Algoritmi Genetici - Raport Tema 3

Leonard Olariu - grupa A5

January 3, 2019

Abstract

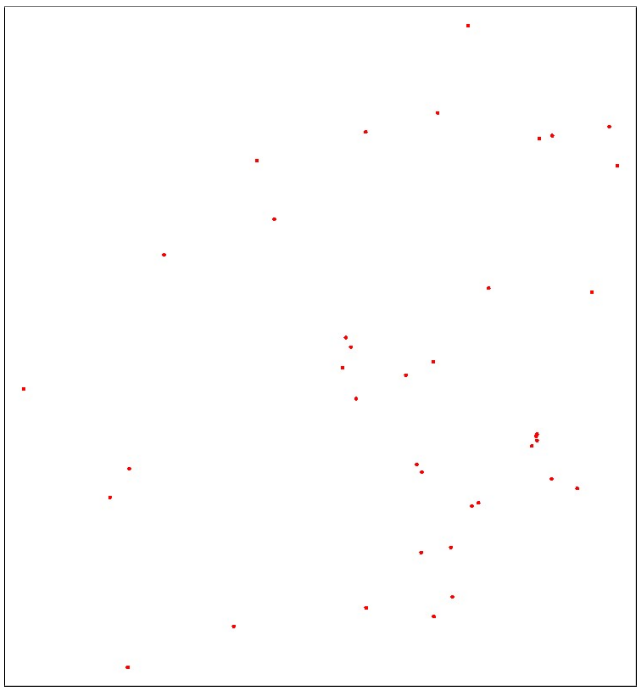
Acest raport urmareste prezentarea si testarea unui algoritm genetic ce doreste cercetarea spatiului de solutii si aproximarea minimului global pentru instante ale problemei comis-voiajorului.

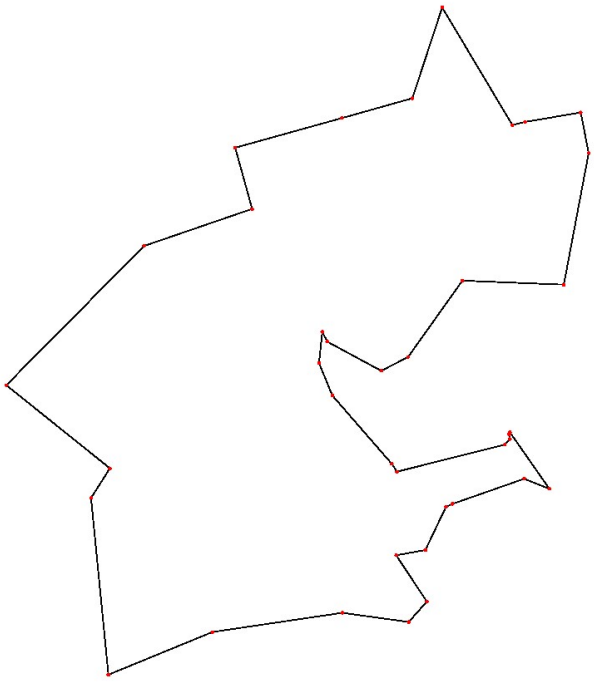
1 Context

Travelling Salesman Problem (TSP) este o problema NP-completa, arhicunoscuta in literatura de specialitate. Din moment ce algoritmul determinist ruleaza in timp factorial, rezolvarea instantelor devine nepractica de la un punct incolo. Spre exemplu, 10! este 3628800, dar 20! este gigantic, 2432902008176640000.

Lipsa puterii de procesare a unui astfel de volum de date a creat nevoia de algoritmi alternativi care pot aproxima rezultatul in timp polinomial. In sectiunile urmatoare voi descrie si analiza comportarea unui algoritm genetic.

Datele de test pot fi descarcate la urmatorul link: http://www.math.uwaterloo.ca/tsp/world/countries.html. Input-urile sunt de forma unei liste de locatii, identificate prin coordonate carteziene (x, y) si vin impreuna cu reprezentari grafice (atat pentru input, cat si pentru solutia optima) de tipul:





2 Descriere Algoritm

Algoritmul propus urmareste schema clasica a unui algoritm genetic, cu mentiunea ca reprezentarea solutiilor candidat, precum si operatorii specifici (mutatie/ incrucisare) trebuie adaptati problemei suport.

```
void geneticAlgorithm () {
    initializePopulation();
    evaluatePopulation();

    while (!termination-condition) {
        selectPopulation(); //select P(i) from P(i-1)

        //alter P(i)
        crossOverPopulation();
        mutatePopulation();

        evaluatePopulation();
    }
}
```

