Trăistaru Vlad-Viorel
Moise Andrei
Grupa 442Aa

Artificial Bee Colony (ABC)

Cuprins

Descrierea algoritmului

Algoritmul începe prin declararea constantelor pi și Euler, precum și a structurii albinei care conține un vector de tip double pentru poziții și variabilă de tip double pentru cost.

```
/* constant values */
constexpr auto M_PI = 3.14159265358979323846;
constexpr auto M_E = 2.7182818284590452354;

using namespace std;

/* bee structure */
struct Bee {
    vector<double> pos;
    double cost = 0.0f;
};
```

Se continuă cu definirea clasei în interiorul căreia se găsesc câmpurile și metodele ce sunt folosite în algoritm. Astfel avem: un câmp privat pentru funcția care este analizată, două câmpuri publice pentru limita inferioară și limita superioară a funcției analizate, un constructor folosit pentru inițializări în care se află subprogramul de setare a limitelor pentru fiecare din cele cinci funcții [1](pg. 464-465) și o metodă care returnează un vector de poziții pentru aceleași funcții.

```
/* class that holds the optimizable functions ready for use */
class OptimizableFunction {
private:
    /* variable that determines the used function */
    int m_choice;
public:
    /* bounds for the used function */
    double m_lowerBound;
    double m_upperBound;

    /* constructor */
    OptimizableFunction(int choice) {
        m_choice = choice;
        setBounds();
    }
```

Urmează două funcții: una care returnează suma cumulativă a unui vector de numere și una care returnează indexul unei potențiale soluții bune [2]. În primul caz avem de a face cu o funcție care trebuie să returneze un vector de tip double, și care ia ca argument un parametru de același tip. În cel de al doilea caz vorbim despre o metodă care trebuie să returneze o variabilă de tip int, dar care ia ca argument un vector de tip double.

```
/* returns the cumulative sum of a vector of numbers */
vector<double> cumSum(vector<double> P) {
    double partialSum = 0;
    for (int i = 0; i < P.size(); i++) {
        partialSum += P[i];
        P[i] = partialSum;
    }
    return P;
}

/* returns index of potentially useful solution */
int fitnessProportionateSelection(vector<double> P) {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> randDist(0, 1);
    double randNumber = randDist(gen);
    vector<double> cumVector = cumSum(P);
    for (int i = 0; i < cumVector.size(); i++) {
        if (randNumber <= cumVector[i]) {
            return i;
        }
    }
    return NULL;
}
```

În cadrul funcției main: se definește infinitatea, modelul generatorului de numere aleatoare și i se dă ca seed timpul curent. Se parcurg toate funcțiile posibile, precum și dimensiunile distanței în care avem: numărul maxim de experimente și iterații, un vector în care se reține cel mai bun cost pentru fiecare experiment, costul mediu și deviația standard pentru toate experimentele, se creează un obiect de tip funcție care ia ca parametru funcția care se dorește să fie analizată, se iau valorile pentru distanțe, se precizează mărimea populației de albine, numărul de albine “angajate”(employed), albine care privesc(onlookers) și albine “cercetași”(scouts) [1] (pg. 461), limita care determină cât de mult se merită să caute o nouă sursă de hrană, parametrul de accelerare folosit pentru găsirea mai rapidă a sursei de hrană, se afișează pentru fiecare funcție numărul acesteia, numărul de experimente, numărul maxim de iterații și dimensiunea distanței.

```
/* create a function object to be tested; parameter is used for choosing a certain function */
OptimizableFunction optimizableFunction(functionNumber);
/* get the bound values for the function */
int distanceMin = optimizableFunction.m_lowerBound;
int distanceMax = optimizableFunction.m_upperBound;

/* colony size of bees (or population size) */
int beesPopulation = 125;
/* employed bees (sending them onto the food sources to measure their nectar amounts / fitness
  value) */
int beesEmployed = int(50.0f / 100.0f * beesPopulation);
/* onlooker bees (select the food sources using the nectar information / fitness value) */
int beesOnlooker = beesEmployed;
/* scout bees (sent to the selected food sources) */
int beesScout = 1;
/* limit used for determining the worth of finding a food source */
int limit = int(round(0.6 * distanceSize * beesPopulation));
/* acceleration parameter used for finding potential food sources faster */
int accel = 1;
```

