UI Zadanie 2 – Evolučný algoritmus

Adam Gábor
AIS ID: 116174
cvičenie: streda 13:00
cvičiaci: Ing. Martin Komák, PhD.

Obsah:

1. Znenie zadania

Úloha

Zenová záhradka je plocha vysypaná hrubším pieskom (drobnými kamienkami). Obsahuje však aj nepohyblivé väčšie objekty, ako napríklad kamene, sochy, konštrukcie, samorasty. Mních má upraviť piesok v záhradke pomocou hrablí tak, že vzniknú pásy.

Pásy môžu ísť len vodorovne alebo zvislo, nikdy nie šikmo. Začína vždy na okraji záhradky a ťahá rovný pás až po druhý okraj alebo po prekážku. Na okraji – mimo záhradky môže chodiť ako chce. Ak však príde k prekážke – kameňu alebo už pohrabanému piesku – musí sa otočiť, ak má kam. Ak má voľné smery vľavo aj vpravo, je jeho vec, kam sa otočí. Ak má voľný len jeden smer, otočí sa tam. Ak sa nemá kam otočiť, je koniec hry. Úspešná hra je taká, v ktorej mních dokáže za daných pravidiel pohrabať celú záhradu, prípade maximálny možný počet políčok. Výstupom je pokrytie danej záhrady prechodmi mnícha. Pokrytie zodpovedajúce presne prvému obrázku (priebežný stav) je napríklad takéto:

Úlohu je možné rozšíriť tak, že mních navyše zbiera popadané lístie. Listy musí zbierať v poradí: najprv žlté, potom pomarančové a nakoniec červené. Príklad vidno na obrázku nižšie. Listy, ktoré zatiaľ nemôže zbierať, predstavujú pevnú prekážku. Fitness funkcia ostáva rovnaká.

Zadanie

Uvedenú úlohu riešte pomocou evolučného algoritmu. (Je možné použiť aj ďalšie algoritmy, ako sú uvedené v probléme obchodného cestujúceho.) Maximálny počet génov nesmie presiahnuť polovicu obvodu záhrady plus počet kameňov, v našom príklade podľa prvého obrázku 12+10+6=28. Fitnes je určená počtom pohrabaných políčok. Výstupom je matica, znázorňujúca cesty mnícha. Je potrebné, aby program zvládal aspoň záhradku podľa prvého obrázku, ale vstupom môže byť v princípe ľubovoľná mapa.

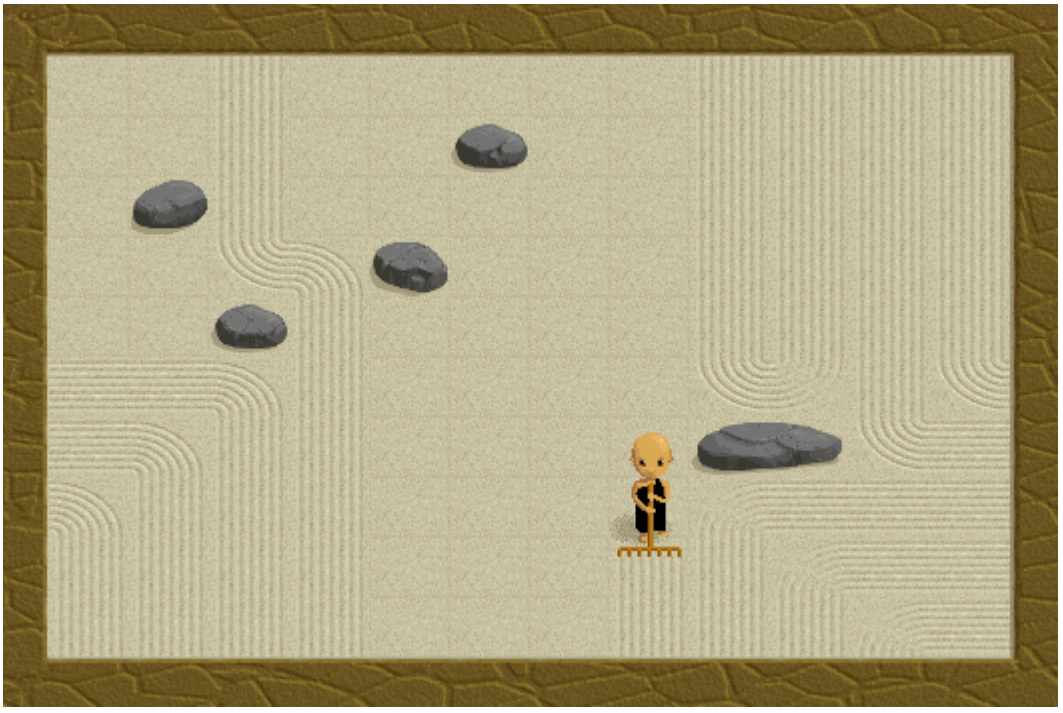
Celé zadanie dostupné na: http://www2.fiit.stuba.sk/~kapustik/zen.html

2. Realizácia projektu

Projekt som sa rozhodol realizovať v programovacom jazyku Python. Súborovú štruktúru tvorí iba súbor main.py, ktorý obsahuje všetky potrebné funkcie. Testovanie implementovaných funkcií prebieha v programe PyCharm 2023.2 na mojom laptope - MSI GF63 s 16GB RAM a procesorom 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz.

3. Logická hádanka Zenová záhrada

Princíp Zenovej záhrady je dobre opísaný už v zadaní, pre správne programovanie funkcionality hádanky je však dôležité špecifikovať, kedy sa hra končí. Úspešný koniec hry nastáva v prípade, keď mních "pohrabal" všetky hracie políčka, a neúspešne hra končí vtedy, keď sa mních na niektorom z políčok zasekne a otočenia o 90 stupňov do oboch strán vedú iba k tomu, že mních čelí ďalšej prekážke. Pre všetky ťahy platí, že mních vstúpi do záhrady z ľubovoľnej strany a ľubovoľného voľného políčka, ďalej sa pohybuje iba po priamkach a po voľných políčkach (prekážku predstavujú jednak kamene a druhak už pohrabané políčka) a zo záhrady do ľubovoľnej strany aj vystúpi (okraj záhrady sa teda za prekážku nepovažuje a mních sa tam vždy môže otočiť a spraviť krok).



