**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 5 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*Скрипець Ольга Олександрівна*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc52291748)

[2 Завдання 4](#_Toc52291749)

[3 Виконання 10](#_Toc52291750)

[3.1 Покроковий алгоритм 10](#_Toc52291751)

[3.2 Програмна реалізація алгоритму 10](#_Toc52291752)

[3.2.1 Вихідний код 10](#_Toc52291753)

[3.2.2 Приклади роботи 10](#_Toc52291754)

[3.3 Тестування алгоритму 11](#_Toc52291755)

[Висновок 12](#_Toc52291756)

[Критерії оцінювання 13](#_Toc52291757)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

# Завдання

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

* обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);
* зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;
* після цього параметр фіксується і змінюються інші параметри;
* далі повторюємо процедуру спочатку, з першого зафіксованого параметру;
* зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

|  |  |
| --- | --- |
| **№** | **Задача** |
| 1 | **Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.  Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика. |
| 2 | **Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.  **Розглядається симетричний, асиметричний та змішаний варіанти.**  В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.  У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.  У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.  Застосування:   * доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); * доставка води; * моніторинг об'єктів; * поповнення банкоматів готівкою; * збір співробітників для доставки вахтовим методом. |
| 3 | **Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число.  Застосування:   * розкладу для освітніх установ; * розкладу в спорті; * планування зустрічей, зборів, інтерв'ю; * розклади транспорту, в тому числі - авіатранспорту; * розкладу для комунальних служб; |
| 4 | **Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S.  Задача вершинного покриттяполягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).  На вході: Граф G = (V, E).  Результат: множина C ⊆ V - найменше вершинне покриття графа G.    Застосування:   * розміщення пунктів обслуговування; * призначення екіпажів на транспорт; * проектування інтегральних схем і конвеєрних ліній. |
| 5 | **Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній.  Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).  Застосування:   * біоінформатика; * електротехніка; |
| 6 | **Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.  Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

|  |  |
| --- | --- |
| **№** | **Алгоритми і досліджувані параметри** |
| 1 | **Генетичний алгоритм:**   * оператор схрещування (мінімум 3); * мутація (мінімум 2); * оператор локального покращення (мінімум 2). |
| 2 | **Мурашиний алгоритм**:   * α; * β; * ρ; * Lmin; * кількість мурах М і їх типи (елітні, тощо…); * маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**   * кількість ділянок; * кількість бджіл (фуражирів і розвідників). |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 1 | Задача про рюкзак + Генетичний алгоритм |
| 2 | Задача про рюкзак + Бджолиний алгоритм |
| 3 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 4 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 5 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 6 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 7 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 8 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |
| 9 | Задача вершинного покриття + Генетичний алгоритм |
| 10 | Задача вершинного покриття + Бджолиний алгоритм |
| 11 | Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм |
| 12 | Задача комівояжера (симетрична мережа) + Бджолиний алгоритм |
| 13 | Задача комівояжера (змішана мережа) + Бджолиний алгоритм |
| 14 | Розфарбовування графа + Генетичний алгоритм |
| 15 | Розфарбовування графа + Бджолиний алгоритм |
| 16 | Задача про кліку (задача розпізнавання) + Генетичний алгоритм |
| 17 | Задача про кліку (задача розпізнавання) + Бджолиний алгоритм |
| 18 | Задача про кліку (обчислювальна задача) + Генетичний алгоритм |
| 19 | Задача про кліку (обчислювальна задача) + Бджолиний алгоритм |
| 20 | Задача про найкоротший шлях + Генетичний алгоритм |
| 21 | Задача про найкоротший шлях + Мурашиний алгоритм |
| 22 | Задача про найкоротший шлях + Бджолиний алгоритм |
| 23 | Задача про рюкзак + Генетичний алгоритм |
| 24 | Задача про рюкзак + Бджолиний алгоритм |
| 25 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 26 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 27 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 28 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 29 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 30 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |

# Виконання

## Покроковий алгоритм

1. Створення початкової популяції:

1.1Випадковим чином генеруємо множину можливих маршрутів (шляхів)

1.2Розраховуємо довжину (вартість) кожного маршруту

2. Цикл ітерацій:

2.1 Відбір:

2.1.1Вибираємо 10% маршрутів з найменшою довжиною (найкращих)

