**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. ІГОРЯ СІКОРСЬКОГО”**
**Інститут прикладного системного аналізу**
**Кафедра системного проектування**

# ЗВІТ

з виконання лабораторної роботи №1
з дисципліни “Методи та системи штучного інтелекту”
на тему: “Рішення задачі комівояжера за допомогою класичного генетичного алгоритму”

Виконав:

Студент 4 курсу групи ДА-82

Муравльов А. Д.

Варіант №18

Київ — 2021

***Мета роботи:*** розглянути ідею, основні принципи та етапи реалізації генетичного алгоритму на прикладі рішення класичної задачі комівояжера, та оцінити його ефективність.

***Завдання:***

1. Реалізувати генетичний алгоритм для пошуку найкоротшого шляху між містами згідно варіанту.

2. Проаналізувати ефективність алгоритму та його збіжність на основі виконаної роботи.

| **№ Варіанта** | **Тип кросоверу** | **Оператор вибору батьківської пари** |
|---|---|---|
| 8 | багатоточковий | панміксія |

**Хід роботи**

***Напишемо програму для реалізації генетичного алгоритму.***
Програма розроблена у середовищі Visual Studio Code, з використанням плагіну jupyter notebook. У якості мови програмування обрано python 3.10.

***Блок схема генетичного алгоритму з урахуванням типу оператору вибору батьків (згідно варіанту), виду кросоверу та мутації:***


Початок
Створення початкової популяції
Оцінка пристосованності початкової популяції
Селекція індивідів
Багатоточковий кроссинговер
Мутація
Створення нової популяції з використанням панміксії
Оцінка пристосованності нової популяції
Перевірка умови останову
0
1
Вибір найкращої особини
Кінець

### *Хід написання програми та лістинг*

Підхід до написання програми, та генетичний алгоритм в контексті задачі комівояжера:

- Ген: місто, що задається координатами (x,y)
- Індивід (хромосома): один шлях, що задовольняє умови задачі комівояжера
- Популяція: набір можливих шляхів
- Батьки: два шляхи що поєднуються для створення нового
- Mating pool: колекція батьків, що використовуються для створення наступної популяції
- Пристосованість: функція, що визначає наскільки короткий кожен шлях
- Мутація: метод для внесення варіацій до популяції
- Елітизм: метод відбору найкращих індивідів до наступного покоління

Створення початкової популяції:

```
def create_starting_population(size,Number_of_city):
    '''Method create starting population
    size= No. of the city
    Number_of_city= Total No. of the city
    '''
    population = []
    for i in range(0,size):
        population.append(create_new_member(Number_of_city))
    return population
```

Оцінка пристосованості популяції:

```
def fitness(route,CityList):
    '''Individual fitness of the routes is calculated here
    route= 1d array
    CityList = List of the cities
    '''
    #Calculate the fitness and return it.
    score=0
    #N_=len(route)
    for i in range(1,len(route)):
```

```
        k=int(route[i-1])
        l=int(route[i])
        score = score + distance(CityList[k],CityList[l])
    return score
```

Селекція індивідів:

```
def selection(popRanked, eliteSize):
    selectionResults=[]
    result=[]
    for i in popRanked:
        result.append(i[0])
    for i in range(0,eliteSize):
        selectionResults.append(result[i])

    return selectionResults
```

Кроссинговер:

```
def crossover(a,b):
    '''
    cross over
    a=route1
    b=route2
    return child
    '''
    child1=[]
    child2=[]
    childA=[]
    childB=[]
    childC=[]
    childD=[]
    child1_2=[]
    child2_2=[]
    childA2=[]
    childB2=[]
    childC2=[]
    childD2=[]
```

```
geneA=int(random.random()* len(a))
geneB=int(random.random()* len(a))

start_gene=min(geneA,geneB)
end_gene=max(geneA,geneB)

for i in range(start_gene,end_gene):
    childA.append(a[i])
for i in range(start_gene,end_gene):
    childC.append(b[i])
childB=[item for item in a if item not in childA]
childD=[item for item in b if item not in childC]
child1=childA+childB
child2=childC+childD
#second
gene1 = int(random.random()* (len(a)-end_gene))
gene2 = int(random.random()* (len(a)-end_gene))
start_gene2=min(gene1,gene2)+end_gene
end_gene2=max(gene1,gene2)+end_gene

for i in range(start_gene2,end_gene2):
    childA2.append(child1[i])

for i in range(start_gene2,end_gene2):
    childC2.append(child2[i])

childB2=[item for item in child1 if item not in childA2]
childD2=[item for item in child2 if item not in childC2]
child1_2=childA2+childB2
child2_2=childC2+childD2

return (child1_2, child2_2)
```

Мутація:

```
def mutate(route,probablity):
    '''
    mutation
    route= 1d array
    probablity= mutation probablity
    '''
    #for mutating shuffling of the nodes is used
    route=np.array(route)
    for swaping_p in range(len(route)):
        if(random.random() < probablity):
            swapedWith = np.random.randint(0,len(route))

            temp1=route[swaping_p]

            temp2=route[swapedWith]
            route[swapedWith]=temp1
            route[swaping_p]=temp2

    return route
```

Створення нової популяції:

```
def next_generation(City_List,current_population,mutation_rate,elite_size):
    population_rank=rankRoutes(current_population,City_List)
    #print(f"population rank : {population_rank}")
    selection_result=selection(population_rank,elite_size)
    #print(f"selection results {selection_result}")
    mating_pool=matingPool(current_population,selection_result)
    #print(f"mating pool {mating_pool}")

    children=breedPopulation(mating_pool)
    #print(f"childern {children}")

    next_generation=mutatePopulation(children,mutation_rate)
    #print(f"next_generation {next_generation}")
    return next_generation
```