2.1 Reprezentarea Solutiilor

Vectori de aceeasi dimensiune cu instanta. Acestia vor reprezenta permutari a locatiilor.

```
struct population {
    int candidateSol[popSize][citiesNum];
    double candidateFitness[popSize], popFitness, wheelStart[popSize];
}; population *currPop = new population();
```

2.2 Initializarea Populatiei

Vom asocia fiecarui numar $0 : n-1$ o valoare random. Sortand aceste numere (corespondente locatiilor) dupa valoarea random asociata obtinem o permutare random. Repetam acest proces pentru fiecare cromozom al populatiei.

```
struct cityRank {
    int city;
    double randomNum;
} cityList[citiesNum];

int compare (cityRank a, cityRank b)
{
    if (a.randomNum < b.randomNum) return 1;
    return 0;
}

void initializePopulation() {
    for (int i = 0; i < popSize; ++i) {
        for (int j = 0; j < citiesNum; ++ j) {
            cityList[j].city = j;
            cityList[j].randomNum = rand() / (RAND_MAX + 1.);
        }

        sort(cityList, cityList + citiesNum, compare);

        for (int j = 0; j < citiesNum; ++j) currPop->candidateSol[i][j] = cityList[j].city;
    }
}
```

2.3 Evaluarea/ Selectia Populatiei

Analog temei 2, cu precizarea ca trebuie sa asiguram o functie fitness care sa fie pozitiva si cu atat mai mare cu cat individul este mai bun. Din moment ce dorim sa minimizam lungimea circuitului, putem considera

$$fitness(i) = \frac{1}{length(i)}$$

```
//compute the fitness of the individual i
for (int j = 0; j < citiesNum-1; ++j)
    currPop->candidateFitness[i] +=
        dist[currPop->candidateSol[i][j]][currPop->candidateSol[i][j+1]];
currPop->candidateFitness[i] +=
    dist[currPop->candidateSol[i][citiesNum-1]][currPop->candidateSol[i][0]];

//assure that shorter length determines greater fitness
currPop->candidateFitness[i] = 1 / currPop->candidateFitness[i];
```

Observatie. In funtie de maniera in care selectam noua generatie, algoritmul genetic poate urmari multiple scheme de implementare:

1. Steady-State Approach with Roulette Wheel Selection
2. Steady-State Approach with Tournament Selection
3. Generational Approach with Roulette Wheel Selection
4. Generational Approach with Tournament Selection

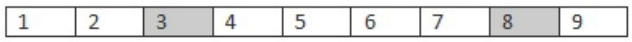
Trebuie mentionat faptul ca o implementare "Steady-State" proceseaza doar cei mai buni N cromozomi (parinti/ copii), iar o abordare generationala are in vedere doar copiii obtinuti in urma incrucisarii. In cadrul rezultatelor experimentale vom trata doar a 3-a schema de implementare.

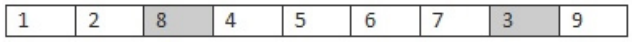
2.4 Alterarea Indivizilor

In literatura de specialitate sunt prezentate multiple variatii ale operatorilor specifici alterarii candidatilor pentru problema suport. Fiecare metoda are in vedere producerea de solutii valide (alte permutari), cu garantia ca nu vor adauga sau pierde vreodata locatii din traseul curent. Nu putem spune ca o variatie este mai buna decat cealalta, deoarece ele se comporta diferit in functie de datele de intrare. In continuare vom prezenta o singura varianta pentru fiecare operator.

2.4.1 Mutatia

Interschimbare a 2 locatii.

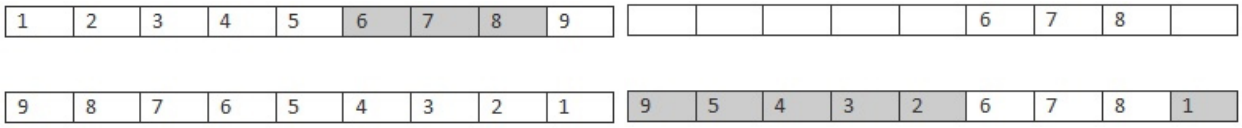





```
void mutatePopulation() {
    for (int i = 0; i < popSize; ++i)
        if (rand() / (RAND_MAX + 1.) < mutationProbability) {
            int cut1 = rand() % (citiesNum), cut2;
            do cut2 = rand() % (citiesNum); while (cut1 == cut2);

            swap(currPop->candidateSol[i][cut1], currPop->candidateSol[i][cut2]);
        }
}
```

2.4.2 Incrucisarea

Incrucisarea ordonata presupune alegerea la intamplare a 2 locatii. Subsetul determinat de acestea este pastrat de copii. Fiecare copil va mosteni valorile lipsa de la celalalt parinte, in ordinea in care acestea sunt intalnite.