2.1.2 Переносимо 20% кращих маршрутів без змін у наступне покоління

2.2 Схрещування:

2.2.1 Випадковим чином обираємо 2 маршрути з решти 80% (з вірогідністю, обернено пропорційною довжині)

2.2.2 Складаємо новий маршрут ("дитина") з 2 батьківських:

2.2.2.1 З першого маршруту беремо елементи з кожного другого сегмента, зберігаючи їх порядок

2.2.2.2 Всі пропущені місця заповнюємо елементами з другого маршруту, які ще не з'явились в "дитин", зберігаючи їх порядок

2.3 Мутація:

2.3.1 З заданою ймовірністю для "дитини":

2.3.1.1 Вибираємо 2 випадкові елементи маршруту

2.3.1.2 Міняємо їх місцями

2.4 Оцінка:

2.4.1 Розраховуємо довжину (вартість) нового маршруту

2.4.2 Додаємо його до нового покоління

2.5 Повторення циклу

## Програмна реалізація алгоритму

### Вихідний код

import random  
from pathlib import Path  
  
class GeneticAlgorithm:  
 PopulationSize = 1000  
 BreedingPoolSize = 100  
 def \_\_init\_\_(self, batch\_number, probability, problem):  
 self.problem = problem  
 self.mutation\_probability = probability  
 self.batch\_size = self.problem.CityNumber // batch\_number  
 self.current\_generation = self.get\_initial\_population()  
  
 def get\_best\_solution(self):  
 return min(self.current\_generation, key=lambda s: s.cost)  
  
 def iterate(self):  
 breeding\_pool = sorted(self.current\_generation, key=lambda s: s.cost)[:self.BreedingPoolSize]  
 best = breeding\_pool[:self.BreedingPoolSize // 5]  
 probabilities = self.get\_probabilities(breeding\_pool)  
 new\_population = []  
  
 for i in range(self.problem.CityNumber):  
 if i < self.BreedingPoolSize // 5:  
 new\_population.append(best[i])  
 else:  
 new\_population.append(self.cross(breeding\_pool[self.choose\_solution(probabilities)],  
 breeding\_pool[self.choose\_solution(probabilities)]))  
 if random.random() < self.mutation\_probability:  
 self.mutate(new\_population[i])  
  
 self.current\_generation = new\_population  
  
 def mutate(self, solution):  
 first = random.randint(0, len(solution.path) - 1)  
 second = random.randint(0, len(solution.path) - 1)  
 solution.path[first], solution.path[second] = solution.path[second], solution.path[first]  
  
 def cross(self, first, second):  
 first\_genes = [first.path[self.batch\_size \* 2 \* (i // self.batch\_size) + i % self.batch\_size]  
 for i in range(self.problem.CityNumber // 2)]  
  
 path = [0] \* self.problem.CityNumber  
 current\_second = 0  
 for i in range(self.problem.CityNumber):  
 if (i // self.batch\_size) % 2 == 0:  
 path[i] = first.path[i]  
 else:  
 while first\_genes.count(second.path[current\_second]):  
 current\_second += 1  
 path[i] = second.path[current\_second]  
 current\_second += 1  
 return Solution(self.problem, path)  
  
 def choose\_solution(self, probabilities):  
 rand = random.random()  
 total = 0.0  
 for i, prob in enumerate(probabilities):  
 total += prob  
 if total > rand:  
 return i  
 return len(probabilities) - 1  
  
 #обчислює ймовірності вибору кожного рішення для формування нового покоління.  
 def get\_probabilities(self, solutions):  
 values = [1.0 / s.cost for s in solutions]  
 total = sum(values)  
 return [val / total for val in values]  
  
 def get\_initial\_population(self):  
 return [Solution(self.problem, list(range(self.problem.CityNumber))).shuffle()  
 for \_ in range(self.PopulationSize)]  
  
  
class Problem:  
 FileName = "problem.txt"  
 CityNumber = 300  
  
 def \_\_init\_\_(self):  
 self.initialize\_matrix()  
  
 def get\_cost(self, path):  
 solution = 0  
 for i in range(self.CityNumber - 1):  
 solution += self.get\_distance(path[i], path[i + 1])  
 solution += self.get\_distance(path[self.CityNumber - 1], path[0])  
 return solution  
  
