
def tournament(generation, k):
    best_player = None

    for i in range(k):
        player = random.choice(generation)
        if best_player is None or player.fitness > best_player.fitness:
            best_player = player

    return best_player
```

# Kríženie

Pri krížení sa zoberú 2 hľadači, určí sa „zlomový bod“, od ktorého si začnú hľadači navzájom vymieňať svoje gény. Zlomový bod je určený náhodne. Napr. ak by bol zlomový bod 32, hľadač 1 by obsahoval 32 svojich génov a 32 génov hľadača 2 a hľadač 2 opačne.

```
def crossover(seeker_1, seeker_2):
    crossover_point = random.randint(0, 63)

    genome_1 = []
    genome_2 = []

    for i in range(64):
        if i >= crossover_point:
            genome_1.append(  (parameter) seeker_1: Any
            genome_2.append(seeker_1.genome[i])
        else:
            genome_1.append(seeker_1.genome[i])
            genome_2.append(seeker_2.genome[i])

    seeker_parent_1 = Seeker(genome_1, virtual_machine(genome_1[:]), 0, 0)
    seeker_parent_2 = Seeker(genome_2, virtual_machine(genome_2[:]), 0, 0)

    return seeker_parent_1, seeker_parent_2
```

# Mutácia

Existuje % šanca, že sa hľadačovi pokľadu vyresetujú gény a nahradia sa novými, náhodne určenými génmi v rozsahu 0-255. Ak by program neobsahoval mutáciu, vývoj hľadačov v generácii by nekonvergoval.

Jakub Taraba
AIS ID: 103159

# Fitness

Počítanie hodnoty fitness funguje tak, že sa od počtu nájdených pokladov odpočíta počet krokov v tisícinách.

```
if seeker.treasures > 0:
    seeker.fitness = seeker.treasures - (len(solution)/1000)
else:
    seeker.fitness -= (len(solution)/1000)
```

# Výsledky

Výstup programu a čas vykonávania programu veľmi záležal na náhode. Boli prípady kedy sa našlo riešenie do 300. generácie a niekedy sa nedokázalo nájsť riešenie ani do 3000. generácie pri 20 hľadačoch v populácii. Čím viac hľadačov v populácii, tým mi program častejšie našiel správne riešenie za kratší čas. Generácie mohli byť aj dosť ovplyvnené elitárstvom, kedy hodnota fitness rástla rýchlejšie pri použití elitárstva.

# Zhodnotenie

To ako program rýchlo nájde riešenie závisí od vstupných hodnôt, počtu pokladov a počtu hľadačov v generácii. Program s použitím elitárstva dosahuje lepšie výsledky z toho dôvodu, že sa do ďalšej generácie dostane N najlepších hľadačov. Nie je vhodné zadávať malý počet hľadačov v generácii, pretože potom je čas hľadania riešenia podstatne dlhší.