Obr. 1 - záhrada v hre Zen Puzzle Game

Zenová záhrada v mojej implementácii môže mať ľubovoľné rozmery od 1x1 až do 16x16 políčok a ľubovoľný počet kameňov (označených “X”), pričom platí, že počet kameňov < počet políčok. Hráč môže kamene rozmiestniť na ľubovoľnú pozíciu v záhrade. Nepohrabané políčka sú označené číslom 0, každý ďalší ťah a všetky ním pohrabané políčka sú označené číslom ťahu.

```
0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  X  0  0  0  0  0  0
0  X  0  0  0  0  0  0  0  0  0  0
0  0  0  0  X  0  0  0  0  0  0  0
0  0  X  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  X  X  0  0
0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0
```

Obr. 2 - výpis reprezentácie záhrady z Obr. 1 v jazyku Python

4. Genetický algoritmus

Genetický algoritmus je nedeterministický spôsob riešenia ťažko riešiteľných problémov, medzi ktoré patrí aj nájdenie optimálnej, úspešne ukončenej hry Zenovej záhrady. Algoritmus vychádza z Darwinovej teórie evolúcie, podľa ktorej prežívajú a množia sa práve najsilnejší jedinci.

V mojej implementácii genetického algoritmu sú takíto jedinci reprezentáciou jednotlivých hier. Hry, ktoré sú bližšie k vyriešeniu, sa v nasledujúcich generáciách množia a mutujú (jemne modifikujú, skúšajú iné cesty). Hry, ktoré problém riešia horšie, sa nemnožia. Jednotlivci (kompletné hry) sú tvorení chromozómami (ťahy v hre) a chromozómy sú tvorené génmi (jednotlivé inštrukcie, podľa ktorých sú ťahy vykonávané).

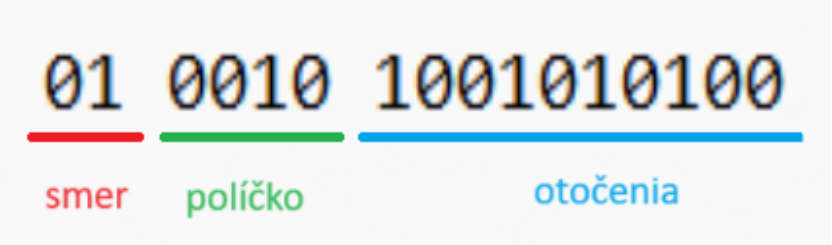
Zo svojej podstaty genetický algoritmus nehľadá maximum, ale optimum - dostatočne dobré riešenie čo najbližšie k užívateľom popísanému ideálnemu riešeniu. V prípade Zenovej záhrady vieme určiť, či sa optimum rovná maximu a ako veľmi sa mu približuje.

4.1. Gén

Keďže je pre správny chod programu dôležitá jeho rýchlosť, je veľmi dôležité, aby boli chromozómy aj jednotlivci implementovaní čo najjednoduchším spôsobom. Preto som si aj pre svoj program zvolil veľmi bežný princíp genetického algoritmu - reprezentáciu génov v bitoch. Každý krok viem opísať sériou šestnástich po sebe idúcich bitov.

Do prvých dvoch bitov viem štyrmi spôsobmi zakódovať smer, ktorým mních pôjde (presnejšie stranu, z ktorej do záhrady vstúpi). Ďalšie štyri bity reprezentujú index poľa vstupného riadka/stĺpca od 0 do 15. Práve tieto štyri bity limitujú maximálny rozmer záhrady 16x16. V prípade potreby a rozšírenia génov udávajúcich políčko napríklad na 6 bitov by už záhrada mohla byť veľká do rozmeru 64x64 políčok. Zvyšných desať bitov reprezentuje rozhodnutie chromozómu pri narazení na prekážku. Jednotka znamená otočenie po pravej ruke, nula otočenie po ľavej ruke. Program ráta s tým, že jeden ťah viac ako desať otočení nevykoná. V praxi je týchto otočení väčšinou vždy menej ako šesť, takže by na zakódovanie jedného ťahu stačilo už aj dvanásť génov.

Tento šestnásť génový model (fungujúci pre záhrady do veľkosti 16x16) je plne postačujúci požiadavke zadania, ktorá hovorí o maximálnom počte génov rovnému strane x + strane y + počtu kameňov v záhrade (v prípade testovacej záhrady 28 génov).



Obr. 3 - 16 bitov (génov) tvoriacich jeden chromozóm

4.2. Chromozóm

Chromozóm tvorený z bitov predstavuje jeden ťah mnícha hrabľami po záhrade. Chromozómy sú testovacou funkciou vykonávané po jednom, pretože záleží aj na ich poradí. V prípade, že počas ťahu mních narazí na prekážku, hýbe sa pomocou bitov jeho otočení. Počet chromozómov (ťahov) v jedincovi (hre) je náhodný a genetický algoritmus ho s pribúdajúcim časom optimalizuje - ťahy môže pridávať aj odoberať v závislosti od potreby riešenia.

Pole n-veľkosti šestnásť bitových chromozómov tvorí jedinca. Keďže nemusí mať každá záhrada 16x16 políčok a pri každom ťahu nemusí mních spraviť desať otočení, niektoré gény v chromozóme ostávajú nepoužité a niektoré chromozómy sú neplatné. Jednou z úloh evolúcie je takéto neplatné ťahy postupne mutáciou eliminovať.

```
010111 Left - 7
0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   X   0   0   0   0   0
0   X   0   0   0   0   0   0   0   0   0
0   0   0   0   X   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   X   X   0
1   1   1   1   1   1   1   1   1   1   1
0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0
```

Obr. 4 - reprezentácia chromozómu (ťahu) v matici záhrady - bity 01 vyjadrujú smer (zľava doprava) a bity 0111 vyjadrujú index počiatočného políčka (index 7, teda 8. riadok); ťah je vykonaný kontinuálne, pretože mních nenarazil na žiadnu prekážku.

4.3. Jedinec a populácia

Počiatočná populácia (nultá generácia algoritmu) je tvorená kompletne náhodne pomocou funkcií **random_individual()**, **random_chromosome()** a neskôr preložená z bitov do inštrukcií pomocou funkcie **chromosome_to_direction()**.

Veľkosť populácie, ktorú som pre riešenie problému zvolil, je 32. Vplyv veľkosti populácie na rýchlosť a výstupy programu neskôr testujem v šiestej kapitole tohto dokumentu.

```python
#genetic algorithm properties
population_size = 32
chromosome_length = 16
population = [random_individual(chromosome_length) for _ in range(population_size)]
max_generations = 100
```

```python
# GENETIC ALGORITHM FUNCTIONS
2 usages
def random_chromosome():
    return format(random.randint( a: 0, (1 << 16) - 1), '016b')

def chromosome_to_direction(chromo):
    sides = ['Top', 'Left', 'Bottom', 'Right']
    side = sides[int(chromo[:2], 2)]
    position = int(chromo[2:6], 2)
    turns = chromo[6:]
    return f"{side} - {position} - {turns}"

1 usage
def random_individual(max_moves):
    return [random_chromosome() for _ in range(random.randint( a: 1, max_moves))]
```

Obr. 5 - vytvorenie počiatočnej populácie a jej naplnenie jedincami - tí sú naplnení chromozómami - kompletne náhodnými reťazcami génov

4.4. Fitness funkcia

Fitness funkcia je funkcia, ktorej úlohou je ohodnotiť jedinca. Lepšie hodnotení jedinci sú bližšie k optimálnemu riešeniu a majú vyššiu šancu sa množiť. Naopak, horšie hodnotení jedinci neposkytujú správne riešenie a do ďalších generácií sa nedostanú.

