Проблем путујућег трговца

Генетски алгоритам

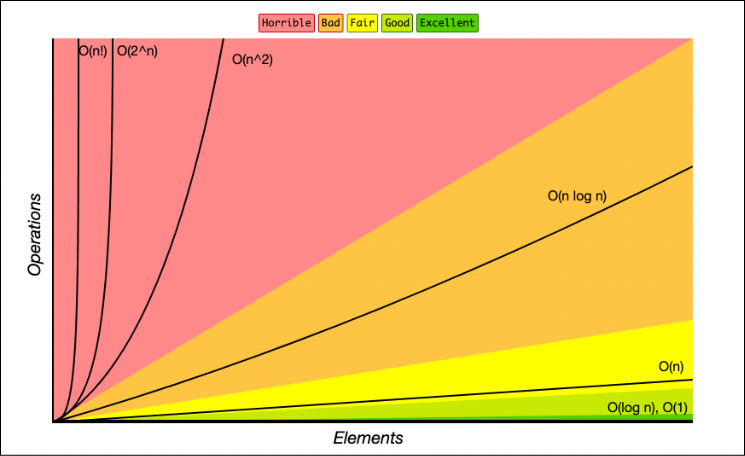
1. УВОД

Проблем путујућег трговца би се могао сажети на следећи начин: трговац има за циљ да пронађе најкраћу могућу руту са циљем да прође кроз све градове како би могао да прода своју робу. Дакле циљ је пронаћи оптималну путању у зависности од дужине пута, односно броја градова и њиховог међусобног растојања.

Овај проблем је могуће решавати такозваном методом „Исцрпне претраге“ која је од раније позната како у свакодневном животу тако и у рачунарству. Међутим ова метода је у употреби само када је величина проблема над којим се иста примењује ограничена, или у околностима када је једноставност имплементације важнија од брзине, те није зајамчена оптималност или савршеност, али је довољна за испуњавање непосредних циљева.

1. АНАЛИЗА ПРОБЛЕМА

Дакле, уколико је број градова које трговац треба да посети мали, рецимо 4 (градови А-Б-В-Г), укупан број комбинација износи n!, односно 24. Повећам броја градова повећава се и комплексност алгоритма, што је и приказано на дијаграму сложености (*Сл. 1*)



Сл. 1: Дијаграм сложености

Према томе потребно је пронаћи ефикаснији и практичнији начин за решавање проблема овакве сложености, а то је генетски алгоритам (у даљем тексту ГА).

Комплексност генетског алгоритма можемо приказати као:

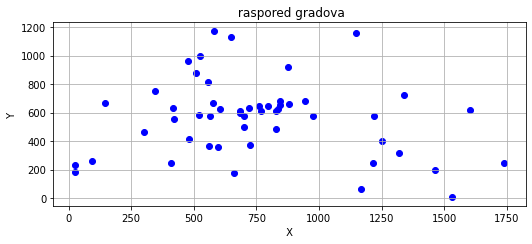
*g\*n\*m*

При чему је g – број генерација, n – величина популације и m – индивидуална величина својствена бази података. Како база коришћена у овом задатку поседује m = 52 тачке/града са одговарајућим координатама, то би значило да треба рачунати 52! различитих комбинација. Међутим, уколико параметре од којих зависи ГА поставимо на рецимо g = 100, n = 100, то би значило да је сада тај број ограничен на 520.000. Из овог примера видимо да је употреба ГА најбоље и најефикасније решење.

1. СТРУКТУРА ПРОГРАМА

С обзиром на то да ГА представља део и веома важан концепт вештачке интелигенције (AI) у погледу најбољих перформанси у тражењу оптималних решења, а како је управо *python* један од најпожељнијих алата за AI, за решење проблема Путујућег трговца коришћен је управо *python* као програмски језик. Kôд је због практичности писан у Google Colab-у.

Библиотеке које су коришћене приликом писања кôда су следеће: *random*, *numpy*, *matplotlib*.*pyplot* и *math*. Следећи корак је била визуализација (Сл. 2), односно приказ градова чије су координате преузете из датотеке *data\_tsp.txt*.



Сл. 2: Приказ распореда градова

Програм је структуиран тако да се састоји из три веће целине, a то су целина која се односи на хромозоме, целина које се односи на саме операторе ГА и MAIN, део у коме се извршава главни део кôда.

У првој целини креирамо *Node* класу у којој повезујемо атрибуте са датим аргументима (*self* представља инстанцу класе). Након тога приступамо датотеци, креирамо *data list* која ће касније бити употребљена у функцијама које се користе у ГА. Уклањамо непотребне размаке уколико постоје, након чега креирамо *Node* објекат. У овом делу дефинисан је и број *N* који је једнак укупном броју индекса из датотеке. У овом делу је такође дефинисана и фунцкија *kreiraj\_matricu\_udaljenosti* јер је потребно познавање удаљености између локација како би се креирао ГА. У оквиру ове функције одмах на почетку дефинисана је и матрица нула величине *NxN* која има за циљ избегавање сувишног рачунања и из ње касније добијамо информацију о удаљености. Затим помоћу две *for* петље рачунамо еуклидско растојање између сваке тачке по формули датој у задатку. У овој целини дефинисана је и *Chromosome* класа како бисмо сазнали удаљеност и *Fitness* вредности… Удаљеност се рачуна помоћу матрице, а удаљеност сваког хромозома је *cost* тог хромозома. И на крају, *Fitness* вредност је дефинисана као 1/*cost,* како бисмо касније у фази селекције изабрали хромозоме са већом *Fitness* вредношћу.

