Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Umelá Inteligencia

Zadanie č.2

Problém obchodného cestujúceho (C)

Genetický algoritmus (1), zakázané prehľadávanie (2)

Akademický rok 2022/2023

Meno: Bc. Irina Makarova
Cvičiaci: Ing. Martin Komák, PhD. **Dátum:** 13.11.2022

Obsah

Definícia problému

Problém obchodného cestujúceho patrí medzi NP-úplne problémy a spočíva v nájdení najlacnejšej uzavretej cesty, ktorá by obsahovala každé miesto z danej množiny, pričom každé miesto je navštívené práve raz. Miesta sú reprezentované dvojicou súradníc ([0; n-1], [0; m-1]), kde m x n je veľkosť mapy. Cena cesty medzi dvoma miestami je rovná Euklidovej vzdialenosti medzi nimi. Algoritmy popísané nižšie fungujú za predpokladu, že existuje cesta medzi každou dvojicou miest z množiny, a že ceny priamej a spiatočnej cesty medzi ľubovoľnou dvojicou miest sú rovnaké. Miesta na mape sa teda dajú reprezentovať úplnym grafom s ohodnotenými obojsmernými hranami. Úlohou algoritmu je nájsť Hamiltonovskú kružnicu s uspokojivo nízkym ohodnotením.

Generovanie inštancie problému

Súradnice miest sa generujú vo funkcii *generateGenes()* náhodne zo zadaného používateľom rozsahu hodnôt. Z dvoch postupnosti celých čísel sa podľa počtu miest náhodne vyberie požadovaný počet x-ových a y-ových súradníc. Následne sa súradnice s rovnakými indexmi v poli spoja do dvojíc a uložia sa do jedného nemenného poľa typu tuple. Takýto postup zaručuje, že nevzniknú dve rovnaké dvojice súradníc, preto výsledný zoznam netreba kontrolovať na výskyt duplicitných prvkov.

```
def generateGenes(mapSize_x, mapSize_y, numOfCities):
    xCoordinates = random.sample(range(mapSize_x), numOfCities)
    yCoordinates = random.sample(range(mapSize_y), numOfCities)
    allCities = tuple(zip(xCoordinates, yCoordinates)) #nahodne v
    return allCities
```

Vygenerované miesta sa počas vykonávania algoritmu uchovávajú v objekte triedy *Cities*, ktorý obsahuje aj funkcie na výpočet ceny cesty medzi ľubovoľnými dvoma miestami a taktiež hodnoty fitness pre zadanú permutáciu miest. Aby sa ušetril pamäťový priestor, do permutácií sa neukladajú hodnoty súradníc, ale indexy miest uložených v *Cities*.

```
class Cities:
    def __init__(self, coordinatesOfCities, numOfCities):
        self.allCitiesCoords = coordinatesOfCities
        self.numOfCities = numOfCities

    def calculateDistance(self, indexOfCity_1, indexOfCity_2):···

    def calculateFitness(self, permutation):···
```

Genetický algoritmus

Genetické algoritmy sú algoritmy heuristického hľadania, pri ktorých sa simuluje prirodzený výber. Jedince sú reprezentované postupnosťou čísel podobnej chromozómom, a tvoria populáciu konštantnej veľkosti. Každý chromozóm predstavuje potenciálne riešenie problému. Gény sú mestá, permutácia miest je chromozóm. Počas behu algoritmu sa populácia vyvíja a dcérske generácie nahrádzajú rodičovské. Nová generácia môže obsahovať jedincov z predošlej generácie, nových jedincov vzniknutých krížením vybraných chromozómov a taktiež náhodne vygenerovaných jedincov. Na populáciu môžu pôsobiť aj mutácie prispievajúce k udržiavaniu jej variability. Spôsob tvorby novej generácie je rozhodujúci pre nájdenie riešenia blízkeho optimálnemu.

Popis implementácie algoritmu

Gén – index miesta v zozname *allCitiesCoords* uloženom v triede *Cities*

Chromozóm – pole veľkosti n obsahujúce celé čísla z rozsahu od 0 do n-1, kde n je veľkosť *allCitiesCoords*

Jedinec – objekt triedy *Permutation*, ukladá v sebe chromozóm a hodnotu fitness

Populácia – objekt triedy *Population*, ukladá v sebe veľkosť generácie a samotnú generáciu v podobe poľa jedincov.

Inicializácia prvotnej populácie

Počiatočná generácia sa vytvára vo funkcii *createRandomIndividuals()* náhodne, ale s ohľadom na to aby všetky jedince boli unikátni. V prvom cykle sa vygeneruje požadovaný počet permutácií indexov miest. Permutácie sa ukladajú do štruktúry set, ktorá neumožňuje pridávanie prvku, ak v nej už je, čím sa zaisťuje, že sa nevyskytnú rovnaké permutácie. V ďalšom cykle sa po jednej odoberajú permutácie zo setu a vytvárajú sa jedinci, ktoré sa ukladajú do poľa *individuals*.

```
def createRandomIndividuals(numOfChromosomes, numOfCities, genes):
    individuals = list() #mnozina jedincov s unikatnymi chromozomami
    randomPermutations = set() #nahodne generovane unikatne permutacie miest
    while len(randomPermutations) < numOfChromosomes: #vytvori unikatne permutacie indexov miest ulozenych v genes
        perm = random.sample(range(numOfCities), numOfCities) #preusporiadane indexy miest v genes
        randomPermutations.add(tuple(perm)) #tuple, lebo do setu sa daju pridat iba hashovatelne objekty
    while len(individuals) < numOfChromosomes:
        permutation = list(randomPermutations.pop())
        individuals.append(Permutation(permutation, genes.calculateFitness(permutation)))
    return individuals
```

Výber vzorky jedincov na kríženie

Na kríženie je potrebné z populácie vybrať dvoch jedincov, pričom uprednostňované by mali byť zdatnejší jedinci. V algoritme sú implementované dva spôsoby výberu jedincov na kríženie. Príslušné funkcie sú zahrnuté v triede *Population*.

