UMELÁ INTELIGENCIA

Matúš Makay

STU FIIT

Cvičenie: Pondelok 11:00

Cvičiaci: Ing. Martin Komák, PhD.

Dokumentácia:

Genetický algoritmus, Simulované žíhanie

Genetický algoritmus

Mojou úlohou bolo aplikovať prvky *genetického programovania* na vyriešenie problému obchodného cestujúceho. **Problém obchodného cestujúceho** spočíva v tom že mám mapu s rozmerom **NxN** a na tejto mape sa nachádza **X miest** pričom mestá tvoria úplny graf a hrana medzi nimi je rovná euklidovej vzdialenosti ktorú získam z ich súradníc na mape a mojou úlohou bolo nájsť čo najlacnejšiu cestu tak že **obchodný cestujúci** prejde každé mesto práve raz a skončí v tom meste kde začal.

Genetické algoritmy sú **adaptívne prehľadávacie algoritmy** ktoré patria do väčšej skupiny evolučných algoritmov. Základné myšlienky týchto typov algoritmov nájdeme v prírode. V jednoduchosti môžeme povedať že simulujú proces prírodnej selekcie kde jedinec ktorý je najsilnejší(v našom prípade s najvyššou fitness) je schopný prežiť rozmnožiť sa a byť v dalšej generácii.

Generácia sa skladá z **jedincov** z ktorých každý ma svoj vlastný chromozóm ktorý reprezentujem poľom char-ov. Generáciu vytváram krížením chromozómov jedincov ktorých som vybral pomocou **turnajového výberu. Fitness hodnotu** jedinca počítam ako súčet euklidových vzdialeností medzi mestami ktoré navštívil v takom poradí v akom sú v **chromozóme**.

Kríženie a mutácie robím v zmysle zadania. **Mutácie** vykonávam na novo vytvorenom jedincovi takým spôsobom že náhodne vyberiem dva gény z chromozónu a tie medzi sebou vymením. **Kríženie** vykonávam vzdy na dvoch jedincoch ktorý boli z turnajového výbery takým spôsobom že náhodne vygenerujem čislo z intervalu [1, dĺžka chromozónu – 2]. Zoberiem časť chromozónu od prvého jedinca časť od druhého a skombinujem ich do nového chromozónu.

Reprezentácia údajov

Mesto

```python
class City:

    def __init__(self, x, y, name):
        self.name = name
        self.set_cordinates(x, y)

    def __str__(self):
        return f"{self.name}"

    def set_cordinates(self, x, y):
        self.cordinates = (x, y)

    def get_cordinates(self):
        return self.cordinates[0], self.cordinates[1]

    def get_cordinates_euclid(self):
        return (self.cordinates[0], self.cordinates[1])

    def get_name(self):
        return self.name
```

Atribúty:

- name = náhodne vybraté písmeno
- x, y = pozícia mesta v 2D poli

Metódy

- Základné get metódy: get_cordinates, get_cordinates_euclid, get_name
- Základné set metódy: set_cordinates

Jedinec

```
class Individual:

    def __init__(self, chromosone):
        self.chromosone = chromosone

    def __str__(self):
        return f"chromosone: {self.chromosone}"

    def get_length_chromosone(self):
        return len(self.chromosone)

    def get_chromosone(self):
        return self.chromosone

    def swap(self, f_idx, s_idx):
        self.chromosone[f_idx], self.chromosone[s_idx] = self.chromosone[s_idx],
self.chromosone[f_idx]


    def mutate(self):
        f_idx = random.randint(0, len(self.chromosone)-1)
        s_idx = random.randint(0, len(self.chromosone)-1)
        self.swap(f_idx, s_idx)

    def get_fitness(self):
        return self.fitness
```

Atribúty:

- Chromozón = cesta prechádzania mestami

Metódy

- Základné get metódy: get_chromosone, get_length_chromosone, get_fitness
- Vykonávanie mutácie:
    1. Mutate = vygeneruje nahodné indexy ktoré sa medzi sebou vymenia
    2. Swap = vymení hodnoty v chromozóne

Mapa

Vytvorenie mapy

```
def create_board(row, column):
    arr=[]
    rows, cols=row, column
    for i in range(rows):
        col = []
        for j in range(cols):
            col.append(1)
        arr.append(col)
    return arr
```

Naplnenie mapy mestami

```
def generate_map(size_map, num_city, city_list, city_names, choices):
    board = create_board(size_map, size_map)
    while(num_city != 0):
        x, y = generate_cordinates(size_map)

        if are_cordinates_save(x, y, size_map, board):

            choices, rand_char = pick_random_char(choices, city_names)
            city = City(x, y, rand_char)
            board[x][y] = city
            num_city -= 1

            city_list.append(city)

    return board, city_list
```

Slovný opis

Pomocou metódy create_board si vytvorím 2D pole ktoré nasledne vo while cykle napĺňam mestami. Pri vytvorením mesta znížim **num_city** o jedna. While cyklus iteruje kým su neni všetky mestá vytvorené.

Proces jednej iterácie:

1. náhodne vygenerujem dve čísla z intervalu [0, dĺžka mapy -1] **x** a **y**
2. Skontrolujem či na políčku už nie je nejaké iné mesto
    a. True: zo stringu **choices** náhodne vyberiem **jeden znak** ktorý bude pridelený mestu ako jeho **meno**. Vytvorím mesto, nastavím mu atributy a uložím ho na prislušnej pozicii v poli. Znížim počet miest ktoré musím ešte vytvoriť a pridám ho do listu miest.
    b. False: iterácia začína znova

Return values = mapa už s mestami na príslušných pozíciach a list vytvorených miest.