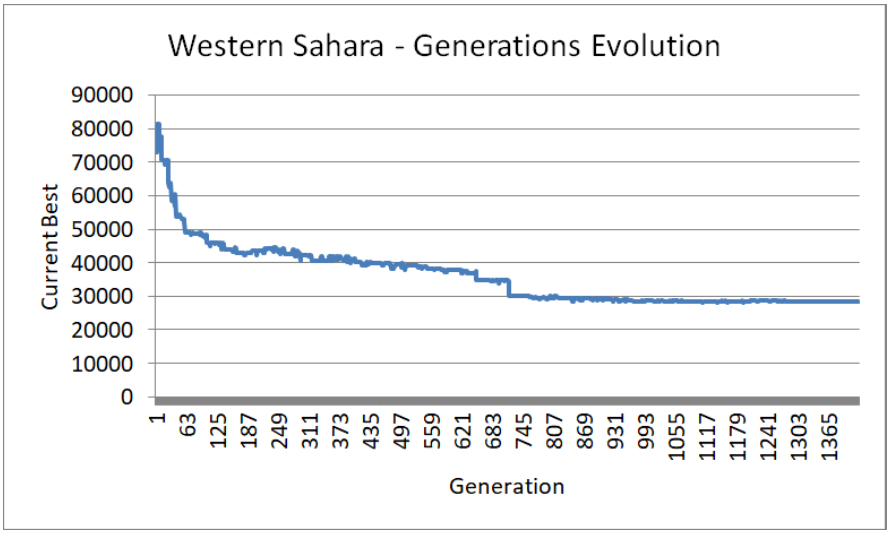
Parents Child

```
void crossOverPopulation() {
    int parent[2], parentCount = 1;
    for (int i = 0; i < popSize; ++i)
        if (rand() / (RAND_MAX + 1.) < crossOverProbability) {
            parentCount = (parentCount+1) % 2;
            parent[parentCount] = i;

            if (parentCount) {
                int cut1 = rand() % (citiesNum), cut2;
                do cut2 = rand() % (citiesNum); while (cut1 == cut2);

                if (cut1 > cut2) swap(cut1, cut2);

                int parentCopy[citiesNum];
                memcpy(parentCopy, currPop->candidateSol[parent[0]], citiesNum * sizeof(int));

                bool used[citiesNum] = {};
                for (int j = cut1; j <= cut2; ++j) used[currPop->candidateSol[parent[0]][j]] = 1;

                int currPos = 0;
                for (int j = 0; j < citiesNum; ++j) {
                    if (currPos == cut1) currPos = cut2+1;
                    if (!used[currPop->candidateSol[parent[1]][j]]) {
                        currPop->candidateSol[parent[0]][currPos++] =
                            currPop->candidateSol[parent[1]][j];
                        used[currPop->candidateSol[parent[1]][j]] = 1;
                    }
                }

                used[citiesNum] = {};
                for (int j = cut1; j <= cut2; ++j) used[currPop->candidateSol[parent[1]][j]] = 1;

                currPos = 0;
                for (int j = 0; j < citiesNum; ++j) {
                    if (currPos == cut1) currPos = cut2+1;
                    if (!used[parentCopy[j]]) {
                        currPop->candidateSol[parent[1]][currPos++] = parentCopy[j];
                        used[parentCopy[j]] = 1;
                    }
                }
            }
        }
}
```

3 Rezultate Experimentale

	popSize	best	officialBest	worst	mean	stDev	time (s)
Western Sahara (29 cities)	1000	27601.17377	27603	39788.99215	31357.72788	3015.89973	19.670
Djibouti (38 cities)	500	6720.05481	6656	10676.29937	8640.40152	983.06817	21.529
Qatar (194 cities)	1000	63589.19108	9352	76822.22269	69660.20428	3678.00181	50.833
	5000	54372.85896	9352	79322.65190	63429.19264	5218.29104	257.162

Observatie. Aceste rezultate au fost obtinute pentru 30 de rulari a programului, cu $mutationProbability = 0.1$ si $crossOverProbability = 0.15$.



4 Concluzii

1. Algoritmul genetic propus reduce semnificativ timpul de executie, putand aborda instante mult mai mari decat un algoritm determinist. Cu toate acestea, precizia scade considerabil odata cu dimensiunea input-ului.

2. Nici complexitatea unui algoritm de tip Hill Climbing - Best Improvement nu se apropie de cea a algoritmului genetic, deoarece stabilirea celei mai bune vecinatati este realizata in complexitate $O(n^2)$ (numarul tuturor mutatiilor posibile).

5 Bibliografie

1. https://profs.info.uaic.ro/ marta/ga/L3/
2. http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5
3. https://medium.com/@becmjo/genetic-algorithms-and-the-travelling-salesman-problem-d10d1daf96a1