Tournament selekcia

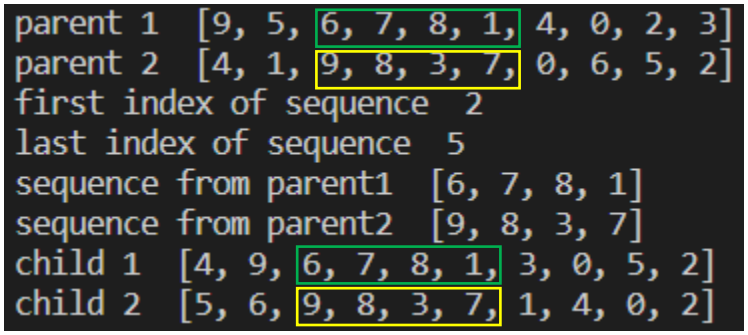
Z populácie sa vyberie vzorka jedincov, z ktorej sa následne zvolí jedinec s najväčšou hodnotou fitness. Pre potreby genetického algoritmu má parameter *numOfIndividualsRequired* vždy hodnotu dva, lebo sa krížia práve dvaja jedinci. Funkcia *tournamentSelection* náhodne vyberie z množiny indexov vzorku zadanej veľkosti, nájde najzdatnejšieho jedinca vo vzorke a uloží ho do poľa, ktoré vracia. V druhom cykle je vybratý jedinec z populácie vyradený, lebo sa majú krížiť dvaja rôzni jedinci.

Roulette wheel selekcia

Jedinci v populácii sú podľa hodnoty fitness zoradené zostupne. Najprv sa pre každého jedinca populácie vypočítajú vo funkcii *determineIntervalsOfSelectionProbability* horné hranice intervalov pravdepodobnosti jeho výberu. Všetky intervaly sú umiestnené na spoločnej osi v rozsahu od 0 do 1. Veľkosť intervalu sa počíta ako podiel hodnoty fitness jedinca a sumy fitness hodnôt všetkých jedincov v populácii; jeho hornou hranicou je súčet hornej hranice predošlého intervalu a jeho veľkosti, pričom hornou hranicou intervalu pravdepodobnosti pre najzdatnejšieho jedinca je samotná veľkosť intervalu, lebo je na osi prvý, pre jedinca s najmenšou hodnotou fitness je horná hranica vždy rovná jednej. Funkcia *rouletteWheelSelection* generuje náhodne desatinné čísla z intervalu [0; 1) a ukladá indexy tých jedincov, do ktorých intervalu vygenerované číslo spadá. Cyklus generovania čísel beží dovtedy, kým sa nevyberie požadovaný (pre genetický algoritmus je to 2) počet unikátnych indexov jedincov.

Tvorba dcérskych jedincov

Pri krížení dvoch rodičovských jedincov dochádza k rekombinácii ich génov za vzniku dvoch dcérskych jedincov, každý z ktorých obsahuje presnú sekvenciu génov z jedného rodiča a zvyšné gény sa doplňujú z druhého rodiča v tom poradí, ako sú v jeho chromozóme. Začiatok a koniec sekvencie sa volia náhodne, ale vždy tak, aby jej dĺžka predstavovala aspoň 30% a najviac 70% dĺžky celého chromozómu.



Zvolená sekvencia sa v dcérskych chromozómoch vždy umiestni na to isté miesto ako je v rodičovských.

Hľadanie optimálneho spôsobu tvorby novej generácie

Vyskúšali sa nasledovné spôsoby tvorby ďalšej generácie:

1. Nová generácia pozostáva len z krížencov
2. Nová generácia pozostáva len z krížencov; na 15% populácie sa aplikujú mutácie (jedince sa volia náhodne a môžu byť mutovaní viackrát; na jedincov s párnym indexom sa aplikuje mutácia s výmenou dvoch susediacich génov, kým pri jedincoch s nepárnym indexom sa preklopí náhodné zvolený úsek chromozómu)
3. Nová generácia je tvorená malým počtom najzdatnejších jedincov (15% populácie) a zvyšné sú kríženec
4. Nová generácia je tvorená malým počtom najzdatnejších jedincov (15% populácie) a zvyšné sú kríženec; na 15% populácie sa aplikujú mutácie (rovnako ako pri spôsobe č.2)
5. Nová generácia je tvorená malým počtom najzdatnejších jedincov (15% populácie), náhodne vygenerovaných jedincov (15%) a zvyšné sú kríženec
6. Nová generácia je tvorená malým počtom najzdatnejších jedincov (15% populácie), náhodne vygenerovaných jedincov (30%) a zvyšné sú kríženec
7. Nová generácia je tvorená malým počtom najzdatnejších jedincov (15% populácie), náhodne vygenerovaných jedincov (15%) a zvyšné sú kríženec; na 15% populácie sa aplikujú mutácie

Pri všetkých volaniach sa na vyber rodičovských párov použila roulette wheel selekcia.