Main

```
#contains every single char from A-Za-z
choices = string.ascii_letters
#stores used chars from choices
city_names = []
city_list = []

board = []

#created board, and city names
board, city_list = generate_map(SIZE_MAP, NUM_CITY, city_list, city_names,
choices)


population = create_first_population(NUM_POP, city_names)
calc_fitness(population, city_list)

convergence = False
best = population[find_best_fitness(population)]
count = 0
```

- Main kód začína inicializovaním základných premenných:
    1. choices = „a-zA-Z“ obsahuje všetky znaky anglickej abecedy velké aj malé pismená. Náhodne vyberám znaky ktoré budú reprezentovať názvy miest
    2. city names = obsahuje všetky znaky vybraté zo stringu **choices**.
    3. city list = obsahuje všetky vytvorené mesta
    4. board = 2D pole reprezentujúce mapu
    5. population = list jedincov ktorých chromozón bol vytvorení náhodne pomocou premennej **city names**
    6. best = najlepší jedinec(s najvačšou fitness hodnotou) s prvej populácie

- Následne pokračuje iteráciami v hlavnom aplikačnom cykle:

```
while(not convergence and LIMIT >= 0):
      population = tournament_selection(population, NUM_POP)
      make_mutations(population)
      calc_fitness(population, city_list)
      cur_best = population[find_best_fitness(population)]
      LIMIT -= 1

      print("cur: ", cur_best.get_fitness())
      if cur_best.get_fitness() > best.get_fitness():
            best = cur_best
      if cur_best.get_fitness() == best.get_fitness():
            count += 1
      if count > 15:
            convergence = True
```

Iterujem kým sa nezačnú opakovať výsledky alebo kým neprekročím zadaný limit.

Popis jednej iterácie

1. Vytvorím novú populáciu krížením tých najlepších náhodne vybratých jedincov.
2. Vykonanie mutácií na novej populácií
3. Vypočím fitness hodnotý nových jedincov
4. Nájdem najlepšieho v novej populácii a porovnám ho s doteraz najlepším jedincom
    i. Ak je lepší nastavím ho na najlepšieho jedinca
    ii. Ak nie je lepši pokračujem v iteráciach

Popis najdôležitejších funkcií

Tournament selection

```
def tournament_selection(list, num_population):
    best = list[find_best_fitness(list)]
    tmp = []
    new_population = []

    while(len(new_population) != num_population-1):

            num_individual = int(random.randint(1, len(list))/2)
        parent_a = pick_parent(num_individual, tmp, list, "")

        try:
            parent_b = pick_parent(num_individual, tmp, list,
parent_a.get_chromosone())
            child = crossover(parent_a, parent_b)
            if child is not False:
                new_population.append(crossover(parent_a, parent_b))

        except AttributeError:
            continue

    new_population.append(best)

    return new_population
```

Vstupy:

- list = zoznam jedincov zo starej generácie
- num_population = pocet jedincov v generácii

Výstup:

- nová generácia ktorá obsahuje aj najlepšieho zo starej generácie

Slovný popis:

Nájdem si najlepšieho jedinca zo starej generácie a uložim ho. Následne sa cyklím kým sa počet jedincov v novej generácii nerovná dĺžke starej generácie – 1 (ako posledneho pridam

najlepšieho zo starej generácie). Po ukončení cyklu pridám do novej generácie najlepšieho zo starej

Priebeh jednej iterácie cyklu:

1. Náhodne si zvolím číslo, podľa ktorého budem vyberať účastníkov turnaja.(výherca turnaja bude jeden z rodičov)
2. Vyberiem si **rodičaA** a následne **rodičaB** z ktorých vytvorím krížením ich chromozónu **potomka** ktorého vložím do novej generácie

Kríženie

```
def crossover(par_a, par_b):

    if par_a is False or par_b is False:
        return False

    split_idx, length = init_crossover(par_a)

    chrom_child = []
    chrom_para = par_a.get_chromosone()
    chrom_parb = par_b.get_chromosone()

    for idx in range(0, split_idx):
        chrom_child.append(chrom_para[idx])

    add_gen_in_child_chrom(split_idx, length, chrom_child, chrom_parb)
    if chrom_child != length:
        add_gen_in_child_chrom(0, split_idx, chrom_child, chrom_parb)

    return Individual(chrom_child)
```

Vstupy:

- para, parb = rodičia vybratý pomocou turnajového výberu

Výstup:

- nový jedinec vzniknutý krížením chromozónu z rodičov

Slovný popis:

Vyberiem si **náhodné čislo(split_idx)** podľa ktorého budem deliť výber génov z **para** a **parb.** Následne do chromozónu potomka vkladám gény z **rodiča** a až po **split_idx.** Potom sa pokúsim vložiť do chromozónu potomka gény z **parb od split_idx** pričom už musím sledovať aby sa nejaké písmeno nevyskytovalo 2x v novom chromozóne. Preto ak nastane situácia že dĺžka chromozónu potomka nie je rovná dĺžke chromozónu rodiča zavolám vkladanie génov od indexu 0 z **parb.**

Mutácie

```
class Individual:

    def swap(self, f_idx, s_idx):
        self.chromosone[f_idx], self.chromosone[s_idx] = self.chromosone[s_idx], 
self.chromosone[f_idx]


    def mutate(self):
        f_idx = random.randint(0, len(self.chromosone)-1)
        s_idx = random.randint(0, len(self.chromosone)-1)

        self.swap(f_idx, s_idx)
```

Slovný opis:

Pri mutácii vyberiem 2 náhodne vybraté gény a vymením ich miesta v chromosome

Počítanie fitness hodnoty

```
class Individual:
```

```
    def calc_fitness(self, city_list):
        #cordinates city
        lenght = self.get_length_chromosone()
        self.fitness = 0
        for idx in range(0, lenght - 1):
            name_l = self.chromosone[idx]
            name_r = self.chromosone[idx+1]
            citya, cityb = self.find_city_in_list(name_l, name_r, city_list)

            self.fitness += self.calc_euclid_range(citya, cityb)

        #uzavrie slucku prve s poslednym mestom
        name_f = self.chromosone[0]
        name_l = self.chromosone[lenght - 1]

        city_f, city_l = self.find_city_in_list(name_f, name_l, city_list)

        self.fitness += self.calc_euclid_range(city_f, city_l)


        self.fitness = 1/self.fitness
```

Slovný popis:

V cykle postupne prechádzam chromozón a počítam vzdialenosti jednotlivých miest a spočítavam celkovu vzdialenosť. Ako poslednu pripočítam vzdialenosť prveho genu a posledneho génu čím uzatvorím cestu.