 def get\_distance(self, source, destination):  
 return self.matrix[source][destination]  
  
 def find\_optimal\_solution(self):  
 solutions = [self.get\_cost(self.construct\_path(j)) for j in range(self.CityNumber)]  
 return min(solutions)  
  
 def construct\_path(self, start\_node):  
 nodes = list(range(self.CityNumber))  
 current\_node = start\_node  
 path = [current\_node]  
 for \_ in range(self.CityNumber - 1):  
 node = current\_node  
 current\_node = min((x for x in nodes if x not in path), key=lambda x: self.get\_distance(node, x))  
 path.append(current\_node)  
 return path  
  
 def initialize\_matrix(self):  
 if Path(self.FileName).is\_file():  
 with open(self.FileName, 'r') as f:  
 lines = f.readlines()  
 self.matrix = [list(map(lambda x: float('inf') if x == 'inf' else int(x), line.split())) for line in lines]  
 return  
 self.generate\_matrix()  
 with open(self.FileName, 'w') as f:  
 for row in self.matrix:  
 f.write(" ".join(map(str, row)) + "\n")  
  
 def generate\_matrix(self):  
 self.matrix = [[0] \* self.CityNumber for \_ in range(self.CityNumber)]  
 for i in range(self.CityNumber):  
 for j in range(i + 1, self.CityNumber):  
 self.matrix[i][j] = random.randint(5, 150)  
 self.matrix[j][i] = self.matrix[i][j]  
 for i in range(self.CityNumber):  
 self.matrix[i][i] = float('inf')  
  
  
class Solution:  
 def \_\_init\_\_(self, problem, path):  
 self.problem = problem  
 self.path = path  
 self.cost = problem.get\_cost(path)  
  
 def shuffle(self):  
 random.shuffle(self.path)  
 self.cost = self.problem.get\_cost(self.path)  
 return self  
  
 def \_\_lt\_\_(self, other):  
 return self.cost < other.cost  
  
  
problem = Problem()  
batch\_number = 4  
probability = 0.3  
algorithm = GeneticAlgorithm(batch\_number, probability, problem)  
with open("result.txt", "w") as file:  
 for i in range(500):  
 for j in range(20):  
 algorithm.iterate()  
  
 line = f"{i \* 20 + 20} {algorithm.get\_best\_solution().cost}\n"  
 print(line)  
  
 file.write(line)

### Приклади роботи

На рисунках 3.1 показано приклад роботи програми.

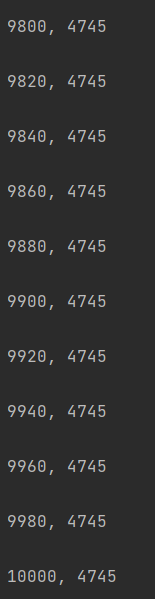


Рисунок 3.1

Тестування алгоритму

Щодо дослідження генетичного алгоритму для задачі комівояжера, основні аспекти включають ефективність різних операторів (схрещування, мутації, локального покращення), вплив розміру популяції та інших параметрів на якість знайдених рішень.

Спочатку я обрала кількість ітерацій на яких буду тестувати алгоритм. У моєму випадку це 10000 ітерацій. Фіксований розмір популяції 1000.

Кожне рішення в популяції оцінюється загальною відстанню пройденого шляху. Алгоритм обирає кращі рішення для пулу розмноження у моєму випадку він буде містити 100 найкращих рішень.

І потім використовує рішення з "пулу розмноження"(BreedingPoolSize) для створення нового покоління рішень

* Більший пул розмноження дозволяє алгоритму досліджувати більше можливих рішень, що може призвести до пошуку кращих рішень, але може зайняти більше часу.
* Менший пул розмноження прискорює алгоритм, але може обмежити його здатність знаходити глобально оптимальне рішення.

Першим досліджувався параметр оператора схрещення (batch\_size) який визначає розмір сегменту при розбитті задач на менші частини.