Перевірка точки останову:

```
def
genetic_algorithm(City_List,size_population=1000,elite_size=70,mutation_Rate=0.01,gene
ration=2000):
    '''size_population = 1000(default) Size of population
        elite_size = 75 (default) No. of best route to choose
        mutation_Rate = 0.05 (default) probablity of Mutation rate [0,1]
        generation = 2000 (default) No. of generation
    '''
    pop=[]
    progress = []

    Number_of_cities=len(City_List)

    population=create_starting_population(size_population,Number_of_cities)
    progress.append(rankRoutes(population,City_List)[0][1])
    print(f"initial route distance {progress[0]}")
    print(f"initial route {population[0]}")
    for i in range(0,generation):
        pop = next_generation(City_List,population,mutation_Rate,elite_size)
        progress.append(rankRoutes(pop,City_List)[0][1])



    rank_=rankRoutes(pop,City_List)[0]

    print(f"Best Route :{pop[rank_[0]]} ")
    print(f"best route distance {rank_[1]}")
    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()

    return rank_, pop
```

Вибір найкращої особини:

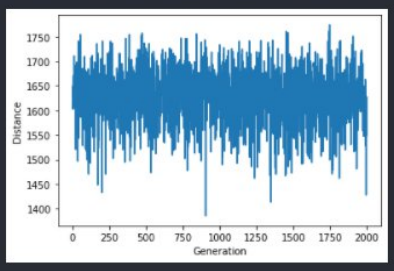
```
def rankRoutes(population,City_List):

    fitnessResults = {}

    for i in range(0,len(population)):

        fitnessResults[i] = fitness(population[i],City_List)

    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse =

False)
```

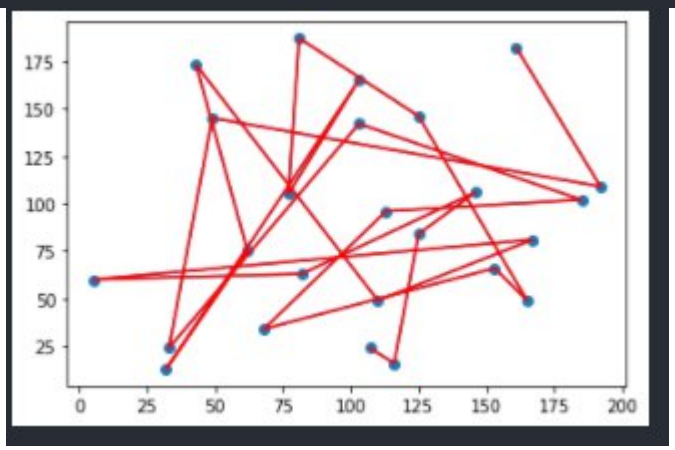
***Результати роботи програми:***

```
rank_,pop=genetic_algorithm(City_List=cityList)
```

✓ 5m 36.8s

```
initial route distance 1631.17349916659
initial route [19, 14, 0, 3, 18, 8, 17, 9, 20, 23, 13, 11, 1, 4, 2, 12, 21, 16, 6, 10, 7, 15, 22, 24, 5]
Best Route :[15  4  1 18  9  8  2 16 11 13  5  3 19 23  7 14 12 10  0 21 24  6 17 22
 20]
best route distance 1616.9292773987597
```



**Висновки:**

Ми розглянули ідею, основні принципи та етапи реалізації генетичного алгоритму на прикладі рішення класичної задачі комівояжера. Ефективніть даного алгоритму досить висока, але все ж можливі похибки через певні фактори.

**Задача комівояжера** полягає у наступному: комівояжер повинен проїхати *n* міст. Для того, щоб зменшити витрати, він повинен побудувати маршрут таким чином, щоб побувати в кожному місті по одному разу і повернутися у початкове (тобто знайти найкоротший шлях.

**Суть «генетичного» підходу у рішенні задачі комівояжера** в тому, що спочатку генерується деяка невелика множина маршрутів - початковою популяцією. Далі до маршрутів цієї початкової множини застосовуються операції вибору батьківської пари, схрещування і мутації.

Основний (класичний) генетичний алгоритм складається з наступних кроків:

1. ініціалізація, або вибір вихідної популяції особин;
2. оцінка пристосованості особин в популяції;
3. перевірка умови зупинки алгоритму;
4. схрещування (двоточкове);
5. застосування генетичних операторів: схрещування (двоточкове) і мутації;
6. формування нової популяції з урахуванням оператору вибору батьківської пари з використанням аутбридінгу;
7. перевірка умови зупинки алгоритму;
8. вибір «найкращої» особини.

У класичному генетичному алгоритмі застосовуються два основних генетичних оператора: оператор схрещування (crossover) та оператор мутації (mutation). Однак слід зазначити, що оператор мутації грає явно другорядну роль в порівнянні з оператором схрещування. Це означає, що схрещування в класичному генетичному алгоритмі здійснюється практично завжди, тоді як мутація — досить рідко. Вірогідність схрещування, як правило, досить велика (звичайно $0,5 < p_c < 1$), тоді як імовірність мутації встановлюється дуже малою (найчастіше $0 < p_m < 0,1$). Це випливає з аналогії зі світом живих організмів, де мутації відбуваються надзвичайно рідко.

За варіантом реалізовано багатоточковий кросинговер та використано панміксію.

Генетичні алгоритми на практиці застосовюються для вирішення наступних задач:

1. Оптимізація функцій
2. Оптимізація запитів в базах даних
3. Різноманітні задачі на графах (задача комівояжера, розфарбування)
4. Налаштування і навчання штучної нейронної мережі
5. Задачі компоновки
6. Створення розкладів
7. Ігрові стратегії