Popis jedneho cyklu:

1. i-ta iteracia
2. zoberiem i-ty gen z chromozonu, zoberiem i+1 gen z chromozónu. Nájdem si v polí miest tieto mestá aby som z nich zistil ich suradnice
3. následne vypočítam euklidovu vzdialenost medzi nimi.
4. Pripočítam euklidovu vzdialenost k celkovej fitness hodnote

Simulované žíhanie

Algortimus **simulovaného žíhania** je **náhodny optimalizačný algoritmus**. To znamená že využíva **náhodnosť na akceptáciu riešenia**, takže narozdiel od ostatných hill-climb algoritmov môže **akceptovať aj horšie riešenie**. Náhodnosť akceptovania horších riešení je zo začiatku vyššia ale postupne pri prehladávaní sa znižuje. Toto správanie dáva algoritmu príležitosť objaviť oblasť lokálneho minimuma a postupne toto minimum vyhladiť.

Akceptovanie horších riešení ako práve najlepšie riešenie je funkcia teploty prehladávania a rozdielu medzi fittnes hodnotou práve najlepšieho riešenia a náhodné vybratého jedinca. Na znižnovanie teploty sa môžu použiť rôzne techniky. Ja som si vybral spôsob s názvom **„fast simulated annealing“** ktorý sa počíta **temperature = initial_temperature / (iteration_number + 1).** Akceptácia náhodného riešenie je aj funkcia kritéria ktoré sa počíta pomocou vzorca **criterion = e^(-(candidate.fitness – best.fitness) / temperature)** tento spôsob sa volá „metropolis acceptance criterion“. Vysledok tejto funkcie porovnám s náhodne vygenerovaným číslom, ak je náhodne vygenerovné čislo menšie ako výsledok funkcie práve vybraného kandidáta akceptujem ako riešenie a ďalej počítam s ním.

Reprezentácia údajov

Údaje reprezentujem rovnako ako pri genetických algoritmoch. Jediny rozdiel je vo get-ovaní fitness hodnoty jedinca. To ale popíšem v algoritmoch nieżšie. Základné atribúty každej class-y reprezentujem rovnako.

Main

```
if sys.argv != 4:
    exit("You have to specify: num_pop, num_city, size_of_map")
NUM_POP = sys.argv[1]
NUM_CITY = sys.argv[2]
SIZE_MAP = sys.argv[3]

ITERATIONS = 1000
INIT_TMP = 500


#contains every single char from A-Za-z
choices = string.ascii_letters
#stores used chars from choices
city_names = []
city_list = []

board = []

#created board, and city names
board, city_list = generate_map(SIZE_MAP, NUM_CITY, city_list, city_names,
choices)


population = create_first_population(NUM_POP, city_names)

calc_fitness(population, city_list)

best_s = population[find_best_fitness(population)]
cur_b = best_s
```

- Main kód začnem inicializáciou počiatočných atríbutoch ako
    1. NUM_POP: počet individuí v populácií
    2. NUM_CITY: počet miest na mape
    3. SIZE_MAP: velkosť poľa
    4. ITERATIONS: maximalny počet iterácii hlavného aplikačného cyklu
    5. INIT_TMP: počiatočná teplota žíhania

6. Choices: „a-zA-Z“ obsahuje všetky znaky anglickej abecedy velké aj malé pismená. Náhodne vyberám znaky ktoré budú reprezentovať názvy miest
7. city names: obsahuje všetky znaky vybraté zo stringu **choices**.
8. city list: obsahuje všetky vytvorené mesta
9. board: 2D pole reprezentujúce mapu
10. population: list jedincov ktorých chromozón bol vytvorení náhodne pomocou premennej **city names**
11. best: najlepší jedinec(s najvačšou fitness hodnotou) s prvej populácie

Hlavný cyklus aplikácie

```
for idx in range(0, ITERATIONS):
    while True:
        if len(population) < NUM_POP/2:
            print_result(best_s, board, city_list)
            exit(5)

        candidate = pick_random(population)

        if candidate.get_fitness() < best_s.get_fitness():
            best_s = candidate
            cur_b = candidate

        cur_temp = INIT_TMP / (idx + 1)
        dif = candidate.get_fitness() - cur_b.get_fitness()
        criteria = exp(-dif/cur_temp)

        if(dif < 0 or rand() < criteria.real):
            cur_b = candidate
            break

        population.remove(candidate)

    population = create_new_pop(cur_b, NUM_POP)
    make_mutations(population)
    calc_fitness(population, city_list)

print_result(cur_b, board, city_list)
```

Slovný opis

Hlavná časť programu sa skladá z dvoch vnorených cyklov.

I. **Vonkajší cyklus for:**

- iteruje od 0 po hodnotu ktorú sme inicializovali na začiatku.
- Ako prvé sa ponorím do **vnútorného cyklu**. Ked ten skončí vykonavanie následne:

1. Vytváram novú populáciu z aktuálne najlepšieho jedinca.
2. Vykonám mutácie na novej populácií
3. Vypočítam fitness hodnotu

- Následne na tejto populácií vykonám mutácie a vypočítam fitness hodnotu jedincov.