* Збільшення batch\_size призводило до того, що оператор схрещування обмінювався більшими блоками шляху між батьківськими рішеннями. Це може призводило до більш різноманітних рішень у наступному поколінні, але також і зменшувало ймовірність знаходження оптимального рішення, тому що деякі корисні комбінації генів можуть бути розбиті.
* Зменшення batch\_size призводило до більш локального обміну генами, що допомогло зберегти корисні комбінації генів і збільшити ймовірність знаходження оптимального рішення

Другим досліджувався параметр що впливає на мутацію (mutation\_probability) він визначає ймовірність того, що рішення буде мутувати.

* Збільшення mutation\_probability призводило до більшої кількості змін у рішеннях, що допомагає дослідити нові області простору пошуку і уникнути потрапляння в локальні мінімуми.
* Зменшення mutation\_probability призводило до меншої кількості змін, що може допомогти зберегти корисні комбінації генів і збільшити ймовірність знаходження оптимального рішення

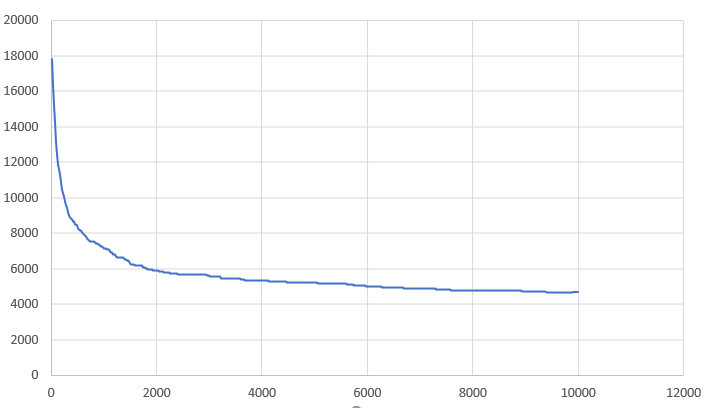


Рисунок 3.2 – Графік залежності результату від кількості ітерацій при найкращих параметрах

Результати тестування наведені в таблиці 3.1

|  |  |  |
| --- | --- | --- |
| Розмір сегментів | Вірогідність мутації | Результат |
| 30 | 0.1 | 5410 |
| 30 | 0.2 | 4988 |
| 30 | 0.3 | 4814 |
| 30 | 0.4 | 4988 |
| 60 | 0.3 | 4847 |
| 2 | 0.3 | 4917 |
| 4 | 0.3 | 4683 |
| 6 | 0.3 | 4898 |
| 10 | 0.3 | 4806 |
| 20 | 0.3 | 5079 |
|  |  |  |

Таблиця 3.1 – Результати тестування

Після додавання локального покращення алгоритм став працювати надто повільно, але 2-opt дійсно значно покращив результати, це видно на рисунку 3.3

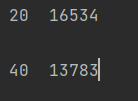


Рисунок 3.3 – декілька ітерацій алгоритму з локальним покращенням

Для порівняння розглянемо рисунок 3.4, де показані декілька ітерацій при найкращих параметрах але без локального покращення.

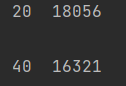


Рисунок 3.4 – декілька ітерацій алгоритму без локального покращення

Висновок

В рамках даної лабораторної роботи було досліджено вплив параметрів генетичного алгоритму на його ефективність при вирішенні задачі комівояжера. Зокрема, досліджувався вплив розміру сегменту для схрещування (batch\_size) та ймовірності мутації (mutation\_probability). Найкращий результат на 10000 ітераціях було досягнуто при наступних значеннях параметрів:

batch\_size = 4

mutation\_probability = 0.3

Розбиття шляху батька на 4 сегменти при схрещуванні дозволяє алгоритму досліджувати більш широкий спектр можливих рішень, не втрачаючи при цьому корисні комбінації генів. Ймовірність мутації 0.3 забезпечує достатній рівень випадковості для пошуку нових рішень, уникаючи при цьому надмірного руйнування корисних комбінацій генів.

Отримані результати можуть бути корисними для дослідників та розробників, які працюють над проблемами оптимізації.

Критерії оцінювання

При здачі лабораторної роботи до 24.12.2023 включно максимальний бал дорівнює – 5. Після 24.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* покроковий алгоритм – 10%;
* програмна реалізація алгоритму – 45%;
* робота з гіт – 20%;
* тестування алгоритму– 20%;
* висновок – 5%.

+1 додатковий бал можна отримати за виконання та захист роботи до 17.12.2023