Hodnota fitness funkcie je súčet políčok, ktoré boli úspešne pohrabané. Aby sa políčko rátalo ako úspešne pohrabané, musí po ňom mních prejsť v rámci legálneho ťahu (teda ťahu spĺňajúceho všetky kritériá hry, vyhýbajúceho sa prekážkam, začatého a končiaceho na okraji záhrady atď.). Aby bol jednotlivec ohodnotený fitness hodnotou, musí jeho hra skončiť buď úspešným pohrabaním všetkých políčok, alebo nemožnosťou vykonať ďalší legálny ťah (napr. vyčerpaním chromozómov alebo zablokovaním ďalších ťahov).

Fitness v prípade môjho programu zisťujú tri funkcie, **test_individual()**, ktorá kontroluje každého jednotlivca v populácii, **make_move()**, ktorá číta chromozómy a vykonáva kroky, pričom kontroluje, či sú legálne a zapisuje ich do matice záhrady, a **calculate_fitness()**, ktorá po ukončení hry sčíta pohrabané políčka. Fitness hodnota hry, ktorá sa skončí uviaznutím mnícha uprostred záhrady, je nezáležiac od počtu pohrabaných políčok nulová.

V prípade zložitejšej implementácie riešenia problému by mohla fitness funkcia okrem počtu pohrabaných políčok hodnotiť aj počet ťahov, na ktorý boli tieto políčka pohrabané. Populácia by tak konvergovala k riešeniam s čo najnižším počtom chromozómov.

```
def test_individual(individual, garden):
    local_garden = [row.copy() for row in garden]
    move_num = 1
    for chromo in individual:
        local_garden, fitness, game_over = make_move(local_garden, chromo, move_num)
        move_num += 1

        if game_over:
            return 0, local_garden  #return fitness 0 if game ended with monk trapped

    #return fitness value if ran out of chromosomes / game ended
    return fitness, local_garden
```

```
1 usage
def make_move(garden, chromo, move_num):
    rows, cols = len(garden), len(garden[0])
    direction = chromo[:2]
    position = int(chromo[2:6], 2)
    turning_bits = list(chromo[6:])

    directions = {
        "00": (1, 0),
        "01": (0, 1),
        "10": (-1, 0),
        "11": (0, -1)
    }

    turns = {
        "00": ["01", "11"],
        "01": ["00", "10"],
        "10": ["01", "11"],
        "11": ["00", "10"]
    }

    dx, dy = directions[direction]
    x, y = (0 if direction == "00" else rows - 1 if direction == "10" else position,
            position if direction == "00" or direction == "10" else 0 if direction == "01" else cols - 1)

    while 0 <= x < rows and 0 <= y < cols and turning_bits:
        next_x, next_y = x + dx, y + dy
```

```
    while 0 <= x < rows and 0 <= y < cols and turning_bits:
        next_x, next_y = x + dx, y + dy

        if garden[x][y] != '0': #if obstacle in front
            break

        if not (0 <= next_x < rows and 0 <= next_y < cols):  #garden boundaries
            garden[x][y] = str(move_num)
            break

        #if next step is valid & not occupied
        if garden[next_x][next_y] == '0':
            garden[x][y] = str(move_num)
            x, y = next_x, next_y
            continue

        #if obstacle, turn
        possible_turns_inside = [dir for dir in turns[direction] if
                                 0 <= x + directions[dir][0] < rows and
                                 0 <= y + directions[dir][1] < cols and
                                 garden[x + directions[dir][0]][y + directions[dir][1]] == '0']

        possible_turns_outside = [dir for dir in turns[direction] if
                                  not (0 <= x + directions[dir][0] < rows
                                       and 0 <= y + directions[dir][1] < cols)]
```

```
            #read turn bit, turn there
            turn_bit = turning_bits.pop(0)
            if direction in ["00", "10"]:
                new_direction = "11" if turn_bit == "1" else "01"
            else:
                new_direction = "00" if turn_bit == "1" else "10"

            if new_direction in possible_turns_inside: #moving inside the garden
                direction = new_direction
            elif possible_turns_inside:
                direction = possible_turns_inside[0]
            elif new_direction in possible_turns_outside: #moving outside the garden
                direction = new_direction
            elif possible_turns_outside:
                direction = possible_turns_outside[0]
            else:  #if no turns
                garden[x][y] = str(move_num)
                fitness = calculate_fitness(garden)
                return garden, fitness, True  #game over

            dx, dy = directions[direction]

        if 0 <= x < rows and 0 <= y < cols and garden[x][y] == '0':
            garden[x][y] = str(move_num)

        fitness = calculate_fitness(garden)
        return garden, fitness, False
```

```
def calculate_fitness(garden):
    return sum(1 for row in garden for cell in row if cell != '0' and cell != 'X')
```

Obr. 6. - funkcie pre simulovanie hry - vstupom je jedinec, čítaním jeho chromozómov sa vykonávajú ťahy v záhrade a výstupom je fitness hodnota jednotlivca

5. Evolúcia

Jedna populácia tvorená čisto náhodne vytvorenými jedincami riešenie hádanky pochopiteľne nenájde. Genetický algoritmus funguje na princípe evolúcie, a tú má v mojom programe na starosti funkcia **selection()** a jej pridružené pomocné funkcie. Rozhodol som sa pre výber jedincov princípom elitizmu a vylúčením členov s extrémnymi hodnotami, a následne turnajom medzi ostatnými členmi. Víťazi turnaja potom krížia svoje gény metódou prekríženia v jednom bode (one-point crossing) a vytvoria potomkov. Potomkovia majú ďalej šancu mutovať.