II. **Vnútorný cyklus while:**

- Principiálne nekonečný while cyklus ktorý skončí až keď odstránim všetkých jedincov v populácií alebo zakceptujem riešenie.
- Popis jednej iterácie
    1. Skontrolujem či sa pole population nie je prazdne ak je skončím inak pokračujem
    2. Vyberiem náhodného jedinca z populácie
    3. Ak má lepšiu fitness hodnotu označím ho ako nového najlšieho jedinca
    4. Vypočítam teplotu na základe iterácie cyklu
    5. Vypočítam kritérium akceptácie
    6. Ak algoritmus:
        a) **akceptuje** riešenie potom označím náhodne vybratého jedinca ako aktuálne najlepšieho a ukočním cyklus. Z neho následne vytvorím novú populáciu
        b) **neakceptuje** riešnie potom odstránim jedinca z populácie a cyklus pokračuje od bodu **1.**

Popis najdôležitejších funkcií

create_new_pop

```
def create_new_pop(parent, NUM_POP):
    population = []

    for i in range(0, NUM_POP):
        population.append(crossover(parent))

    return population
```

crossover

```
def crossover(parent):

    rand_idx = random.randint(1, len(parent.get_chromosone())-2)
    chromosone = parent.get_chromosone()

    new_chromosone = []
    chrom1 = chromosone[0: rand_idx]
    chrom2 = chromosone[rand_idx:]

    for i in range(0, len(chrom2)):
        new_chromosone.append(chrom2[i])

    for i in range(0, len(chrom1)):
        new_chromosone.append(chrom1[i])
    return Individual(new_chromosone)
```

Slovný opis

- Nového jedinca vytváram iba z chromozónu jedného rodiča.
- Náhodne vygenerujem číslo podľa ktorého rozpolím rodičov chromozón a gény v ňom swapnem

make_mutations

```
def make_mutations(list):
```

```
    for individual in list:
        individual.mutate()
        individual.mutate()
        individual.mutate()
        individual.mutate()
        individual.mutate()
        individual.mutate()
```

Slovný opis

- Nad každým jedincom v poli zavolám funkciu mutovania 6x kvôli zvýšeniu náhodnosti riešenia

mutate

```
    def mutate(self):
        f_idx = random.randint(0, len(self.chromosone)-1)
        s_idx = random.randint(0, len(self.chromosone)-1)

        self.swap(f_idx, s_idx)
```

Slovný opis

- Nádhodne vygenerujem dve čísla a gený na týchto indexoch vymením

Testovanie

Návod na spustenie

```
if len(sys.argv) == 2:
    SIZE_MAP = int(sys.argv[1])
    NUM_POP = randint(20, SIZE_MAP)
    NUM_CITY = randint(30, SIZE_MAP-2)
elif len(sys.argv) == 4:
    SIZE_MAP = int(sys.argv[1])
    NUM_POP = int(sys.argv[2])
    NUM_CITY = int(sys.argv[3])
elif len(sys.argv) == 3:
    SIZE_MAP = randint(50, 150)
    NUM_POP = int(sys.argv[1])
    NUM_CITY = int(sys.argv[2])
elif len(sys.argv) == 5:
    SIZE_MAP = int(sys.argv[1])
    NUM_POP = int(sys.argv[2])
    NUM_CITY = int(sys.argv[3])
    INIT_TMP = int(sys.argv[4])
else:
    SIZE_MAP = randint(50, 150)
    NUM_POP = randint(20, 100)
    NUM_CITY = randint(20, 52)
    INIT_TMP=1000
```

Program môžete spustiť nasledovným spôsobom. V linuxe sa nastavíte do priečinku zad2/src a potom sa dáte do priečinku genetic_algorithm ak chcete spúštať geneticky algoritmus. Alebo sa môžete presunút do simul_zihania ak chcete spúštať algoritmus simulovaného žíhania. Ak už ste v jednom z vyššie spomenutych priečinkov následne príkazom „python main.py **size**" spustíte program so želaným rozmerom mapy . Takisto môžete program spustiť príkazom „python main.py **size num_pop num_city**" vtedy sa nastaví rozmer mapy, počet individui v populácii a počet miest na mape. Ak nezadáte žiadne vstupné argumenty program všetko nastaví náhodne.

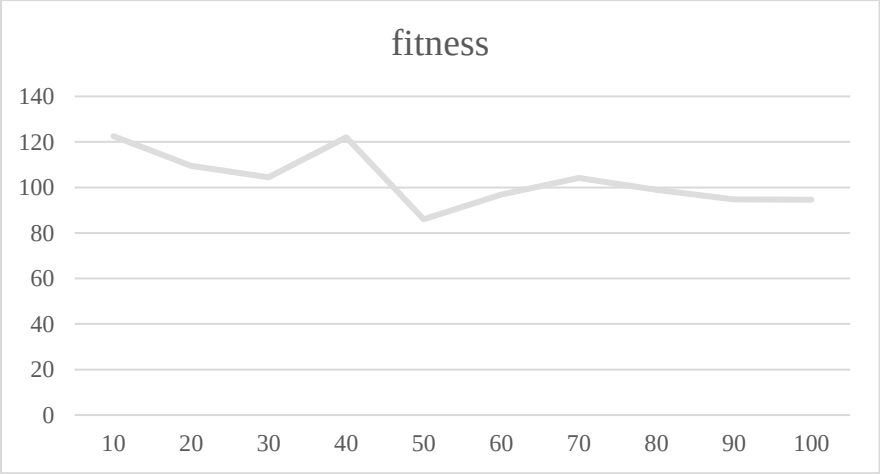
Spôsob testovania

Genetický algoritmus a aj algoritmus simulovaného žíhania testujem rovnakým spôsobom. Postupne mením **rozmer mapy**, **počet individuí v populácii**, a **počet miest v mape** a sledujem **vplyv zmeny** týchto premenných na **najlepšie nájdený výsledok** a **čas** za ktorý sa úlohy vykonajú. Pričom ak mením rozmer mapy tak ostatné premenné nemením. Takto postupujem pre všetky prípady