Pri testovaní sa v cykle volala funkcia *geneticAlgorithm* sedemkrát s rôznymi argumentmi. Pre jednoduchšie porovnanie výsledkov dostávali v každom cykle všetky volania ako vstupné argumenty rovnakú počiatočnú populáciu a súradnice miest. V ďalšom cykle sa generovala nová inštancia problému (súradnice miest) a nová prvotná generácia.

```
for trial in range(trials):
    genes = Cities(generateCities(sizeX, sizeY, numOfCities), numOfCities)
    population = Population(numOfChromosomes, createRandomIndividuals(numOfChromosomes, numOfCities, genes))

    pop1 = copy.deepcopy(population)
    initFitness1, resultFitness1, progress1 = geneticAlgorithm(numGenerations, sizeX, sizeY, numOfCities, numOfChromosomes, matingPoolSize, eli
    sumInit1 += initFitness1
    sumResult1 += resultFitness1
    addDataPoints(resultsData1, progress1, numGenerations)

    pop2 = copy.deepcopy(population)
    initFitness2, resultFitness2, progress2 = geneticAlgorithm(numGenerations, sizeX, sizeY, numOfCities, numOfChromosomes, matingPoolSize, eli
    sumInit2 += initFitness2
    sumResult2 += resultFitness2
    addDataPoints(resultsData2, progress2, numGenerations)

    pop3 = copy.deepcopy(population)
    initFitness3, resultFitness3, progress3 = geneticAlgorithm(numGenerations, sizeX, sizeY, numOfCities, numOfChromosomes, matingPoolSize, eli
```

Pre každé zo 7 volaní funkcie *geneticAlgorithm* sa vytvorili a inicializovali premenne, ku ktorým sa po každom cykle pripočítavali príslušné návratné hodnoty.

```
sumInit1 = 0 #priemerna cena najkratsej cesty z pociatocnej generacie
sumResult1 = 0 #priemerna cena najkratsej cesty z finalnej generacie
resultsData1 = [] #priemerne hodnoty fitness pre kazdu vygenerovanu generaciu
initializeDataPoints(resultsData1, numGenerations)
```

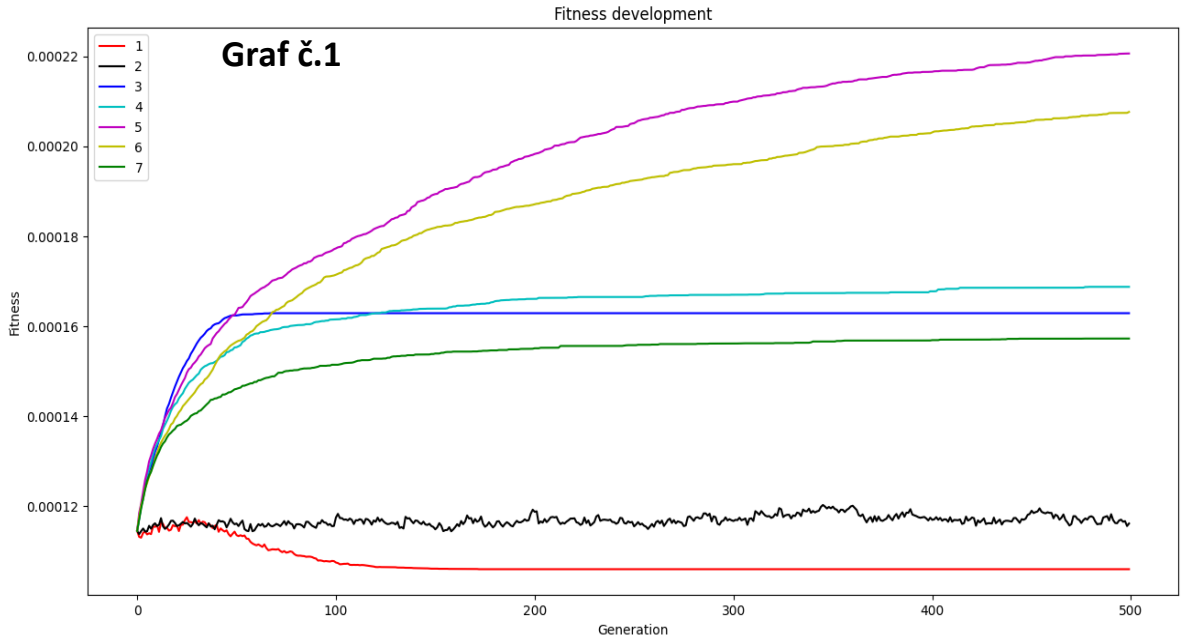
Po každom volaní vracala funkcia *geneticAlgorithm* cenu najlacnejšej cesty z počiatočnej generácie, cenu najlacnejšej cesty z finálnej generácie a zoznam fitness hodnôt najlepšieho jedinca z každej generácie. Po skončení cyklu sa hodnoty vyššie uvedených premenných vydelili počtom opakovaní cyklu (premenná *trials*), vývoj hodnoty fitness sa zobrazil do grafu a priemerná zmena ceny cesty sa vypísala do konzoly.