Elitizmus spoločne s vylúčením najhorších jedincov bol pri výbere selekcie jasná voľba, pretože je pre mňa dôležité uchovávať si n-počet najlepších a takisto sa zbaviť n-počtu najhorších jedincov (napríklad hier, ktoré sa končia uviaznutím mnícha v záhrade). Pre výber turnajovou metódou som sa rozhodol na úkor rulety, pretože mi pre testovacie účely príde modifikácia veľkosti a počtu turnajov príde jednoduchšia a škálovateľnejšia ako v prípade rulety, ktorá sa prakticky nijako nemodifikuje a nedokáže reagovať na potreby mojej implementácie.

```python
def selection(population, mutation_rate):
    n_elites = 4
    n_remove = 4
    n_tournaments = 8
    tournament_size = 3

    #elitism
    elites = elitism_selection(population, n_elites)

    #elite and bottoms doesnt enter tournament
    reduced_population = sorted(population, key=lambda x:\
        test_individual(x, original_garden)[0], reverse=True)[n_elites:-n_remove]


    #tournament
    tournament_winners = tournament_selection(reduced_population, n_tournaments, tournament_size)
```

```python
    #one-point crossover
    offspring = []
    for i in range(0, len(tournament_winners), 2):
        offspring1, offspring2 = one_point_crossover(tournament_winners[i], tournament_winners[i + 1])
        offspring.append(offspring1)
        offspring.append(offspring2)

    #mutation
    for i in range(len(offspring)):
        if random.random() < mutation_rate:  # Use the dynamic mutation rate
            offspring[i] = mutate(offspring[i])

    #new generation
    new_generation = elites + offspring + reduced_population\
        [:population_size - n_elites - len(offspring) - n_remove]


    return new_generation
```

Obr. 7 - funkcia selection(), ktorá najprv vykoná výber elitizmom a vylúči najhorších jedincov, potom vyberie ostatných pomocou turnaja, z víťazov turnaja vytvorí nových potomkov a zmutuje ich - vstupom je súčasná generácia a výstupom je ďalšia

5.1. Selekcia elitizmom

Selekcia elitizmom, ktorej parameter - počet jedincov, ktorí sa pre ďalšiu generáciu zachovajú - je definovaný v hlavičke funkcie **selection()**, je vykonávaná jednoduchou funkciou **elitism_selection()** využívajúcou lambda výraz. Táto funkcia zoradí jedincov v generácii podľa fitness hodnoty a vráti zoradené pole generácie. Z tohto poľa je potom vrchných n členov posunutých do ďalšej generácie bez zmeny a spodných n členov vyhodených.

```python
def elitism_selection(population, n_elites):
    sorted_population = sorted(population, key=lambda x: test_individual(x, original_garden)[0], reverse=True)
    return sorted_population[:n_elites]
```

Obr. 8 - funkcia elitism_selection()

5.2. Selekcia turnajom

Do turnaja vstupujú všetci jedinci, ktorí neboli posunutí do ďalšej generácie vďaka elitizmu alebo vyradení kvôli nedostatočnému fitness skóre. Princíp turnaja spočíva v tom, že sa náhodne vyberie n jedincov z populácie, a víťazom turnaja je zvolený ten jedinec, ktorý má maximálnu fitness hodnotu. V turnaji súperia v mojej implementácii najčastejšie traja jedinci, formát turnaja je bližšie špecifikovaný v kapitole 6.2.2, v ktorej s ním aj experimentujem.

Výsledkom turnaja je n víťazov, ktorí sa ďalej krížia - vytvárajú sa z nich noví jedinci.

```
def tournament_selection(population, n_tournaments, tournament_size):
    winners = []
    for _ in range(n_tournaments):
        tournament = random.sample(population, tournament_size)
        winner = max(tournament, key=lambda x: test_individual(x, original_garden)[0])
        winners.append(winner)
    return winners
```

Obr. 9 - funkcia tournament_selection()

5.3. Kríženie a mutácia

Kríženie jedincov prebieha metódou prekríženia v jednom bode (one-point crossing). Tento typ kríženia bol zvolený pre jeho jednoduchosť. Vo všeobecnosti nie je jednoduché kríženie pri riešení hlavolamov, ktoré si vyžadujú určitú postupnosť krokov, dobrým nápadom, avšak v tomto prípade je kríženie (v jednom bode) prospešné. Je to kvôli tomu, že takmer každé optimálne riešenie záhradu delí na sektory. Jedným ťahom sa dá záhrada rozdeliť na dva menšie sektory, ktoré sú od seba nezávislé a teda poradie, v ktorom sa políčka v sektoroch hrabú, či dokonca poradie jednotlivých ťahov v sektore výsledok riešenia hádanky ovplyvní iba minimálne. V prípade, že má napríklad nejaký jedinec výborné riešenie jedného sektoru záhrady, ale problém s riešením iných sektorov, kríženie dokáže zabezpečiť, že sa silná časť riešenia ponechá a vyskúša sa k nej kompletne iná časť riešenia zvyšných sektorov v záhrade.

```python
def one_point_crossover(parent1, parent2):
    if len(parent1) <= 1 or len(parent2) <= 1:
        return parent1, parent2  #if parent has no or 1 chromosome, return the parent

    crossover_point = random.randint( a: 1, len(parent1) - 1)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    return offspring1, offspring2
```

Obr. 10 - funkcia one_point_crossover(), ktorá implementuje kríženie v jednom bode a vracia dvoch potomkov

Novovytvorení jedinci sú potom poslaní do funkcie **mutate()**, ktorá má za úlohu ešte viac diferencovať ich chromozómy. Môj program rozlišuje tri druhy mutácie, ktoré nie sú vzájomne exkluzívne a môže nastať 1 až 3 z nich. Ide o náhodné pridanie n-počtu chromozómov, náhodné odobratie n-počtu chromozómov a a náhodné preklopenie n-počtu génov (bitov) v časti chromozómu, ktorá je zodpovedná za zmenu strany pri narazení na prekážku.

```python
def mutate(individual):
    num_mutations = random.randint( a: 1,  b: 3)  #no of mutations

    for _ in range(num_mutations):
        mutation_type = random.choice(['toggle_turn_bit', 'add_move', 'remove_move'])

        if mutation_type == 'toggle_turn_bit':
            chromosome = random.choice(individual)
            idx = individual.index(chromosome)
            num_bits_to_toggle = random.randint( a: 1,  b: 4)  #change 1 to 4 bits
            chromosome_list = list(chromosome)
            for _ in range(num_bits_to_toggle):
                turn_bit_idx = random.randint( a: 6,  b: 15)
                chromosome_list[turn_bit_idx] = '1' if chromosome_list[turn_bit_idx] == '0' else '0'
            individual[idx] = ''.join(chromosome_list)

        elif mutation_type == 'add_move':
            num_moves_to_add = random.randint( a: 1,  b: 4)  #add 1 to 4 moves
            for _ in range(num_moves_to_add):
                individual.append(random_chromosome())

        else:  # 'remove_move'
            num_moves_to_remove = random.randint( a: 1,  b: 4)  #remove 1 to 4 moves
            for _ in range(num_moves_to_remove):
                if len(individual) > 1:
                    individual.remove(random.choice(individual))

    return individual
```

Obr. 11 - funkcia mutate()

