**Мета роботи**: Ознайомитися з основними теоретичними відомостями, вивчити еволюційні оператори схрещування та мутації, що використовуються при розв’язуванні задач комбінаторної оптимізації.

**Завдання:** Розробити за допомогою довільної мови програмування програмне забезпечення для вирішення задачі комівояжера.

В - 5

Використовуючи турнірну селекцію.

**Результати виконання програми**

Програма запускалася 6 раз з різною кількістю міст і популяцією. Для порівняння було вибрано кількість міст 10, 20, а популяцію 20, 100,200.

Координати для 10 міст

[12,2;5,3;15,15;17,3;8,16;4,13;6,14;6,19;5,14;2,9;]

Координати для 20 міст

[7,20;13,8;4,13;5,18;12,15;8,15;6,7;13,17;1,13;11,6;16,3;20,12;16,12;6,19;13,5;11,10;13,12;7,3;16,3;19,2;]

Результати для 20 міст(популяція 200):

Final distance: 71

Time: 41240мс

Solution:

|16, 12|20, 12|19, 2|16, 3|16, 3|13, 5|11, 6|7, 3|6, 7|1, 13|4, 13|8, 15|5, 18|6, 19|7, 20|13, 17|12, 15|13, 12|11, 10|13, 8|

В програмі використовувалося двох точкове впорядковуючи схрещування , мутація класичне інвертування і турнірна селекція.

Таблиця порівняння кількості міст і популяції

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Кількість міст | 10 | | | 20 | | |
| Популяція | 20 | 100 | 200 | 20 | 100 | 200 |
| Час виконання,c | 2.27 | 11.33 | 22.6 | 4.06 | 19.99 | 41.08 |
| Довжина шляху | 50 | 50 | 50 | 75 | 71 | 71 |
| Довжина шляху в 3 лаб. | 53.6 | 53.6 | 53.6 | 91.38 | 87.22 | 80.76 |

Код програми

**City.class**

**public** **class** City {

**int** x;

**int** y;

// Constructs a randomly placed city

**public** City(){

**this**.x = (**int**)(Math.*random*()\*200);

**this**.y = (**int**)(Math.*random*()\*200);

}

// Constructs a city at chosen x, y location

**public** City(**int** x, **int** y){

**this**.x = x;

**this**.y = y;

}

// Gets city's x coordinate

**public** **int** getX(){

**return** **this**.x;

}

// Gets city's y coordinate

**public** **int** getY(){

**return** **this**.y;

}

// Gets the distance to given city

**public** **double** distanceTo(City city){

**int** xDistance = Math.*abs*(getX() - city.getX());

**int** yDistance = Math.*abs*(getY() - city.getY());

**double** distance = Math.*sqrt*( (xDistance\*xDistance) + (yDistance\*yDistance) );

**return** distance;

}

**public** String toString(){

**return** getX()+", "+getY();

}

}

**GA.class**

**package** tsp;

**public** **class** GA {

/\* GA parameters \*/

**private** **static** **final** **double** *mutationRate* = 0.015;

**private** **static** **final** **int** *tournamentSize* = 5;

**private** **static** **final** **boolean** *elitism* = **true**;

// Розвивається населенням більше одного покоління

**public** **static** Population evolvePopulation(Population pop) {

Population newPopulation = **new** Population(pop.populationSize(), **false**);

// Тримати кращих людей якщо елітарнсть включена

**int** elitismOffset = 0;

**if** (*elitism*) {

newPopulation.saveTour(0, pop.getFittest());

elitismOffset = 1;

}

// Crossover population

// Цикл розміром нового населення і створення осіб поточного населення

**for** (**int** i = elitismOffset; i < newPopulation.populationSize(); i++) {

// Select parents

Tour parent1 = *tournamentSelection*(pop);

Tour parent2 = *tournamentSelection*(pop);

// Crossover parents

Tour child = *crossover*(parent1, parent2);

// Add child to new population

newPopulation.saveTour(i, child);

}

// Mutate the new population a bit to add some new genetic material

**for** (**int** i = elitismOffset; i < newPopulation.populationSize(); i++) {

*mutate*(newPopulation.getTour(i));

}

**return** newPopulation;

}

//двохточкове впорядковуюче

**public** **static** Tour crossover(Tour parent1, Tour parent2) {

// Create new child tour

Tour child = **new** Tour();

**int** p1 = (**int**) (Math.*random*() \* parent1.tourSize());

**int** p2 = (**int**) (Math.*random*() \* parent1.tourSize());

**for**(**int** i = 0; i < child.tourSize(); i++)

{

**if**(i>=p1 && i<=p2) child.setCity(i, **null**);

**else** child.setCity(i, parent1.getCity(i));

}

**int** n = 0;

**for** ( **int** j = 0; j < parent2.tourSize(); j++)

{

**boolean** t = **false**;

**for** ( **int** k = 0; k < child.tourSize(); k++)

{

**if**(parent2.getCity(j) == child.getCity(k)) {

t = **true**;

**break**;

}

}

**if** (t== **false**){

child.setCity(p1+n, parent2.getCity(j));

n = n+1;

}

}

**return** child;

}

//класичне інвертування

**private** **static** **void** mutate(Tour tour) {

**int** tourPos1 = (**int**) (tour.tourSize() \* Math.*random*());

**int** tourPos2 = (**int**) (tour.tourSize() \* Math.*random*());

**if** (tourPos2 < tourPos1) {

**int** q = tourPos1;

tourPos1 = tourPos2;

tourPos2 = q;