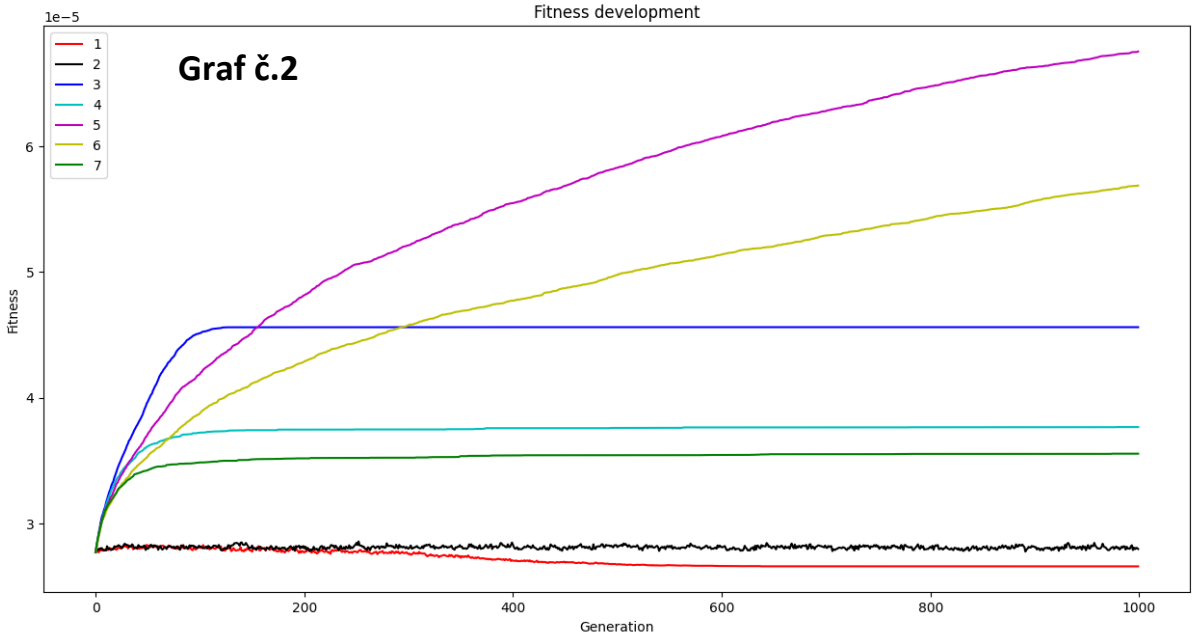
Graf č.1: Priemerne hodnoty zmien ceny cesty a priemerné hodnoty fitness pre 100 rôznych náhodne vygenerovaných inštancií problému. Počet miest je 20, veľkosť generácie je 20, počet generácií je 500, veľkosť mapy je 200x200

```
METHOD_ 1
average initial cost:  1761.2446925943023
average final cost:  1917.7347228804556
average improvement:  -156.49003028615334
METHOD_ 2
average initial cost:  1761.2446925943023
average final cost:  1750.6894084090789
average improvement:  10.5552841852234
METHOD_ 3
average initial cost:  1761.2446925943023
average final cost:  1241.3432203068921
average improvement:  519.9014722874101
METHOD_ 4
average initial cost:  1761.2446925943023
average final cost:  1203.619495450562
average improvement:  557.6251971437403
METHOD_ 5
average initial cost:  1761.2446925943023
average final cost:  918.4895012370002
average improvement:  842.755191357302
METHOD_ 6
average initial cost:  1761.2446925943023
average final cost:  978.267294721473
average improvement:  782.9773978728292
METHOD_ 7
average initial cost:  1761.2446925943023
average final cost:  1287.7275317810906
average improvement:  473.5171608132116
```

Graf č.2: Priemerne hodnoty zmien ceny cesty a priemerné hodnoty fitness pre 100 rôznych náhodne vygenerovaných inštancií problému. Počet miest je 40, veľkosť generácie je 40, počet generácií je 1000, veľkosť mapy je 200x200

```
METHOD_ 1
average initial cost:  3622.589629069866
average final cost:  3785.8617831479323
average improvement:  -163.27215407806625
METHOD_ 2
average initial cost:  3622.589629069866
average final cost:  3577.3916114781114
average improvement:  45.198017591754706
METHOD_ 3
average initial cost:  3622.589629069866
average final cost:  2210.774877193222
average improvement:  1411.814751876644
METHOD_ 4
average initial cost:  3622.589629069866
average final cost:  2669.4617627031607
average improvement:  953.1278663667053
METHOD_ 5
average initial cost:  3622.589629069866
average final cost:  1494.597442332864
average improvement:  2127.992186737002
METHOD_ 6
average initial cost:  3622.589629069866
average final cost:  1773.48263395857
average improvement:  1849.106995111296
METHOD_ 7
average initial cost:  3622.589629069866
average final cost:  2826.109547161572
average improvement:  796.4800819082939
```