5.3.1. Adaptívna mutácia

Zaujímavý koncept využitý v mojom kóde, vďaka ktorému sa môže vytváranie nových generácií prispôsobovať aktuálnemu vývoju hľadania riešenia, je adaptívna mutácia. Šanca, s akou môžu noví jedinci mutovať, je od 0.05 do 0.25 (alebo v rozmedzí akýchkoľvek ďalších hodnôt, ďalej spomínané v kapitole 6.2.3.) v závislosti od toho, ako veľmi funkcia konverguje = ako veľmi sa posledná generácia líšila od tých pred ňou. V prípade, že evolúcia stagnuje, sa miera mutácie zvýši a vďaka tomu sa do populácie dostanú noví jedinci s novými, potenciálne lepšími možnosťami riešenia problému.

```
#adaptive mutation properties
prev_avg_fitness = 0
mutation_rate = 0.05
max_mutation_rate = 0.25
min_mutation_rate = 0.05

#making generations
best_individual = None
best_fitness = 0

for generation in range(max_generations):
    fitnesses = [test_individual(individual, original_garden)[0] for individual in population]
    avg_fitness = sum(fitnesses) / len(fitnesses)

    #get mutation rate
    if avg_fitness <= prev_avg_fitness:
        mutation_rate += 0.05
        mutation_rate = min(mutation_rate, max_mutation_rate)
    else:
        mutation_rate = min_mutation_rate  #reset to minimum if evolution happened

    prev_avg_fitness = avg_fitness
```

Obr. 12 - implementácia adaptívnej mutácie

6. Testovanie algoritmu

Po otestovaní správneho fungovania všetkých komponentov programu som testoval účinnosť nastavenia jednotlivých prvkov genetického algoritmu a to, akým spôsobom ovplyvňujú riešenie problému. Všetky testy boli vykonávané na záhrade zo vzorového príkladu. Pre testovanie správnej adaptability algoritmu som ale vyskúšal riešiť aj prázdnu záhradu bez kameňov či záhradu s inak/náhodne rozloženými kameňmi, ako aj záhrady rôznych rozmerov.

```
Best Garden from the last generation:
12  12  9   9   2   4   7   8   15  15  8   7
12  12  9   9   2   4   7   8   15  15  8   7
12  12  9   9   2   4   7   8   15  15  8   7
12  12  9   9   2   4   7   8   15  15  8   7
12  12  9   9   2   4   7   8   8   8   8   7
12  12  9   9   2   4   7   7   7   7   7   7
12  12  9   9   2   4   4   4   4   4   4   4
12  12  9   9   2   2   2   2   2   2   2   2
1   1   1   1   1   1   1   1   1   1   1   1
5   5   5   5   5   5   5   13  13  13  13  13
```

Obr. 13 - riešenie pre záhradu rozmeru 10x12 bez kameňov - s "defaultnými" nastaveniami genetického algoritmu trvalo nájdenie úplného riešenia v priemere približne 150-200 generácií

```
Best Garden from the last generation:
26  24  5   14  17  0   17  14  1   0   3   13  7   2   4   10
24  24  5   14  17  X   17  14  1   0   3   13  7   2   4   10
0   X   5   14  17  17  17  14  1   0   3   13  7   2   4   10
5   5   5   14  X   0   0   14  1   0   3   13  7   2   4   10
0   0   X   14  14  14  14  14  1   0   3   13  7   2   4   10
1   1   1   1   1   1   1   1   1   0   3   13  7   2   4   10
11  11  11  9   9   8   8   8   X   X   3   13  7   2   4   10
20  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
30  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
30  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
21  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
21  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
21  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
20  20  11  9   9   8   0   8   6   6   3   13  7   2   4   10
11  11  11  9   9   8   0   8   6   6   3   13  7   2   4   10
12  12  12  9   9   8   0   8   6   6   3   13  7   2   4   10
```

Obr. 14 - nájdenie optima pre záhradu rozmeru 16x16 s kameňmi na pozícii zo vzorového príkladu - po 500 generáciách je algoritmus schopný pokryť približne 230 z 250 políčok

```
Best Garden from the last generation:
9   19  19  8   2   12
9   X   8   8   2   12
8   8   8   X   2   2
3   3   3   3   3   3
7   7   7   7   7   7
11  11  11  11  11  11
```

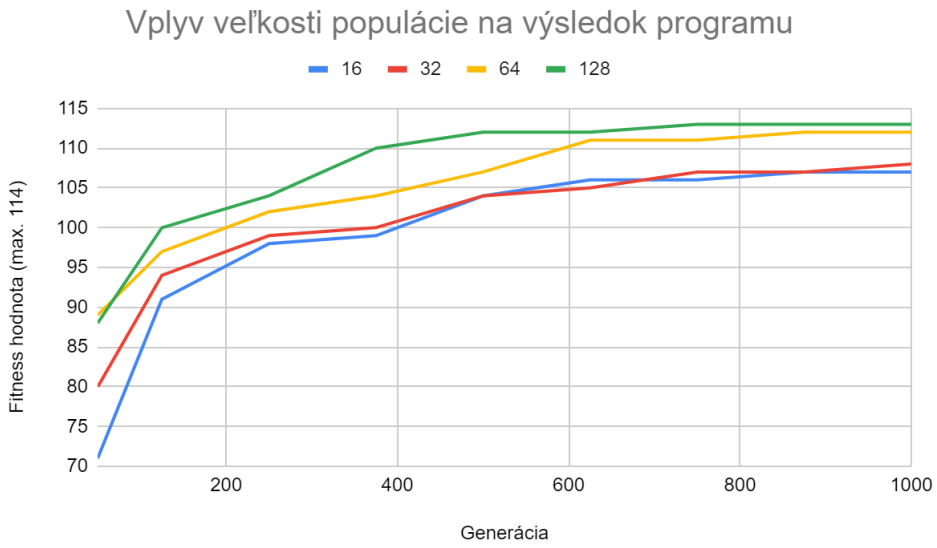
Obr. 15 - nájdenie riešenia pre záhradu 6x6 s dvoma kameňmi - nájdenie riešenia trvalo 150 generácií a počet ťahov je iba o dva vyšší, ako minimum ťahov pre danú hádanku

6.1. Vplyv veľkosti populácie na výsledok

Prvým variabilným faktorom, pre ktorý som chod programu testoval, bola veľkosť počiatočnej populácie. Testoval som algoritmus na vzorke 16, 32, 64 a 128 jedincov.

Počet jedincov síce zlepšuje výkon algoritmu, ale zároveň algoritmus spomaľuje. Zatiaľčo simulácia 1000 generácií so 16 jedincami trvá približne pätnásť sekúnd, pri 128 jedincoch je to už cez tri minúty. V priemere takýto program nájde riešenie s hodnotením fitness o 8 väčším a zároveň riešenie, ktoré je veľmi blízko maximu. Všetky testované veľkosti konvergovali približne v rovnakej miere.

V nasledujúcich podkapitolách a testoch skúšam na zopár vzorkách aj dynamickú veľkosť populácie, ktorá sa s pribúdajúcimi generáciami (a nenájdením riešenia) zväčšuje, v praxi ale takýto prístup výsledok zmenil iba minimálne a vo väčšine prípadov len spomaľoval dobu behu programu.