У другој целини груписани су оператори за ГА. Прва функција креирана у овој целини је *kreiraj\_random\_listu* која служи за креирање *random* хромозома. Важно је напоменути да почетна тачка мора уједно да буде и крајња. Наредна функција је функција иницијализације која као параметре прима *data* и *population\_size* и помоћу које прва генерација иницијалне популације бива спремна. Након тога следи можда и најважнија функција, а то је *selekcija* која врши одабир хромозома од родитеља како би се креирали хромозоми потомака. Практично, унутар ове функције правимо надметање између свих хромозома у популацији. У почетку креиране су насумичне четири int вредности (4 „улазнице“ за надметање) након чега бирамо 4 хромозома односно кандидата, при чему је број улазнице индекс од листе популације. Следећи корак је одабир победничког хромозома, који се бира помоћу *Fitness* вредности. У овој целини дефинисана је и *crossover* функција. Коришћен је метод такозваног тачкастог укрштања који подразумева креирање насумичног breakpoint-а. Први део хромозома потомка је идентичан хромозому првог родитеља до дефинисаног breakpoint-а. Након тога потомку се додају преостали хромозоми другог родитеља. Наредни веома значајан оператор ГА је мутација. На први поглед функција *mutacija* делује једноставно. У њој се генеришу два насумична индекса која служе за замену два чвора хромозома. Значај овог оператора лежи у томе што он обезбеђује разноврсност решења и служи као превентива. Следећа *find\_best* функција проверава све хромозоме у генерацији и враћа нам најбољег. Последња функција ове целине је *kreiranje\_nove\_generacije* која као параметре прима стопу мутације и претходну генерацију (*prethodna\_generacija, mutation\_rate*). Пре креирања скроз нове генерације, проналазимо најбољи хромозом претходне генерације и директно га додељујемо новој генерацији. Овај поступак је познат као елитизам. У *for* петљи генеришемо нови хромозом на следећи начин: одабирамо два родитељска хромозома користећи функцију *selekcija* и креирамо два дечија хромозома, након чега уколико је насумично генерисан број мањи од стопе мутације (*mutation\_rate*) имплементирамо мутацију на првог потомка, у супротном прескачемо оператор мутације и додељујемо хромозоме новој генерацији.

Трећа целина је део у коме се извршава већи део кода. У њему су дефинисани параметри за ГА. Главна функција читавог програма је *genetski\_algoritam*. Унутар ње креирамо прву генерацију помоћу функције иницијализације док се у *for* петљи одвија остатак. Приликом сваког уласка у петљу креира се нова генерација у зависности од претходне. Ова функција нам враћа генерацију решења и *costs\_for\_plot*. Наредне две функције служе за исцртавање графика.

1. КРИТЕРИЈУМ ОПТИМАЛНОСТИ

Критеријум оптималности проблема путујућег трговца је сума удаљености свака два узастопна града кроз које је трговац прошао.

1. НАЧИН ИМПЛЕМЕНТАЦИЈЕ ОПЕРАТОРА

Начин имплементације свих оператора је описан у III поглављу овог извештаја, као и у коментарима у приложеном кôду

.

1. ПАРАМЕТРИ АЛГОРИТМА

Параметри алгоритма су дати у табели испод

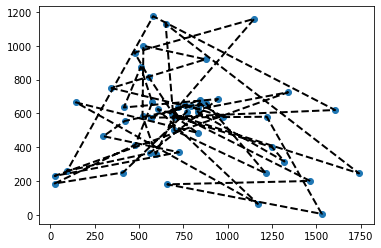
Табела 1: Параметри ГА

|  |  |  |  |
| --- | --- | --- | --- |
|  | Назив параметра | Ознака у коду | Вредност |
|  | Број генерација | g | 200 |
|  | Величина популације | n | 200 |
|  | Стопа мутације | mutation\_rate | 0,1 |
|  |  |  |  |

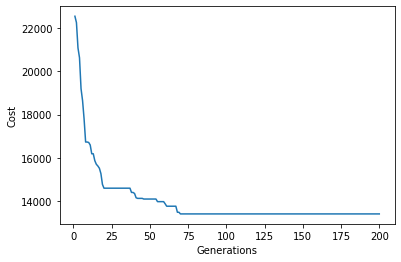
1. РЕЗУЛТАТИ

За наведене параметре генетског алгоритма добијено је решење(*Сл. 3*):

Пређени пут: 13414.35



Сл. 3: Визуализација путање



Сл. 4: Конвергенција

На основу графика са слике 4 (*Сл. 4*) може се закључити да је алгоритам конвергирао на минимум након седамдесетак итерација.