Raport tema nr. 3

1. **Descrierea problemei**

Data fiind o problema NP-dificila in optimizarea combinatoriala, scrieti un algoritm euristic si un algoritm genetic care sa rezolve problema. Problema aleasa de noi este problema Comisului Voiajor (TSP).

1. **Algoritmul utilizat**

Pentru rezolvarea acestei probleme, vom implementa atat un **algoritm euristic**, cat si unul **genetic**. Acesta din urma are la baza o strategie evolutiva, plecand de la o populatie initiala de cromozomi care, pe masura ce avanseaza, **evolueaza** cu ajutorul unor **mutatii**. Doar cei mai puternici indivizi (cromozomi) vor supravietui timp de mai multe **generatii**.

**Terminologie**

O **populatie** este formata dintr-o multitudine de cromozomi. O **generatie** este reprezentata de o populatie la un anumit moment.

In contextul algoritmului, un **cromozom** este un drum valid al comisului voiajor, adica incepe si se termina in acelasi oras. Un cromozom este format din mai multe **gene,** iar o gena reprezinta un oras in programul nostru.

Pe parcursul programului, aceasta populatie evolueaza si se vor obtine cromozomi din ce in ce mai buni, in sensul ca valoarea costului drumurilor, reprezentati de cromozomi, va scadea pana se va obtine un minim global.

In primul rand, sunt **selectati** cromozomii ce vor forma generatia viitoare. Acest lucru se realizeaza printr-o selectie de tipul „Roata Norocului”. Apoi, cromozomii sunt supusi **incrucisarii** si **mutatiei.**

**Pseudocod:**

Genereaza o populatie initiala, random

Cat timp populatia evolueaza

1. Evalueaza populatia curenta
2. Selecteaza cromozomii pentru urmatoarea generatie (Roata Norocului)
3. Incruciseaza cromozomii
4. Supune cromozomii mutatiei

Returneaza cea mai buna solutie gasita.

**Detalii de implementare**

Populatia va fi reprezentata in memorie printr-o matrice care retine mai multi cromozomi, fiecare cromozom fiind o lista de orase (gene).

int chromosomes[MAX\_CHROMOSOMES][MAX\_CITIES];

Generarea aleatoare a populatiei initiale se va face pe baza unui liste de orase si distante data, interschimband orasele intre ele de swapNr ori:

swapNr = 100;

for (i = 0; i < swapNr; ++i) {

swap(chromosome[rand() % City::nr], chromosome[rand() % City::nr]);

}

Fiecare cromozom este evaluat in functie de **fitness-ul** lui (cat de bun este). In cazul problemei noastre, valoarea fitness-ului pentru orasul x va fi 1/y, unde y reprezinta costul distantei cromozomului x.

rez.lg = CalcDistDrum(chromosome);

rez.fitness = 1 / rez.lg;

**Selectia** se realizeaza prin Roata Norocului. Pentru aceasta, calculam fitness-ul fiecarui cromozom si suma totala a acestor fitness-uri, S. Calculam apoi probabilitatile individuale si cumulate astfel:

for (int i = 0; i < popSize; ++i)

prob[i] = chromosomeFitness[i] / sumFitness;

probCumulated[0] = 0;

for (int i = 1; i < popSize; ++i)

probCumulated[i] = probCumulated[i - 1] + prob[i - 1];

Evident, probCumulated[0] = 0 si probCumulated[n+1] = 1, unde n este numarul total de cromozomi, adica marimea populatiei.

Apoi, timp de n iteratii, generam un numar random intre (0, 1). Va intra in generatia urmatoare acel cromozom j pentru care probCumulated[j] <= randomNumber <= probCumulated[j+1].

for (int i = 0; i < popSize; ++i) {

randomNr = RandomDouble(0, 1);

for (int j = 0; j < popSize; ++j)

if (probCumulated[j] < randomNr && randomNr <= probCumulated[j + 1])

chosen[i] = j;

}

Construim **urmatoarea generatie:**

for (int i = 0; i < popSize; ++i)

Copy(auxChromosomes[i], chromosomes[i], City::nr);

int i, aux = popSize;

popSize = 0;

for (i = 0; i < aux; ++i)

Copy(chromosomes[popSize++], auxChromosomes[chosen[i]], City::nr);

Aplicam operatorii de mutatie asupra noii populatii. **Incrucisarea** are sansa de a fi aplicata fiecarui cromozom in functie de o probabilitate pe care o stabilim noi, P\_CROSS:

for (int i = 0; i < popSize; ++i) {

randomNr = RandomDouble(0, 1);

if (randomNr < P\_CROSS) {

chosenForCrossing.push\_back(i);

chromosomeWillBeCrossed[i] = true;

}

}

Metoda de incrucisare aleasa este PMX (Partially-Mapped Crossover), care functioneaza in felul urmator: se alege mai intai un punct de impartire al cromozomilor. Prima bucata a cromozomului 1 va fi copiata, gena cu gena, in prima bucata a cromozomului 2. Pentru a pastra validitatea cromozomului, orasele din cromozomul 2 nu sunt doar suprascrise. Orasul de pe pozitia p din cromozomul 2 se va interschimba in cadrul sau cu orasul care se gaseste pe aceeasi pozitie in cromozomul 1. Procedeul se va face si vice-versa, obtinandu-se astfel doi noi cromozomi:

split = rand() % City::nr;

for (int i = 0; i <= City::nr; ++i) {

newChromosome1[i] = chromosomes[pos1][i];

newChromosome2[i] = chromosomes[pos2][i];