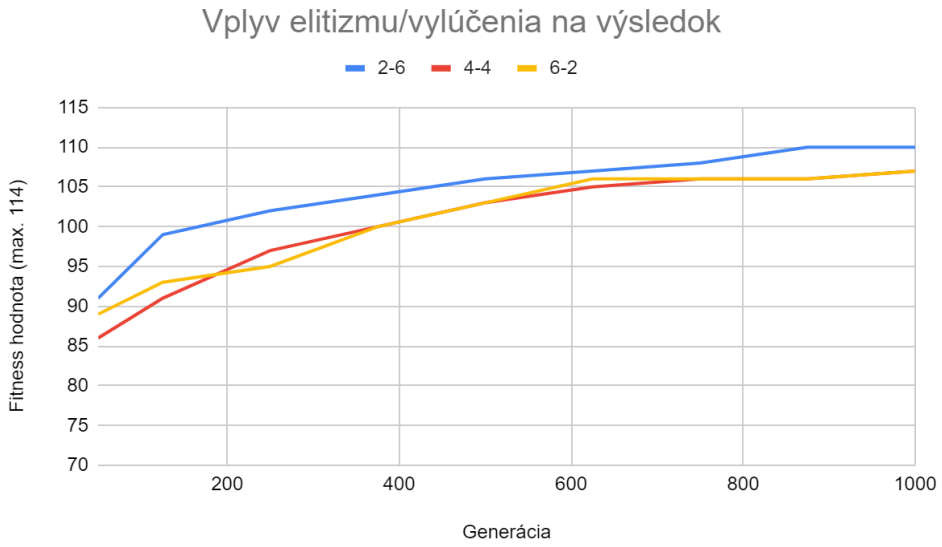
Obr. 16 - testovanie vplyvu veľkosti populácie na výsledok algoritmu

6.2. Vplyv spôsobu selekcie na výsledok

Okrem vplyvu veľkosti populácie som skúmal aj vplyv hodnôt parametrov metód selekcie. Menil som počet vybraných a vylúčených jedincov v rámci elitizmu, formát a veľkosť turnaja a takisto výskyt a agresivitu mutácie.

6.2.1. Vplyv nastavenia elitizmu na výsledok

Testoval som pomer jedincov, ktorí sú v rámci selekcie elitizmom automaticky posunutí do ďalšej generácie a tých, ktorí sú automaticky vylúčení. Ako najúčinnejší sa v prípade 32 jedincov ukázal pomer 2-6 (v prípade vyššieho počtu jedincov pokojne 2-12 či 2-16). Nižší počet elitných jedincov zaručí, že sa najlepší výsledok stále v generácii zachová, zároveň dá ale ďalším, tiež veľmi dobrým jedincom, šancu rozmnožiť sa. Program s vyšším počtom elít a/alebo nižším počtom vylúčených jedincov má tendenciu rýchlejšie konvergovať a to aj v nižšej hodnote fitness.

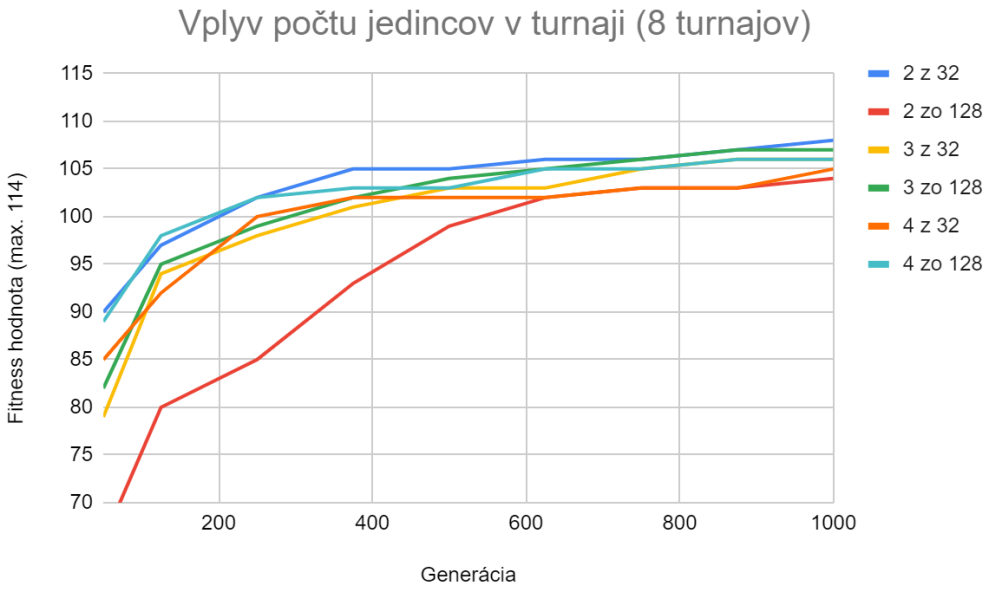


Obr. 17 - vplyv pomeru elitných a vylúčených jedincov na výsledok programu, najlepším riešením je vylúčiť veľa jedincov a posunúť ďalej len zopár elitných

6.2.2. Vplyv formátu a veľkosti turnaja na výsledok

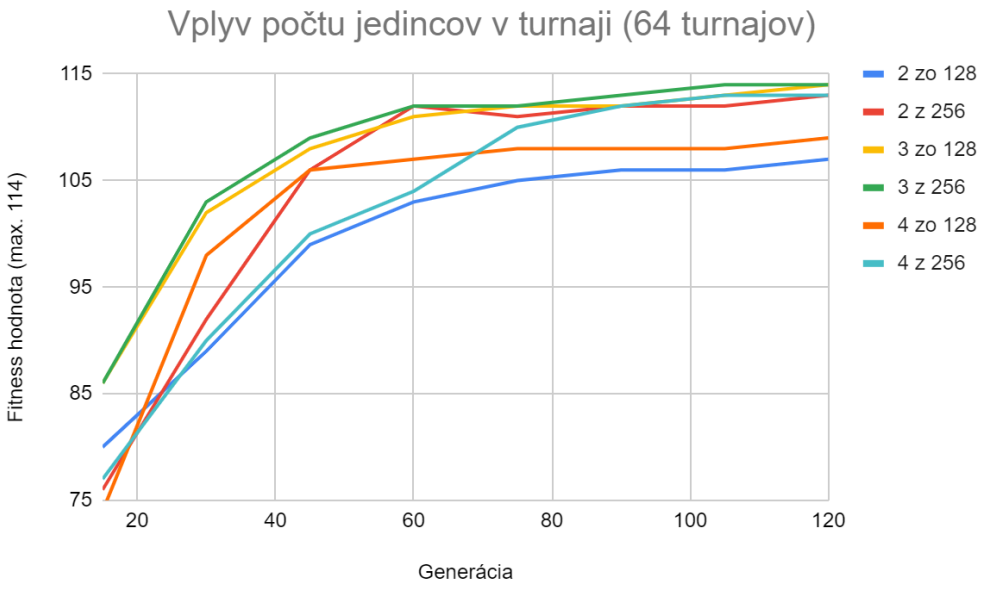
Test, v rámci ktorého som menil parametre turnaja, sa ukázal pre ďalšiu optimalizáciu môjho algoritmu ako kľúčový. V prípade “vykonania” ôsmich turnajov sa turnaj, v ktorom