Genetický algoritmus

Závislosť najlepšieho výsledku od počtu individui v populacii

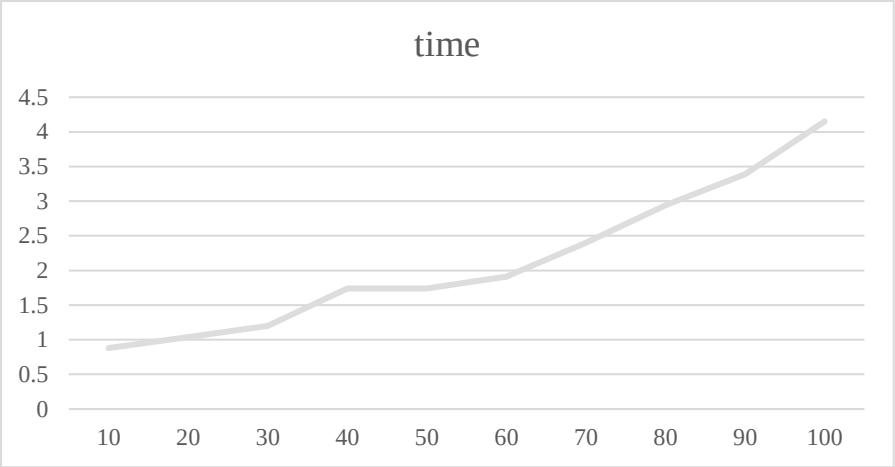
num_pop	10	20	30	40	50	60	70	80	90	100
fitness	122.52	109.52	104.52	121.98	86.01	96.83	104.12	98.91	94.81	94.64



- Môžeme vidieť že pri zvyšujúcom počte individui a nemennom počte miest algoritmus dostáva viacej priestoru pre náhodnosť generovania výsledkov a postupne nájde lepšie a lepšie riešenia.

Závisloť času vykonania od počtu individui v populácii

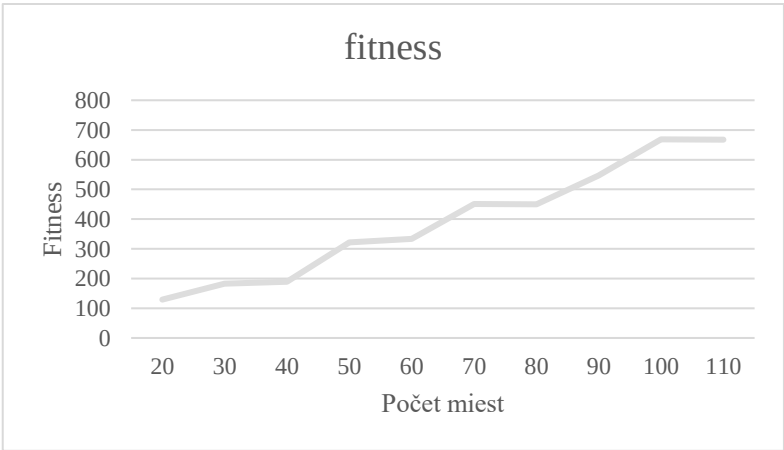
num_pop	10	20	30	40	50	60	70	80	90	100
time	0.88	1.04	1.2	1.74	1.74	1.91	2.4	2.94	3.39	4.15



- Vidíme že s narastajúcim počtom individui čas vykonania exponenciálne narastá. Samozrejme je to kvôli zvačšenej výpočtovej náročnosti pri generoví novej populácie.

Závislosť najlepšieho výsledku od rozmeru mapy

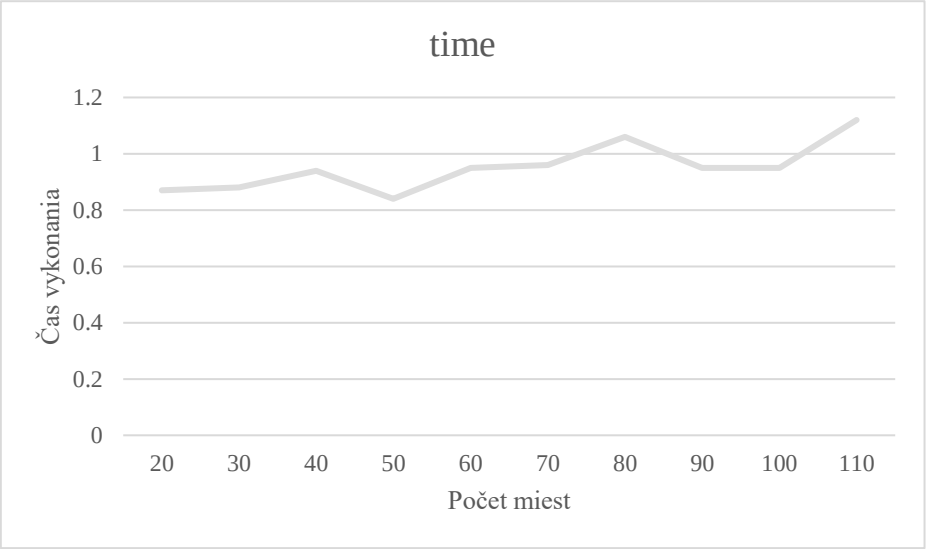
size	20	30	40	50	60	70	80	90	100	110
fitness	129.1	182.7	189.04	321.84	333.43	450.56	449.63	546.28	668.44	667.03



- Vidíme že s narastajúcim rozmerom najlepšie riešenie prirodzene narastá nakolko sa s veľkou pravdepodobnosťou zvušujú vzdialenosti medzi mestami

Závislosť času vykonania od rozmeru mapy

size	20	30	40	50	60	70	80	90	100	110
time	0.87	0.88	0.94	0.84	0.95	0.96	1.06	0.95	0.95	1.12