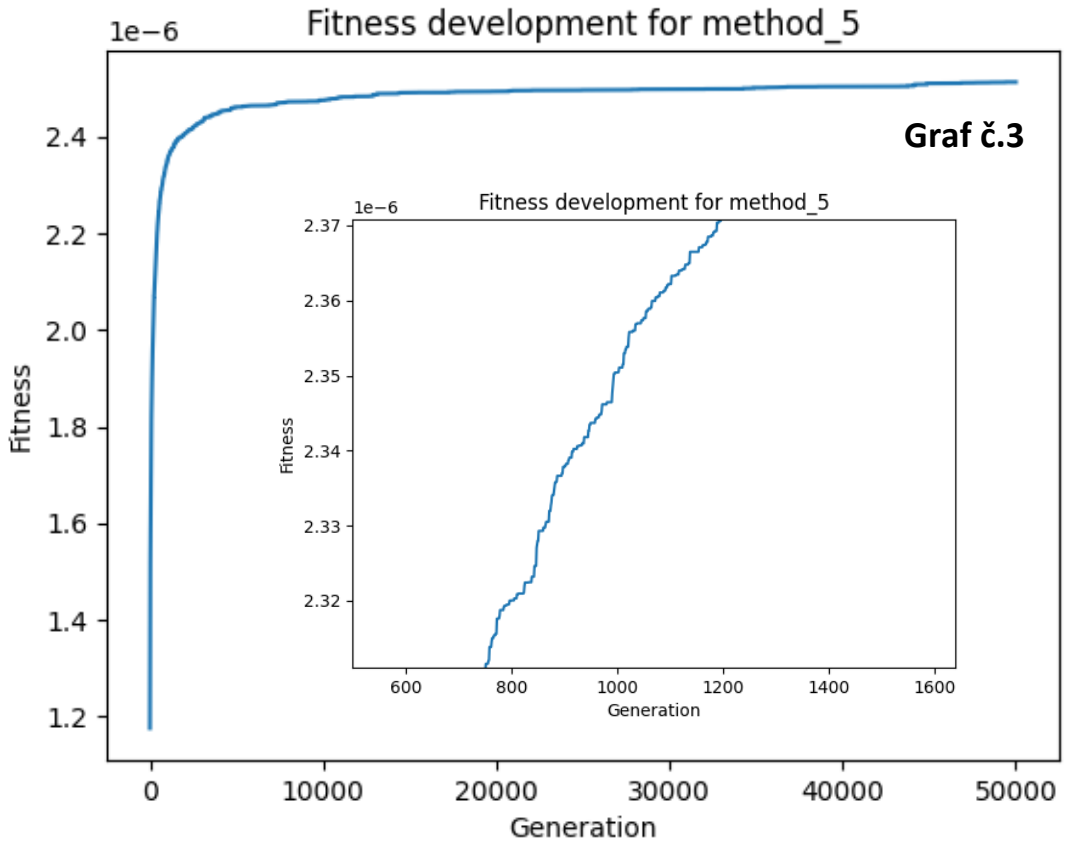




Z grafov je vidieť, že 5. spôsob generovania novej populácie vedie k najväčšiemu zníženiu ceny cesty. Zajuímavým zistením je, že samotné kríženie nevedie k vývoju populácie smerom k väčším hodnotám fitness aj napriek tomu, že spôsob selekcie rodičovských párov uprednostňuje zdatnejších jedincov. Zavedenie mutácií vedie k vzniku menších odchýlok vo fitness hodnotách jednotlivých generácií, čo je vidieť z pílovitého tvaru krivky č.2. výskyt mutácií však nie je postačujúci na vznik výrazne lepších riešení ako náhodne vygenerované počiatočné. Prenos už malého počtu najzdatnejších jedincov do ďalšej generácie vedie k nájdeniu výrazné lepšieho riešenia oproti počiatočnému. Avšak pri tomto spôsobe dochádza k rýchlej konvergencii –

približne po 50. generácii pri grafe 1 a po 100. pri grafe 2 sa hodnoty fitness ustália. Vývoj populácie pri spôsoboch č. 4 a č. 7 je podobný. Spôsob č. 5, pri ktorom sa do novej generácie zavádza okrem najzdatnejších jedincov aj malý počet náhodne vygenerovaných jedincov, sa ukázal ako najlepší. Ani po 1000. generácii nedochádza ku konvergencii. Pravdepodobne preto, že noví jedinci udržujú variabilitu populácie, a tým umožňujú preskúmať potenciálne lepšie riešenia. Podiel takýchto jedincov je populácii je významný pre nájdenie riešenia blízkeho optimálnemu. Z porovnania kriviek č. 5(15 novovytvorených jedincov v generácii) a č. 6(30 novovytvorených jedincov v generácii) je jasné, že príliš častý výskyt náhodne vygenerovaných jedincov v populácii znižuje kvalitu výsledku. Rovnakým spôsobom ju znižuje aj príliš veľká frekvencia mutácií. Spôsob č. 7 generuje výrazne horšie riešenia ako spôsob č. 5 hoci jediným rozdielom v nich je aplikovanie mutácií.

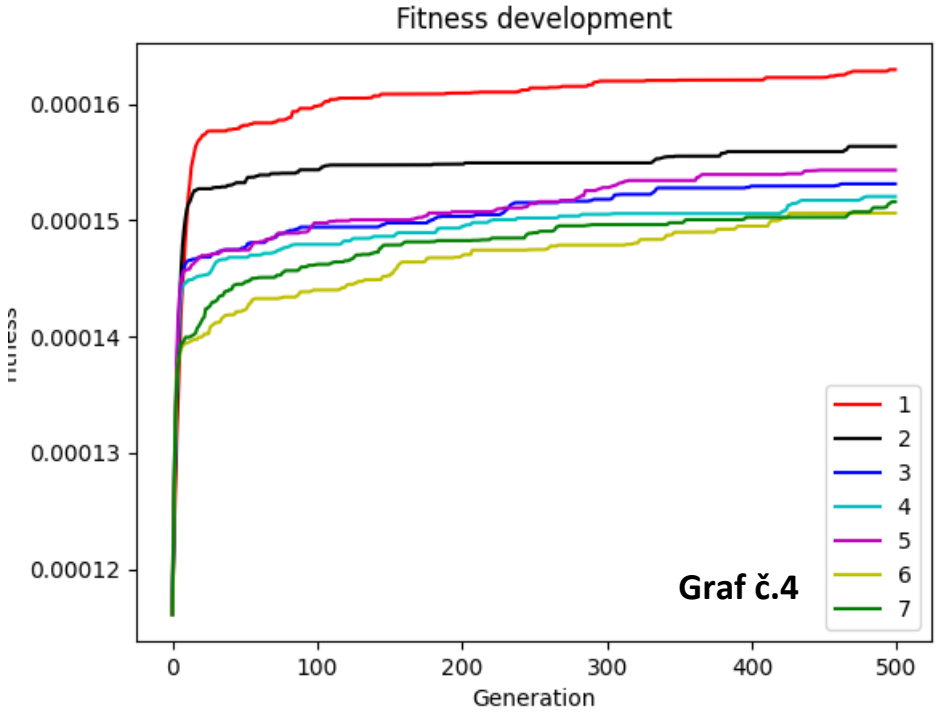
Graf č.3: Konvergencia funkcie pri použití spôsobu č. 5 na tvorbu novej generácie. Priemerne hodnoty zmien ceny cesty a hodnoty fitness pre 100 rôznych náhodne vygenerovaných inštancií problému. Počet miest je 20, veľkosť generácie je 20, počet generácií je 50000, veľkosť mapy je 200x200



Hľadanie optimálnej veľkosti vzorky pri tournament selekcii

Nová generácia sa vytvárala podľa spôsobu č. 5, rodičovské páry sa vyberali pomocou tournament selekcie s rôznou veľkosťou vzorky (3, 5, 7, 9, 11, 15, 18). čím je vzorka menšia, tým lepšie riešenia sa vygenerujú. S narastajúcou veľkosťou vzorky sa výber stáva čoraz menej náhodným a čoraz častejšie sa krížia najzdatnejší jedinci. Pri veľkostiach vzoriek blízkych veľkosti populácie má menej zdatný jedinec skoro nulovú šancu zanechať potomstvo, preto sa variabilita populácie stráca rýchlejšie, algoritmus skúma menej rozmanité riešenia a dospeje k riešeniam vzdialenejším od optimálneho než pri malých vzorkách.