proti sebe súperili dvaja jednotlivci, ukázal pri malom počte jednotlivcov ako rýchly a takisto rýchlo konvergujúci, no pri vyššom počte jednotlivcov ako veľmi nespoľahlivý. Pri turnaji dvoch jednotlivcov je závislosť výberu lepšieho jednotlivca lineárna a podobá sa náhodnému výberu. Pri turnaji so štyrmi jednotlivcami zas takýto výber častejšie favorizuje silnejších jednotlivcov a aj preto sa konvergenčné hodnoty grafu podobajú hodnotám pri vysokej miere elitizmu a funkcia rýchlo konverguje. Ako ideálna hodnota sa ukázali traja jednotlivci.



Obr. 18 - vplyv počtu jednotlivcov v turnaji pri ôsmich turnajoch

Ďalším testom, ktorý som vykonal, bol vplyv počtu jednotlivcov v turnaji pri "vykonaní" až šesťdesiatich štyroch turnajov v každej generácii. Keďže chcem každého jedinca krížiť ideálne len raz až dvakrát, tento počet turnajov si vyžadoval aj vyšší počet jedincov v populácii. Vyšší počet turnajov aj jedincov zaistí viac krížení (viac potenciálne vyriešených subsektorov, ktoré sa dajú dokopy), čo sa v prípade môjho riešenia ukazuje ako veľmi dobrý krok. Aj pri 64 turnajoch platí, že ideálny počet jedincov v turnaji sú traja, prípadne štyria.



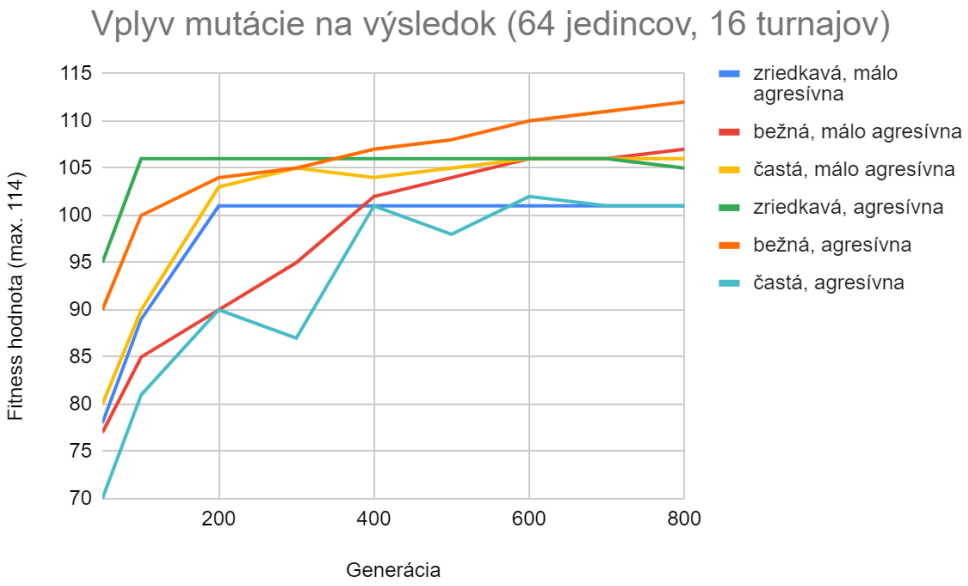
Obr. 19 - vplyv počtu jedincov v turnaji pri šestnástich turnajoch - v porovnaní s ôsmimi turnajmi funkcia konverguje oveľa skôr - už po približne šesťdesiatich generáciách

6.2.3. Vplyv mutácie na výsledok

Pri testovaní nastavení mutácie som si vybral rôzne kombinácie periodicity a agresivity mutácie. Pri zriedkavej mutácii mutovalo 0.1 - 10% jedincov, pri bežnej 0.5 - 25%, pri častej 5 - 50%. Málo agresívna mutácia pridávala/odoberala 1 až 4 ťahy a 1 až 4 bity otáčania, agresívna mutácia menila 8 bitov otáčania a modifikovala 1 až 8 ťahov.

Testy ukázali, že pri zriedkavej mutácii funkcia veľmi rýchlo konvergovala a keďže mutovala veľmi málo, riešenie sa nezlepšovalo. Agresívna mutácia poskytla pri zriedkavom výskyte v priemere jemne lepšie riešenie ako menej agresívna. Naopak, príliš častá mutácia sa podobala v oboch prípadoch nastavenia agresivity náhodnému generovaniu populácie a dokonca nastal prípad, že niektoré generácie dosahovali horšie výsledky ako ich predchodcovia. Bežná miera mutácie (adaptívnej), ktorá mnou bola nastavená pôvodne, sa ukázala pre moju implementáciu ako najlepšie riešenie. Funkcia primerane rýchlo rastie a primerane rýchlo konverguje. Agresívnejšia forma mutácie, pri ktorej sa mení väčšia časť génov, sa ukázala ako jemne účinnejšia.

Vplyv mutácie som testoval na populácii 64 jedincov a šestnástich turnajoch troch jedincov.