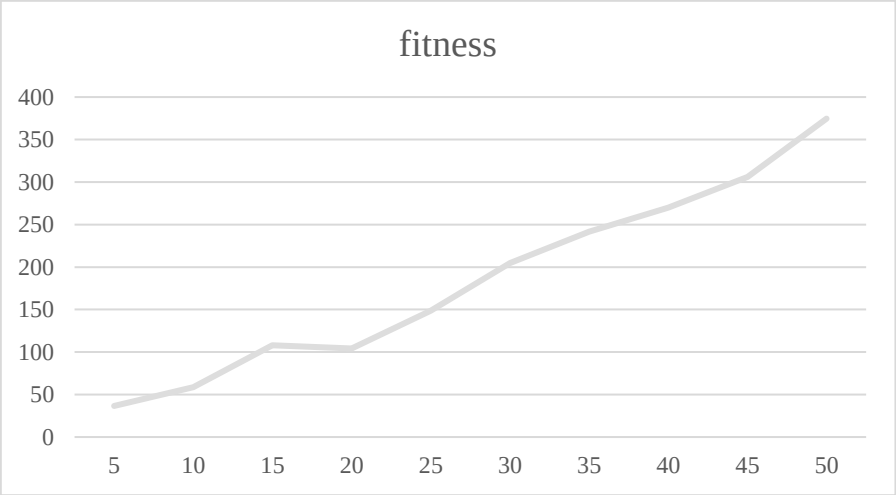


- Časová závislosť postupne narastá so zvyšujúcim sa počtom miest ale tento nárast nie je výrazný.

Závislosť najlepšie nájdeného riešenia od počtu miest na mape

- Kedže mestá označujem pomocou „string.ascii_letters“ ktorý má rozmer 52 maximálny počet miest na mape je 52 aby platilo že každé mesto je unikátne

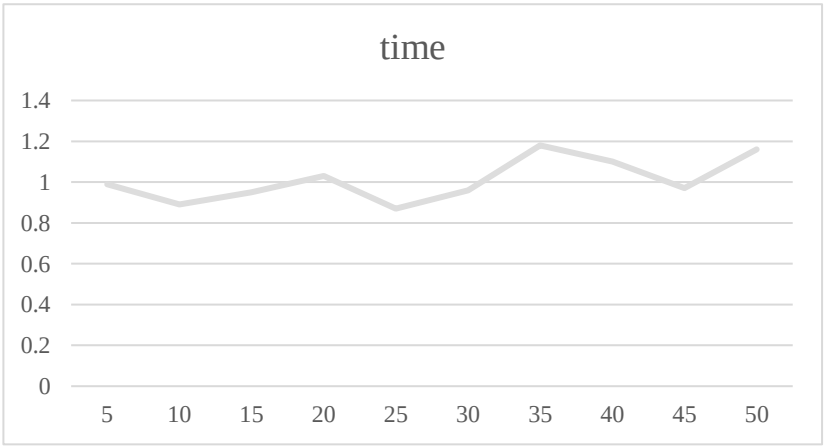
pocet_miest	5	10	15	20	25	30	35	40	45	50
fitness	36.6	58.65	108.08	104.33	148.54	204.83	241.56	269.99	305.9	374.63



- Vidíme že so zvyšujúcim počtom miest na mape fitness hodnota značne narastá

Závislosť času vykonania riešenia od počtu miest na mape

pocet_miest	5	10	15	20	25	30	35	40	45	50
time	0.99	0.89	0.95	1.03	0.87	0.96	1.18	1.1	0.97	1.16

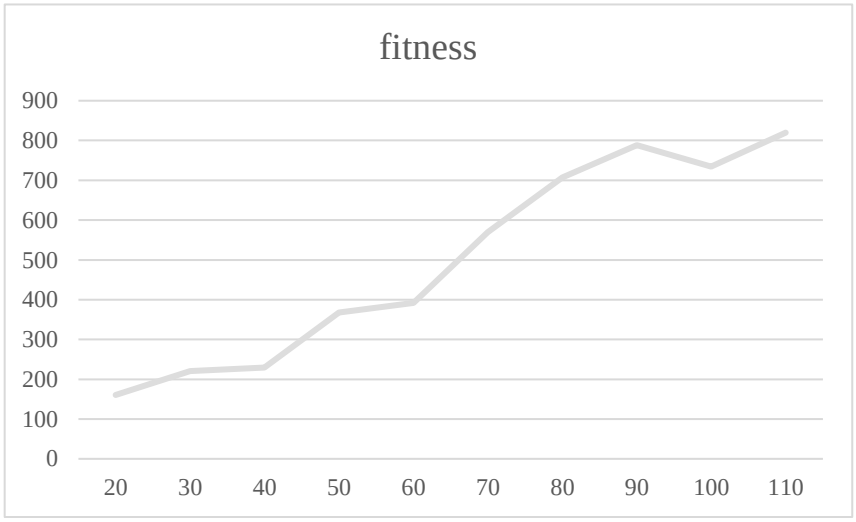


- Vidíme že počet miest na mape časovú závislosť výrazne neovplyvňuje

Simulované žíhanie

Zavislosť najlepšeho riešenia od rozmeru mapy

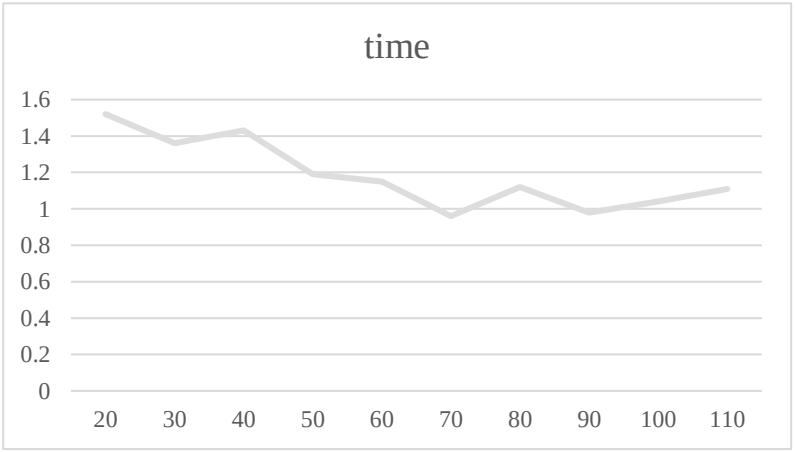
size	20	30	40	50	60	70	80	90	100	110
fitness	160.47	220.54	229.19	367.69	392.1	569.72	707.41	788.17	734.07	819.73



