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

«Проєктування алгоритмів»

«Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2»

Варіант 3

Виконав студент ІП-14 Гайдучек Максим Андрійович

(шифр, прізвище, ім'я, по батькові)

Перевірив Соколовський Владислав Володимирович

( прізвище, ім'я, по батькові)

Київ 2022

**Лабораторна робота №5**

**Тема роботи:** Проектування і аналіз алгоритмів для вирішення NP-складних задач.

**Мета роботи:** вивчити основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору

прийнятних параметрів алгоритму.

**Варіант:** 3

**Завдання**: Задача комівояжера (асиметрична мережа) + Генетичний алгоритм

**Хід роботи**

1. Покроковий алгоритм
2. Ініціалізувати початкову популяцію з випадковими генами, які представляють собою маршрути комівояжера
3. Генерувати нову популяцію, виділяючи відсоток найкращих осіб, а всі інші особи схрещувати та з деякою ймовірністю мутувати
4. Повторювати кроки b-c поки не буде досягнуто максимальної кількості ітерацій
5. Найкращу особу повернути як результат
6. Код алгоритму

package lab.algorithms;  
  
import lab.entities.Path;  
  
import java.util.\*;  
import java.util.stream.IntStream;  
  
public class GeneticAlgorithm {  
  
 private static final int *POPULATION\_SIZE* = 1000;  
 private static final int *BREEDING\_POOL\_SIZE* = *POPULATION\_SIZE* / 10;  
 private static final int *MAX\_ITERATIONS* = 100000;  
  
 public static void solve(int[][] matrix, double mutationProbability, int parents) {  
 Path[] generation = *createInitialGeneration*(matrix);  
 Random random = new Random();  
 for (int i = 0; i < *MAX\_ITERATIONS*; i++) {  
 Path[] breedingPool = Arrays.*stream*(generation)  
 .sorted(Comparator.*comparing*(Path::getCost))  
 .limit(*BREEDING\_POOL\_SIZE*)  
 .toArray(Path[]::new);  
 Path[] best = Arrays.*stream*(breedingPool)  
 .limit(*BREEDING\_POOL\_SIZE* / 5)  
 .toArray(Path[]::new);  
 double[] probabilities = *getProbabilities*(breedingPool);  
 Path[] newPopulation = new Path[matrix.length];  
  
 for (int j = 0; j < matrix.length; j++) {  
 if (j < *BREEDING\_POOL\_SIZE* / 5) {  
 newPopulation[j] = best[j];  
 continue;  
 }  
  
 Path first = breedingPool[*choosePath*(probabilities)];  
 Path second = breedingPool[*choosePath*(probabilities)];  
 newPopulation[j] = *cross*(matrix, parents, first, second);  
 if (random.nextDouble() < mutationProbability) {  
 *mutate*(newPopulation[j]);  
 }  
 }  
 generation = newPopulation;  
 System.*out*.printf("%d %d\n",  
 i, *getBestPath*(generation).getCost());  
 }  
 }  
  
 private static Path[] createInitialGeneration(int[][] matrix) {  
 Path[] generation = new Path[*POPULATION\_SIZE*];  
 for (int i = 0; i < *POPULATION\_SIZE*; i++) {  
 List<Integer> pathList = new ArrayList<>(IntStream.*range*(0, matrix.length).boxed().toList());  
 Collections.*shuffle*(pathList);  
 int[] path = pathList.stream().mapToInt(Integer::intValue).toArray();  
 generation[i] = new Path(path, *calcCost*(matrix, path));  
 }  
 return generation;  
 }  
  
 private static int calcCost(int[][] matrix, int[] path) {  
 int sum = 0;  
 for (int i = 0; i < path.length - 1; i++) {  
 sum += matrix[path[i]][path[i + 1]];  
 }  
 return sum + matrix[path[path.length - 1]][path[0]];  
 }  
  
 private static void mutate(Path path) {  
 Random random = new Random();  
 int first = random.nextInt(path.getPath().length - 1);  
 int second = random.nextInt(path.getPath().length - 1);  
 int temp = path.getPath()[first];  
 path.getPath()[first] = path.getPath()[second];  
 path.getPath()[second] = temp;  
 }  
  
 private static Path cross(int[][] matrix, int parents, Path first, Path second) {  
 int parentSize = matrix.length / parents;  
 int[] firstGenes = new int[matrix.length / 2];  
 for (int i = 0; i < matrix.length / 2; i++) {  
 firstGenes[i] = first.getPath()[parentSize \* 2 \* (i / parentSize) + i % parentSize];  
 }  
  
 int[] path = new int[matrix.length];  
 int currentSecond = 0;  
 for (int i = 0; i < matrix.length; i++) {  
 if ((i / parentSize) % 2 == 0) {  
 path[i] = first.getPath()[i];  
 continue;  
 }  
  
 while (Arrays.*stream*(firstGenes).boxed().toList().contains(second.getPath()[currentSecond])) {  
 currentSecond++;  
 }  
 path[i] = second.getPath()[currentSecond++];  
 }  
  
 return new Path(path, *calcCost*(matrix, path));  
 }  
  
 private static int choosePath(double[] probabilities) {  
 Random random = new Random();  
 double probability = random.nextDouble();  
 double sum = 0.0;  
 for (int i = 0; i < probabilities.length; i++) {  
 sum += probabilities[i];  
 if (sum > probability) {  
 return i;  
 }  
 }  
 return probabilities.length - 1;  
 }  
  
 private static double[] getProbabilities(Path[] paths) {  
 double[] values = new double[paths.length];  
 double sum = 0.0;  
 for (int i = 0; i < paths.length; i++) {  
 values[i] = 1.0 / paths[i].getCost();  
 sum += values[i];  
 }  
 for (int i = 0; i < values.length; i++) {  
 values[i] /= sum;  
 }  
  
 return values;  
 }  
  
 private static Path getBestPath(Path[] generation) {  
 return Arrays.*stream*(generation)  
 .min(Comparator.*comparing*(Path::getCost))  
 .orElse(null);  
 }  
}

1. Приклади роботи алгоритму

На рисунках 3.1 і 3.2 показані приклади роботи програми.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3.1 – Приклад роботи алгоритму

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3.2 – Приклад роботи алгоритму

1. Тестування алгоритму

У таблиці 4.1 наведено значення цільової функції зі збільшенням кількості ітерацій.

Таблиця 4.1 - значення цільової функції зі збільшенням кількості ітерацій

|  |  |
| --- | --- |
| Кількість ітерацій | Ціна найкращого шляху |
| 10 | 975 |
| 100 | 1017 |
| 1000 | 858 |
| 10000 | 918 |
| 100000 | 933 |
| 1000000 | 951 |
| 10000000 | 901 |
| 100000000 | 853 |

На рисунку 4.1 наведений графік, який показує якість отриманого розв'язку.

Рисунок 4.1 - Графік залежності розв'язку від числа ітерацій

**Висновок**: під час виконання даної лабораторної роботи було розв’язано задачу комівояжера та розроблено генетичний алгоритм. Було протестовано код та досліджено залежність розв'язку від числа ітерацій. Оскільки алгоритм має елемент випадковості, то чим більше ітерацій проведемо, тим більша ймовірність на знаходження правильного результату.