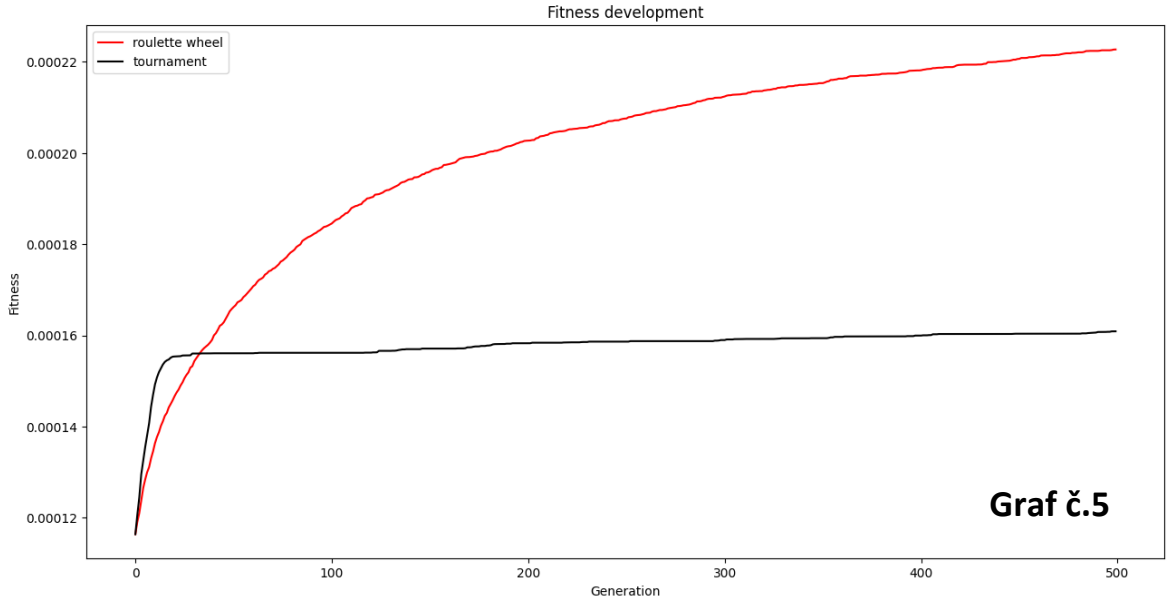
Graf č.4: Priemerne hodnoty fitness pre 100 rôznych náhodne vygenerovaných inštancií problému. Počet miest je 20, veľkosť generácie je 20, počet generácií je 500, veľkosť mapy je 200x200



Porovnanie tournament a roulette wheel selekcie

Nová generácia sa vytvárala podľa spôsobu č. 5, rodičovské páry sa vyberali pomocou roulette wheel selekcie a tournament selekcie s veľkosťou vzorky = 3. z grafu je jasné, že selekcia pomocou roulette wheel je nadradená. Výber rodičovských párov pomocou tournament selekcie vedie k rýchlej konvergencii grafu a horším riešeniam.

Graf č.5: Priemerne hodnoty fitness pre 100 rôznych náhodne vygenerovaných inštancií problému. Počet miest je 20, veľkosť generácie je 20, počet generácií je 500, veľkosť mapy je 200x200



Tabu search

Patrí medzi algoritmy heuristického hľadania a prestavuje nadstavbu horolezeckého algoritmu, ktorý z počiatočného stavu vygeneruje aplikovaním základnej operácie množinu možných riešení a v ďalšom cykle preskúma najlepšie z nich. Jeho nevýhodou je to, že dokáže rýchlo uviaznuť v lokálnom maxime, a teda nepreskúma stavy, ktoré vedú k potenciálne lepšiemu výsledku. Na predchádzanie zacykleniu sa a uviaznutiu v lokálnom maxime sa do takzvaného tabu zoznamu ukladá časť priebehu hľadania alebo naposledy vykonané operácie. Ak sa najlepšie riešenie vygenerované z aktuálneho nachádza v tabu, nebude preskúmané, s výnimkou situácie, že je lepšie ako najlepšie doteraz vygenerované riešenie.

Popis implementácie algoritmu

Stav – premutácia indexov miest; pole veľkosti n obsahujúce celé čísla z rozsahu od 0 do n-1, kde n je veľkosť *allCitiesCoords*

Uzol – objekt triedy *Permutation*, ukladá v sebe stav a hodnotu fitness

Generovanie počiatočného riešenia

Počiatočna permutácia miest sa generuje náhodne rovnako ako pri genetickom algoritme.

```
cities = Cities(generateCities(sizeX, sizeY, numOfCities), numOfCities)
currentNode = createRandomIndividuals(1, numOfCities, cities)[0]
```