- Môžeme vidieť že s narastajúcim rozmerom fitness hodnota prirodzene narastá so zvačšujúcou sa vzdialenosťou medzi nimi

Zavislosť času vykonania od rozmeru mapy

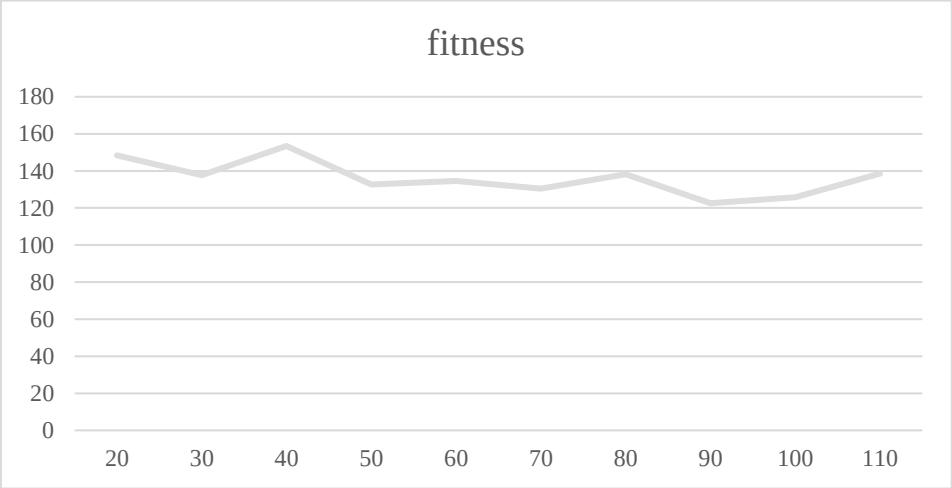
size	20	30	40	50	60	70	80	90	100	110
time	1.52	1.36	1.43	1.19	1.15	0.96	1.12	0.98	1.04	1.11



- Vidíme že časová závislosť nie je na rozmere mapy závislá

Zavislosť najlepšeho riešenia od počtu individuí v populácii

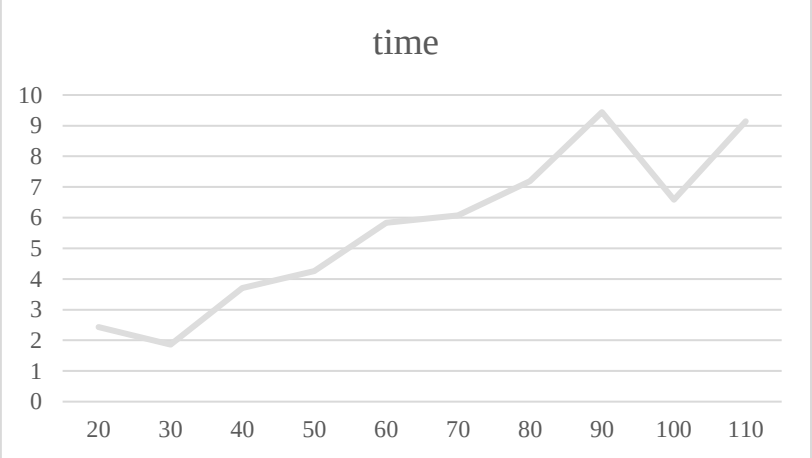
num_population	20	30	40	50	60	70	80	90	100	110
fitness	148.37	137.77	153.45	132.72	134.63	130.4	138.14	122.57	125.82	138.51



- V grafe môžeme vidieť že s narastajúcim počtom individuí v populácii sa fitness hodnota znižuje. Toto správanie je žiadúce.

Zavislosť času vykonania od počtu individuí v populácii

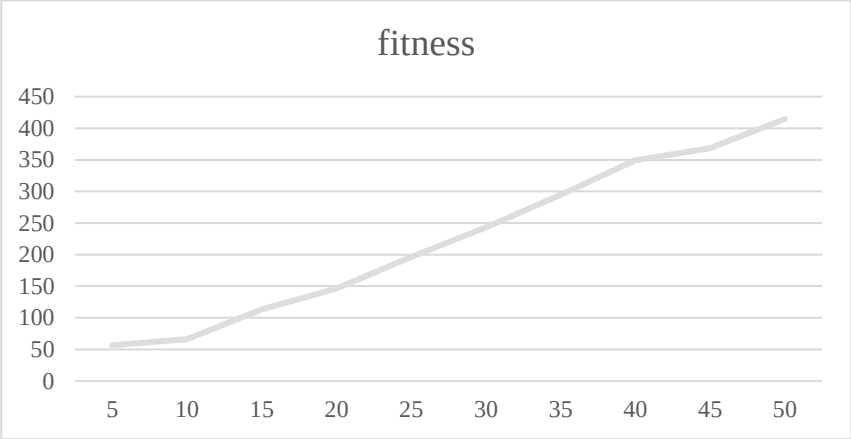
num_population	20	30	40	50	60	70	80	90	100	110
time	2.43	1.86	3.7	4.26	5.83	6.07	7.19	9.44	6.58	9.14



- Vidíme že časová závislosť rastie až nechcene vysoko. Toto správanie otestujem znižovaním teploty žíhania nakolko algoritmus by mal byt rychlejší ako genetické algoritmy ale podla tohto správania to tak nevyzerá