}

for (int i = 0; i < split; ++i) {

i1 = i2 = 0;

while (newChromosome1[i2] != chromosomes[pos2][i] && i2 < City::nr)

++i2;

newChromosome1[i2] = newChromosome1[i];

newChromosome1[i] = chromosomes[pos2][i];

while (newChromosome2[i1] != chromosomes[pos1][i] && i1 < City::nr)

++i1;

newChromosome2[i1] = newChromosome2[i];

newChromosome2[i] = chromosomes[pos1][i];

}

**Mutatia** se aplica fiecarui cromozom, acesta avand probabilitatea P\_MUTATION de a fi alterat.

Am implementat doua mutatii. Una dintre ele doar inverseaza doua gene in cromozomul respectiv. A doua varianta de mutatie contine algoritm Greedy. Se selecteaza in mod aleatoriu 2 gene (orase) ale cromozomului si se oglindeste toata secventa cuprinsa intre cele 2 gene selectate. Cromozomul ramane cu aceasta schimbare doar daca mutatia e dus la un drum cu o lungime mai buna.

Pentru a ridica sansele gasirii drumurilor mai bune, vom incerca sa gasim un drum mai bun de MAX\_MUTATION\_SWAPS ori. Daca dupa acest numar de iteratii nu am gasit o mutatie a cromozomului care sa imbunatateasca drumul, atunci cromozomul ramane neschimbat. P\_MUTATION este adapdat in functie de varianta de mutatie folosita.

Un „trick” pe care l-am facut aici pentru a obtine rezultate mai bune a fost sa aplic mutatia acelor cromozomi care nu intra in incrucisare.

void MutateChromosomes(int chromosomes[][MAX\_CITIES], int popSize) {

double randomNr;

for (int i = 0; i < popSize; ++i) {

if (chromosomeWillBeCrossed[i]) continue;

randomNr = RandomDouble(0, 1);

if (randomNr <= P\_MUTATION)

swap(chromosomes[i][rand() % City::nr], chromosomes[i][rand() % City::nr]);

}

}

void MutateChromosomesv2(int chromosomes[][MAX\_CITIES], int popSize) {

double randomNr, lgInitial;

int randomPos1, randomPos2;

for (int i = 0; i < popSize; ++i) {

if (chromosomeWillBeCrossed[i]) continue;

randomNr = RandomDouble(0, 1);

if (randomNr <= P\_MUTATION) {

lgInitial = CalcDistDrum(chromosomes[i]);

for (int j = 0; j < MAX\_MUTATION\_SWAPS/5; ++j) {

randomPos1 = rand() % City::nr;

randomPos2 = rand() % City::nr;

if (randomPos2 < randomPos1)

swap(randomPos1, randomPos2);

int st, dr;

for (st = randomPos1, dr = randomPos2; st < dr; ++st, --dr)

swap(chromosomes[i][st], chromosomes[i][dr]);

if (CalcDistDrum(chromosomes[i]) < lgInitial)

continue;

for (st = randomPos1, dr = randomPos2; st < dr; ++st, --dr)

swap(chromosomes[i][st], chromosomes[i][dr]);

}

}

}

}

**Algoritmul euristic** ales de noi foloseste o abordare Greedy pentru a obtine drumul de cost minim. Practic, plecam din fiecare oras, alegand de fiecare data cel mai apropiat oras fata de cel in care ne aflam la un moment dat prin care nu am trecut, iar cand terminam drumul curent comparam cu cel mai bun gasit pana in acel moment:

for (int startVf = 0; startVf < City::nr; ++startVf) {

memset(haveCity, 0, sizeof(haveCity));

drum[0] = startVf;

haveCity[drum[0]] = true;

for (lgDrum = 1; lgDrum < City::nr; ++lgDrum) {

bestDist = INF;

for (int i = 0; i < City::nr; ++i)

if (!haveCity[i] && dist[drum[lgDrum - 1]][i] < bestDist) {

bestDist = dist[drum[lgDrum - 1]][i];

bestNeighbour = i;

}

drum[lgDrum] = bestNeighbour;

haveCity[bestNeighbour] = true;

}

**3. Rezultate experimentale**

Rezultatele au fost obtinute cu o populatie initiala de 2000 de cromozomi, P\_CROSS = 0.025 si P\_MUTATION = 0.33.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritm testat | Date de test | Numar de orase | Prima rulare | A doua rulare |
| Euristic | Berlin | 52 | 8182 |  |
| EIL | 101 | 736 |  |
| Djibouti | 38 | 6770 |  |
| Genetic | Berlin | 52 | 7925 | 7789 |
| EIL | 101 | 704 | 706 |
| Djibouti | 38 | 6659 | 6659 |

Un alt lucru crucial este cum definim „Cat timp populatia evolueaza”. Asta este ceea ce permite ca in viitor sa obtinem rezultate mai bune. Cu cat lasam populatia sa evolueze mai mult, cu atata o sa obtinem rezultate mai bune. Desigur, la schimbul unui timp de executie mai mare. Am definit aceasta evolutie astfel:

bool PopulationIsEvoluating(int lastBestGeneration, int currentGeneration) {

if (currentGeneration - lastBestGeneration >= 1000)

return false;

return true;

}

Cat timp am o evolutie in ultimele 1000 de generatii, continui algoritmul.