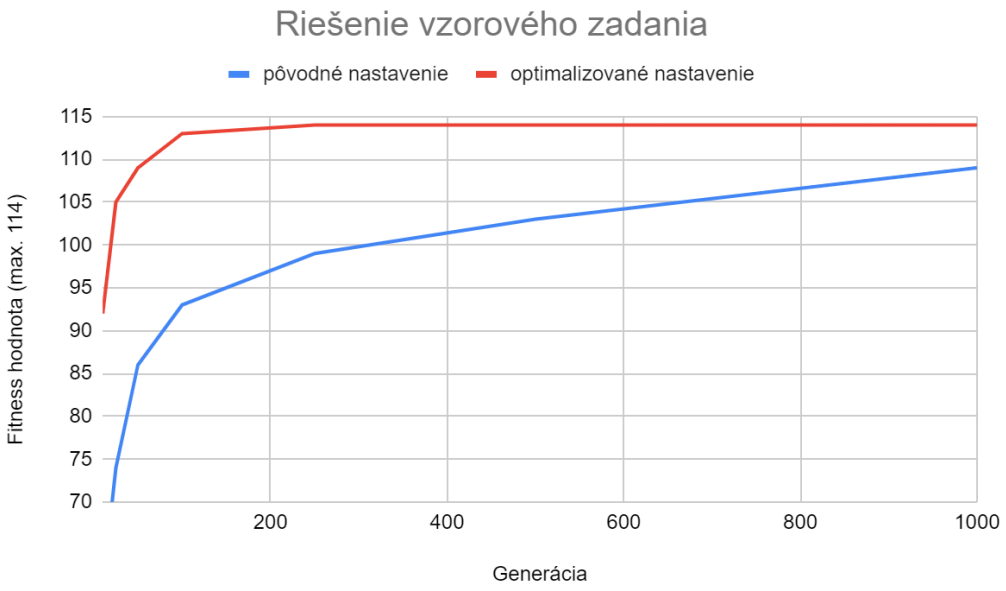


Obr. 20 - vplyv mutácie na výsledok programu

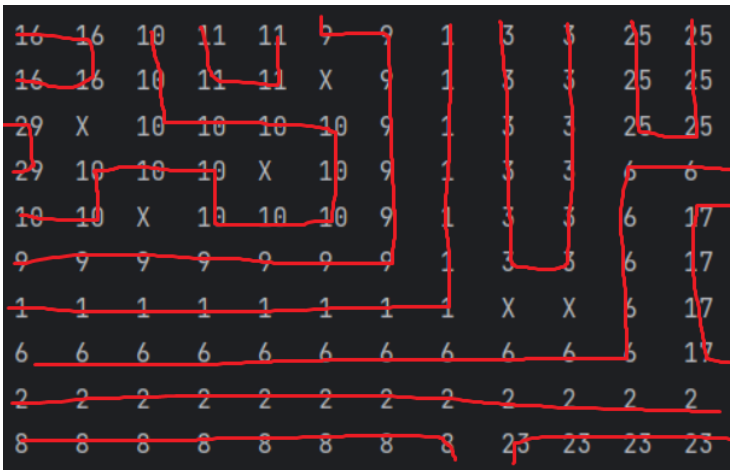
6.3. Riešenie vzorového príkladu

Vzorová záhrada (videná napríklad v Obr. 1) má rozmery 10x12 a obshauje šesť rozmiestnených kameňov. Genetický algoritmus som testoval v dvoch scenároch - prvým bolo mnou vybrané pôvodné nastavenie - 32 jedincov, 4 elity, 4 vylúčení, 8 turnajov s 3 jedincami a adaptívna mutácia s šancou od 0.05 do 0.25. Druhým scenárom bolo optimalizované nastavenie, ktoré vyšlo z predošlých testovaní algoritmu. Počet jedincov sa zmenil na 128 (256 a 512 fungujú obdobne, rýchlosť programu je obetovaná na úkor jemne vyššej presnosti). Počet elít a vylúčení pri takom vysokom počte jedincov nezohrával veľkú úlohu, avšak po vzore testovania vplyvu nastavenia elít som ich počet zmenil na 4 elity a 24 vylúčených. Počet turnajov som zvýšil na 64 a stále vyberal z troch jedincov. Mutácia ostala nezmenená, keďže sa pôvodné nastavenie ukázalo ako optimálne.

Aj napriek tomu, že je evolúcia optimalizovaného riešenia značne pomalšia, optimum aj maximum hádanky dokáže nájsť v omnoho menej generáciách a preto je celková doba čakania na výsledok programu značne kratšia. Zatiaľčo pôvodnému nastaveniu trvá nájdenie maxima cez 2000 generácií, s upraveným nastavením to program zvládne v priemere za 150 a značne sa k nemu priblíži už po päťdesiatich, čo je v mojom programovacom prostredí ekvivalent necelých piatich sekúnd behu programu.



Obr. 21 - porovnanie pôvodného nastavenia parametrov selekčnej funkcie a optimalizovaného nastavenia vychádzajúceho z vykonaných testov



Obr. 22 - kompletné riešenie vzorového riešenia (hádanky) a vizualizácia obdobného riešenia v softvéri Zen Puzzle Garden

6.4. Možné rozšírenie programu

Jedným zo spôsobov, ktorým by sa dala rozšíriť efektivita programu, je zvolenie takzvaného "remote islands" spôsobu selekcie. V rámci tejto metódy sú vyvíjané dve rôzne generácie (alebo viac rôznych generácií). Tieto generácie sa vyvíjajú nezávisle od seba a každých n generácií si vymenia zopár jedincov. Programovací jazyk Python by mi navyše dovolil takúto metódu evolúcie vykonávať pomocou paralelných, viacvláknových výpočtov, takže by ovplyvnila rýchlosť programu iba minimálne.

Keďže mi ale spôsob implementácie tejto metódy príde pre účely zadania a predmetu Umelá inteligencia komplikovaný, rozhodol som sa na ňom svoj algoritmus nestavať. Ďalším dôvodom je to, že už súčasný výkon algoritmu mi poskytuje dostatočne dobré výsledky.

7. Záver

Genetický algoritmus, ktorý som vytvoril, úspešne rieši hádanku Zenovej záhrady, a to pri rôznych veľkostiach záhrady až do 16x16 políčok (pri zmene dĺžky chromozómu možné riešiť aj väčšie záhrady) a rôznom počte a rozložení kameňov. Vzorový zadaný problém je algoritmus schopný pri optimálnom nastavení parametrov selekčnej funkcie vyriešiť v priebehu necelých sto generácií a v rádoch sekúnd. Algoritmus je v prípade optimálneho nastavenia parametrov schopný nájsť optimum riešenia problému či rad správnych sekvencií ťahov iba za zopár desiatok generácií.

Algoritmus je možné ďalej rozšíriť, napríklad odmeňovaním riešení s najmenším počtom krokov alebo paralelným výpočtom použitým k simulovaniu evolúcie, čo by ešte viac upresnilo a urýchlilo hľadanie skutočného maximálneho riešenia hádanky.

8. Zdroje použité pri tvorbe kódu a dokumentácie

Zdroje použité k pochopeniu konceptu genetických algoritmov:

[1] UI 04: Umelá inteligencia - Ing. Lukáš Kohútka, PhD.
[2] UI 05: Umelá inteligencia - Ing. Lukáš Kohútka, PhD.
[2] UI 06: Umelá inteligencia - Ing. Lukáš Kohútka, PhD.

Zdroje použité k pochopeniu riešenia Zenovej záhrady a vizualizácii v dokumentácii:

[1] http://www2.fiit.stuba.sk/~kapustik/zen.html
[2] https://zen-puzzle-garden.en.softonic.com/
[3] https://www.researchgate.net/publication/226997009_A_genetic_algorithm_for_the_Zen_Puzzle_Garden_game

Zdroje použité pri využívaní dátových štruktúr a lambda výrazov v jazyku Python

[1] https://www.w3schools.com/python/python_string_formatting.asp
[2] https://www.w3schools.com/python/python_lambda.asp

Zdroje použité pri programovaní a ladení funkcií:

[1] https://chat.openai.com/chat/ - *využité k programovaniu pomocných funkcií a diskusii o možnostiach implementácie rôznych druhov selekcií v mojom programe*