Zavislosť najlepšeho riešenia od počtu miest na mape

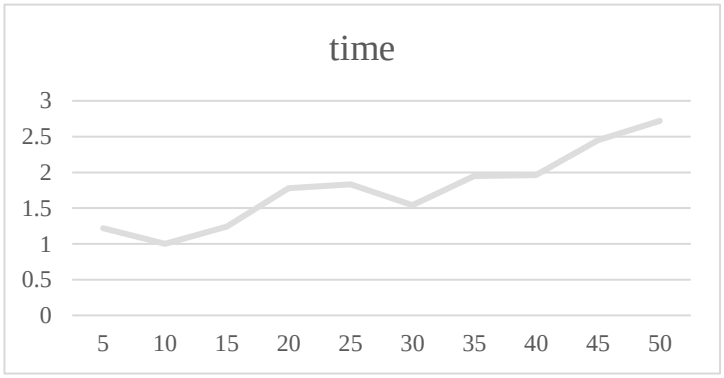
num_city	5	10	15	20	25	30	35	40	45	50
fitness	56.65	66.02	113.51	146.02	196.41	243.14	294.53	349.02	368.66	414.59



- Vidime že najlepšia fitness hodnota rastie s pribúdajúcimi mestami na mape. Toto správanie sme očakávali nakoľko algoritmus musí pridať viacej génov do chromozónu z čoho logicky vyplýva vyššia fitness hodnota

Zavislosť času vykonania od počtu miest na mape

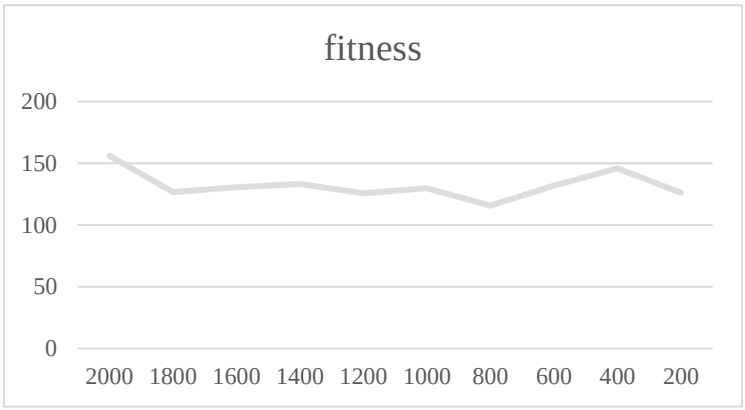
num_city	5	10	15	20	25	30	35	40	45	50
time	1.22	1	1.24	1.78	1.83	1.54	1.95	1.96	2.45	2.72



- Vidíme že časová závislosť rastie primerane ale nie výrazne s narastajúcim počtom miest.

Zavislosť najlepšieho riešenia od počiatočnej teploty žíhania

temperature	2000	1800	1600	1400	1200	1000	800	600	400	200
fitness	156.05	126.58	130.58	133.24	125.65	129.76	115.66	131.86	145.91	126.08

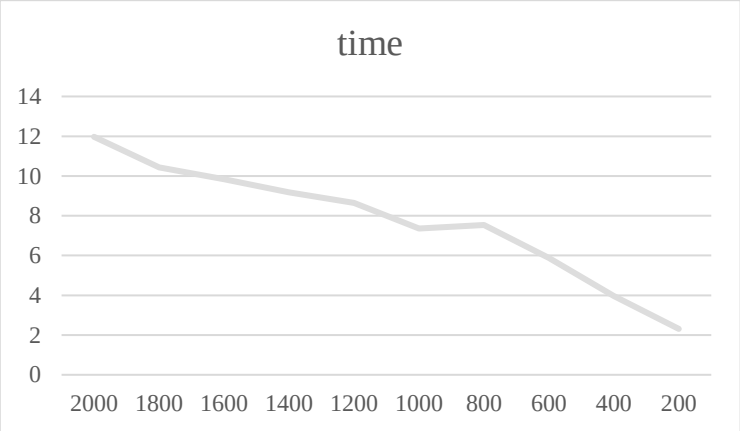


- Pri tomto meraní sa ukázala náhodnosť a nepredvídatelnosť algoritmu. Meranie som zopakoval viac krát a trend nachádzania lepšej fitness hodnoty pri nižších teplotách sa ukázal ako nespolahlivý.

Zavislosť času vykonania od počiatočnej teploty žíhania

Testované na 100 jedincov v populácii

temperature	2000	1800	1600	1400	1200	1000	800	600	400	200
time	11.97	10.43	9.84	9.18	8.65	7.35	7.54	5.88	3.97	2.31



- Vidíme že časová závislosť je silno previazaná s počiatočnou teplotou žíhania.

  Pričom so znižujucou sa teplotou algoritmus končí oveľa skôr

Záver k testovaniu

Pri testovaní sa ukázalo že genetické algoritmy sú zvačša časovo náročnejšie na čas. Najviac sa to ukáže ak algoritmu simulovaného žíhania nastavíme nižšiu počiatočnú teplotu žíhania. Na druhú stranu genetický algoritmus poskytoval spolahlivejšie výsledky. Ak by som potreboval menej presné výsledky v kratšom čase použil by som algoritmus simulovaného žíhania. Ak by nezáležalo na čase vykonávania ale na čo najlepšom výsledku zvolil by som genetický algoritmus.

Algoritmy najviac časovo ovplyvňoval počet jedincov v populácii. Pri zvyšovaní počtu jedincov čas vykonávania exponenciálne rástol pre oba algoritmy. Pre algoritmus simulovaného žíhania sa toto správanie dalo obmedziť znížením teploty žíhania.

Najlepšie výsledky fitness hodnoty dosahovali oba algoritmy pri čo najvačšom počte jedincov v populácii. Ako som už spomenul vyššie so zvýšením počtu jedincov čas vykonávania exponenciálne narastal. Zaujímavosťou pri algoritme simulovaného žíhania bolo to že tento časový nárast sa dal obmedziť znížením teploty žíhania a výsledky sa dramaticky nezhoršili. Neboli také nízke ako pri vysokej teplote ale podľa môjho názoru boli dostačujúce.