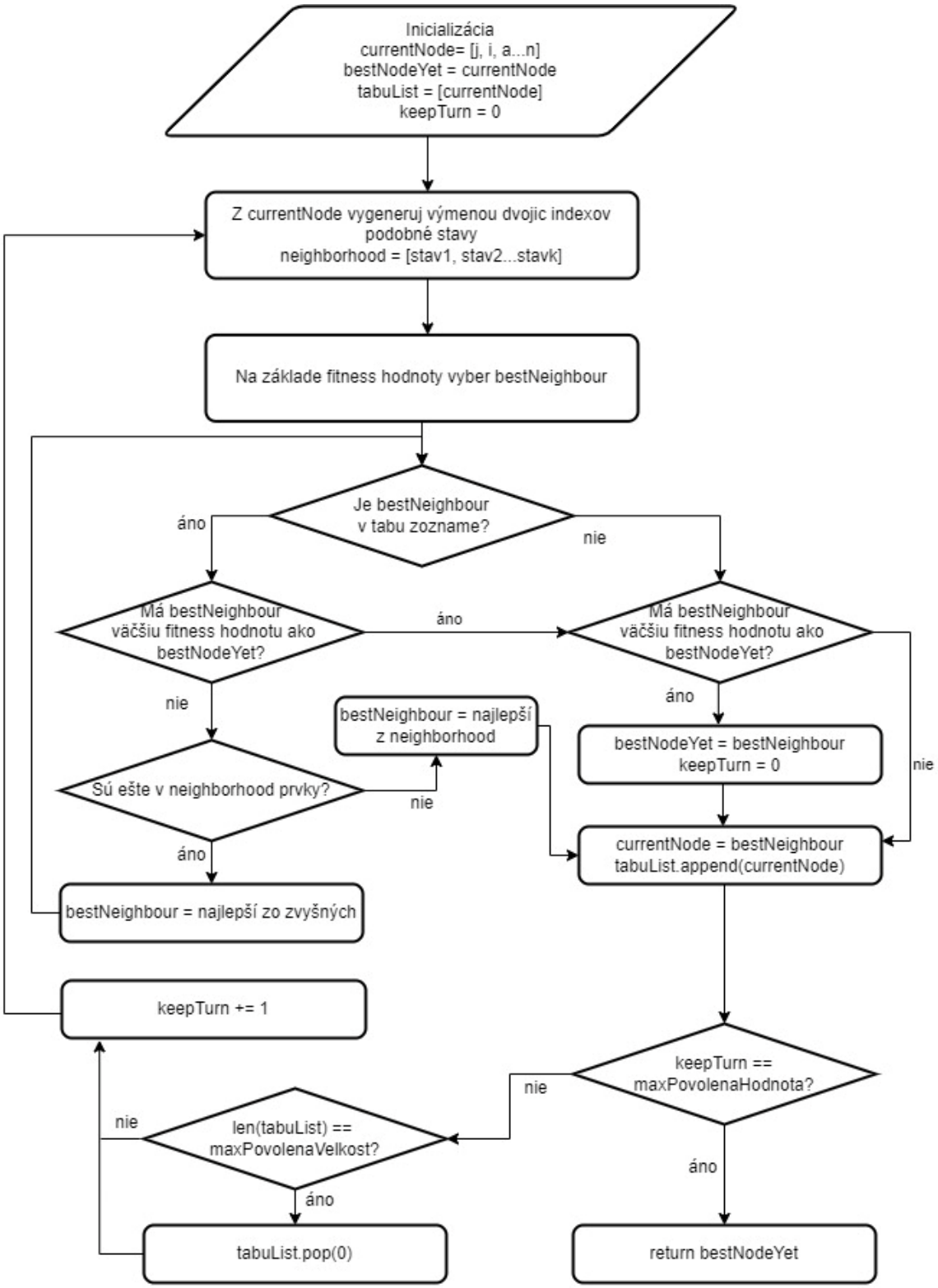
Tvorba príbuzných riešení modifikáciou aktuálneho riešenia

Operácia na vygenerovanie príbuzných riešení je výmena pozícií indexov dvoch náhodne zvolených miest, pre každé riešenie existuje n*n-1 / 2 takýchto výmen, kde n je počet miest – 1 (za počiatočný bod sa považuje 0, a tento bod sa neprehadzuje). Množina uzlov podobných aktuálnemu sa vytvára vo funkcii *generateNeighbours*, ktorá vracia množinu všetkých permutácií vzniknutých výmenou indexov každej dvojice miest, okrem počiatočného. Permutácie sú zoradené zostupne podľa fitness hodnoty (prevrátenej dĺžky cesty).

```
def generateNeighbours(node, numOfCities, genes):
    indexOfRoutStartingPoint = (node.path).index(0)
    neighborhood = list()
    for indOfElement in range(numOfCities):
        if indOfElement == indexOfRoutStartingPoint:
            continue
        for newIndOfElement in range(indOfElement + 1, numOfCities):
            neighbourPath = (node.path).copy()
            if newIndOfElement == indexOfRoutStartingPoint:
                continue
            tmp = neighbourPath[newIndOfElement]
            neighbourPath[newIndOfElement] = neighbourPath[indOfElement]
            neighbourPath[indOfElement] = tmp
            neighborhood.append(Permutation(neighbourPath, genes.calculateFitness(neighbourPath)))
    neighborhood.sort(reverse=True)
    return neighborhood
```

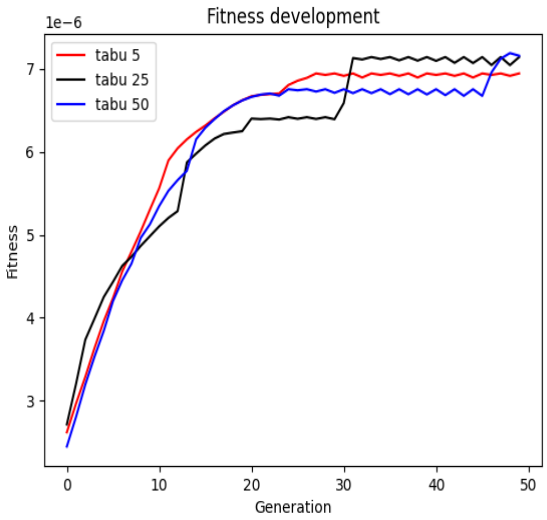
Cyklus sa vykonáva, kým sa počas n cyklov nepodarí nájsť riešenie s vyššou hodnotou fitness ako je doteraz najlepšia.