}

**for**(**int** i = 0; i <= (tourPos2 - tourPos1 +1)/2 ; i++)

{

City q = tour.getCity(tourPos1+i);

tour.setCity(tourPos1+i, tour.getCity(tourPos2-i));

tour.setCity(tourPos2-i, q);

}

}

// Selects candidate tour for crossover

**private** **static** Tour tournamentSelection(Population pop) {

// Create a tournament population

Population tournament = **new** Population(*tournamentSize*, **false**);

// For each place in the tournament get a random candidate tour and

// add it

**for** (**int** i = 0; i < *tournamentSize*; i++) {

**int** randomId = (**int**) (Math.*random*() \* pop.populationSize());

tournament.saveTour(i, pop.getTour(randomId));

}

// Get the fittest tour

Tour fittest = tournament.getFittest();

**return** fittest;

}

}

**class Population**

**package** tsp;

**public** **class** Population {

// Holds population of tours

Tour[] tours;

// Construct a population

**public** Population(**int** populationSize, **boolean** initialise) {

tours = **new** Tour[populationSize];

**if** (initialise) {

// Створення індивідумів

**for** (**int** i = 0; i < populationSize(); i++) {

Tour newTour = **new** Tour();

newTour.generateIndividual();

saveTour(i, newTour);

}

}

}

// Saves a tour

**public** **void** saveTour(**int** index, Tour tour) {

tours[index] = tour;

}

// Gets a tour from population

**public** Tour getTour(**int** index) {

**return** tours[index];

}

// Gets the best tour in the population

**public** Tour getFittest() {

Tour fittest = tours[0];

// Loop through individuals to find fittest

**for** (**int** i = 1; i < populationSize(); i++) {

**if** (fittest.getFitness() <= getTour(i).getFitness()) {

fittest = getTour(i);

}

}

**return** fittest;

}

// Gets population size

**public** **int** populationSize() {

**return** tours.length;

}

}

**TourManager.class**

**import** java.util.ArrayList;

**public** **class** TourManager {

// Holds our cities

**private** **static** ArrayList *destinationCities* = **new** ArrayList<City>();

// Adds a destination city

**public** **static** **void** addCity(City city) {

*destinationCities*.add(city);

}

// Get a city

**public** **static** City getCity(**int** index){

**return** (City)*destinationCities*.get(index);

}

// Get the number of destination cities

**public** **static** **int** numberOfCities(){

**return** *destinationCities*.size();

}

**}**

**Class TSP\_GA**

**package** tsp;

**import** java.util.Date;

**public** **class** TSP\_GA {

**public** **static** **void** main(String[] args) {

// Create and add our cities

City city = **new** City(7, 20);

TourManager.*addCity*(city);

City city2 = **new** City(13, 8);

TourManager.*addCity*(city2);

City city3 = **new** City(4, 13);

TourManager.*addCity*(city3);

City city4 = **new** City(5, 18);

TourManager.*addCity*(city4);

City city5 = **new** City(12, 15);

TourManager.*addCity*(city5);

City city6 = **new** City(8, 15);

TourManager.*addCity*(city6);

City city7 = **new** City(6, 7);

TourManager.*addCity*(city7);

City city8 = **new** City(13, 17);

TourManager.*addCity*(city8);

City city9 = **new** City(1, 13);

TourManager.*addCity*(city9);

City city10 = **new** City(11, 6);

TourManager.*addCity*(city10);

City city11 = **new** City(16, 3);

TourManager.*addCity*(city11);

City city12 = **new** City(20, 12);

TourManager.*addCity*(city12);

City city13 = **new** City(16, 12);

TourManager.*addCity*(city13);

City city14 = **new** City(6, 19);

TourManager.*addCity*(city14);

City city15 = **new** City(13, 5);

TourManager.*addCity*(city15);

City city16 = **new** City(11, 10);

TourManager.*addCity*(city16);

City city17 = **new** City(13, 12);

TourManager.*addCity*(city17);

City city18 = **new** City(7, 3);

TourManager.*addCity*(city18);

City city19 = **new** City(16, 3);

TourManager.*addCity*(city19);

City city20 = **new** City(19, 2);

TourManager.*addCity*(city20);

// Initialize population

Population pop = **new** Population(200, **true**);

System.*out*.println("Initial distance: " + pop.getFittest().getDistance());

Date currentTimeBefore = **new** Date();

**long** timeBefore = currentTimeBefore.getTime();

//System.out.println("Time: " + timeBefore);

// Evolve population for 100 generations

pop = GA.*evolvePopulation*(pop);

**for** (**int** i = 0; i < 100000; i++) {

pop = GA.*evolvePopulation*(pop);

}

Date currentTimeAfter = **new** Date();

**long** timeAfter= currentTimeAfter.getTime();;

//System.out.println("Time: " + timeAfter);

**long** time = timeAfter-timeBefore;

// Print final results

System.*out*.println("Finished");

System.*out*.println("Final distance: " + pop.getFittest().getDistance());

System.*out*.println("Time: " + time + "мс");

System.*out*.println("Solution:");

System.*out*.println(pop.getFittest());

}

}

**Висновки:** виконавши лабораторну роботу я вивчив еволюційні оператори схрещування та мутації, що використовуються при розв’язуванні задач комбінаторної оптимізації. Реалізував за допомогою програмної мови Java програмне забезпечення для вирішення задачі комівояжера, в результаті програма дає результати близькі до оптимальних при кількості міст до 20, при більшій кількості шлях є неоптимальним. Також був виявлений зв'язок що при збільшенні кількості популяції довжина шляху зменшується, але час роботи програми збільшується