Diagram algoritmu zakázaného prehľadávania



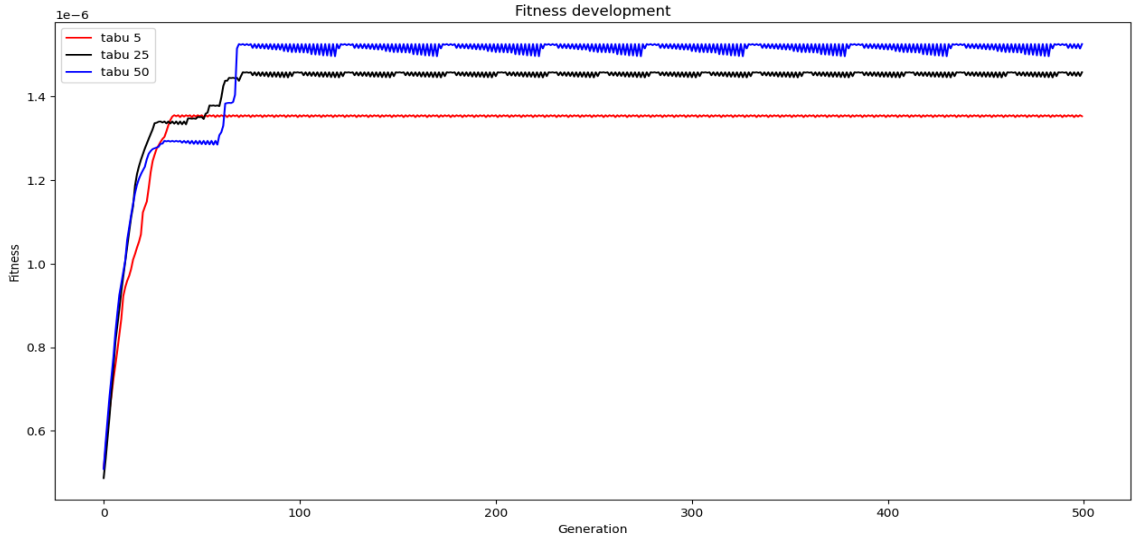
Vplyv veľkosti tabu zoznamu na priebeh a výsledok hľadania

Vývoj najlepšej hodnoty fitness pre 1 náhodne vygenerovanú inštanciu problému. Počet miest je 30, počet generácií je 50, veľkosti tabu zoznamu sú 5, 25 a 50 a veľkosť mapy je 500x500

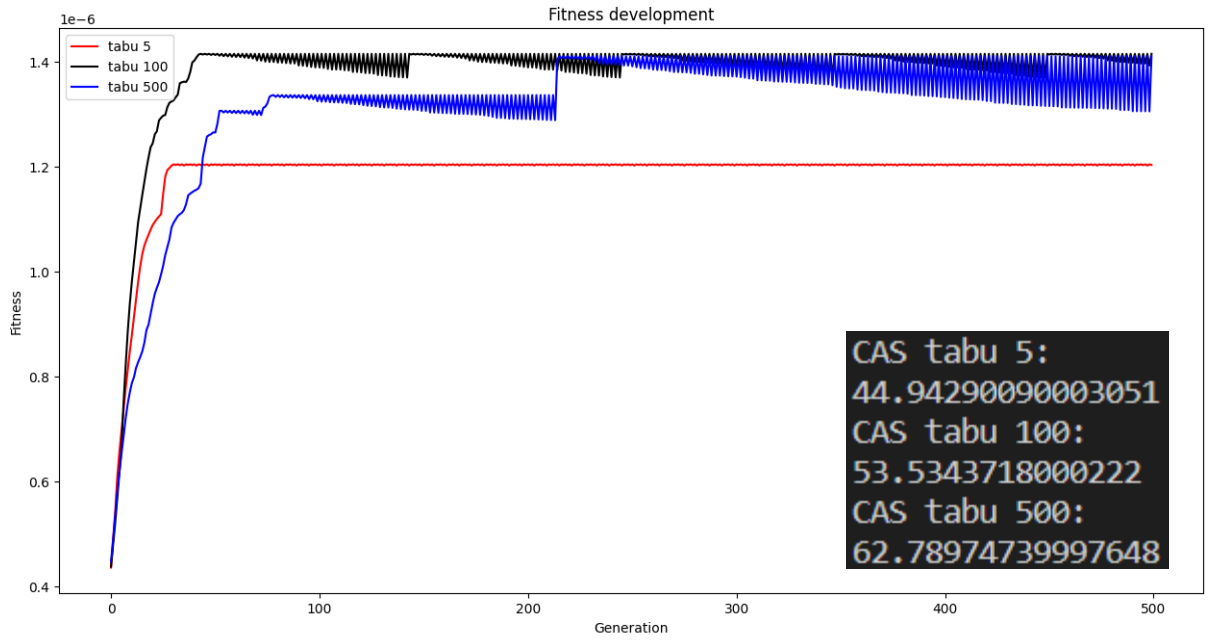



```
METHOD_ 1
average initial cost:  7627.369687871723
average final cost:  2881.421608450037
average improvement:  4745.948079421685
METHOD_ 2
average initial cost:  7360.396687714121
average final cost:  2801.7142048673295
average improvement:  4558.682482846792
METHOD_ 3
average initial cost:  8160.325771663707
average final cost:  2459.328849881341
average improvement:  5700.996921782366
```

Vývoj najlepšej hodnoty fitness pre 1 náhodne vygenerovanú inštanciu problému. Počet miest je 40, počet generácií je 500, veľkosti tabu zoznamu sú 5, 25, 50 a veľkosť mapy je 200x200



Vývoj najlepšej hodnoty fitness pre 1 náhodne vygenerovanú inštanciu problému. Počet miest je 40, počet generácií je 500, veľkosti tabu zoznamu sú 5, 100, 500 a veľkosť mapy je 200x200



Z grafov je vidieť, že počas prvých 50-100 cyklov fitness hodnota najlepšieho dosiahnutého riešenia narastá. Okolo 50. cyklu algoritmus uviazne v lokálnom maxime a istý počet cyklov sa presúva medzi niekoľkými známymi stavmi až kým sa nedopracuje k novému najlepšiemu riešeniu. Pri rovnakom vstupe a parametroch trvá vykonávanie algoritmu s väčším tabu zoznamom dlhšie. Algoritmus s malým tabu zoznamom konverguje rýchlejšie, kým algoritmus s väčším tabu zoznamom vygeneruje zvyčajne lepšie riešenie