Se iterează prin toate experimentele unde: se definește vectorul pentru întreaga populație de albine, se inițializează vectorul la început cu date goale, se inițializează cea mai bună soluție cu cel mai mare cost(infinit), precum și distanțele pentru vectorul de albine cu valori aleatoare și se calculează costurile. Se declară un vector pentru numărarea albinelor abandonate și un vector pentru păstrarea în memorie a celui mai bun cost pentru fiecare iterație.

```
/* vector for the entire population of bees */
vector<Bee> beesVector;
/* initialize the vector with empty bees */
for (int i = 0; i < beesPopulation; i++) {
    Bee initBee;
    beesVector.push_back(initBee);
}
/* initialize the best solution with the worst cost */
Bee bestSolution;
bestSolution.cost = inf;
/* initializing distances for the bees vector with random values and calculating costs */
for (int i = 0; i < beesPopulation; i++) {
    uniform_real_distribution<> distance(distanceMin, distanceMax);
    for (int j = 0; j < distanceSize; j++) {
        beesVector[i].pos.push_back(distance(gen));
    }
    beesVector[i].cost = optimizableFunction.getResult(beesVector[i].pos);
    if (beesVector[i].cost <= bestSolution.cost) {
        bestSolution = beesVector[i];
    }
}
/* vector for counting the abandoned bees */
vector<double> abandonedBees(beesPopulation, 0);
/* vector for keeping in memory the best cost for every iteration */
vector<double> bestCost(maxIterations, 0);
```

Se parcurg toate iterațiile, toate albinele “angajate”, se alege o albină la întâmplare care nu este cea curentă, se calculează un coeficient diferit de accelerare pentru fiecare distanță, se definește o nouă bază pentru albine, se memorează poziția pentru noua albină care este egală cu poziția curentă la care se adună produsul dintre diferența poziției curente cu poziția aleatoare [3] și coeficientul de accelerare. [1](pg. 461-463, (2.2)) În continuare se calculează noul cost. Dacă noul cost este mai bun, atunci se înlocuiește cu cel vechi, altfel este abandonat. [1](pg. 463)

```
int randomBeesIndex = randomBees[rand() % randomBees.size()];

/* calculate a different acceleration coefficient for every distance */
vector<double> accelCoef;
uniform_real_distribution<> acc(-1, +1);
for (int j = 0; j < distanceSize; j++) {
    accelCoef.push_back(accel * acc(gen));
}

/* define a new bee */
Bee newBee;
/* get the new bee position */
for (int j = 0; j < distanceSize; j++) {
    /* new bee position is equal to current bee's position + (current bee's position - random bee's position) * acceleration
      coefficient */
    newBee.pos.push_back(beesVector[i].pos[j] + (beesVector[i].pos[j] - beesVector[randomBeesIndex].pos[j]) * accelCoef[j]);
}
/* calculate the new cost */
newBee.cost = optimizableFunction.getResult(newBee.pos);

/* if the new cost is better */
if (newBee.cost <= beesVector[i].cost) {
    /* replace the old cost with the better one */
    beesVector[i] = newBee;
}
/* else abandon it */
else {
    abandonedBees[i] += 1;
}
```

Se declară un vector pentru calcularea valorilor de fitness, suma valorilor respective, costul mediu al tuturor albinelor. [1](pg. 462-463, (2.1)) Se calculează costul mediu, valorile de fitness, probabilitatea ca albina să fie selectată.

```
/* vector for calculating the fitness values */
vector<double> fitnessValues(beesEmployed, 0);
/* sum of fitness values */
double fSum = 0;
/* average cost of all the bees */
double averageCost = 0;
/* calculating the average cost */
for (int i = 0; i < fitnessValues.size(); i++) {
    averageCost += beesVector[i].cost;
}
averageCost /= fitnessValues.size();
/* calculating the fitness values */
for (int i = 0; i < fitnessValues.size(); i++) {
    fitnessValues[i] = pow(M_E, -beesVector[i].cost / averageCost);
    fSum += fitnessValues[i];
}
/* calculating probability of being selected */
vector<double> prob(beesEmployed, 0);
for (int i = 0; i < prob.size(); i++) {
    prob[i] = fitnessValues[i] / fSum;
}
```

Urmează etapa albinelor onlooker unde se selectează indexul albinei pe baza probabilității [2] și se pune ca fiind cel curent, se alege o albină care nu este cea curentă, se calculează un coeficient diferit de accelerare pentru fiecare distanță, se definește o nouă albină, se ia noua poziție care este egală cu poziția curentă la care se adună produsul dintre diferența poziției curente cu poziția aleatoare și coeficientul de accelerare, se calculează noul cost. Dacă noul cost este mai bun, atunci se înlocuiește cu cel vechi, altfel este abandonat. [1](pg. 463)

```
/* select a bee index based on the probability and make it the current one */
int i = fitnessProportionateSelection(prob);
```

```
int randomBeesIndex = randomBees[rand() % randomBees.size()];

/* calculate a different acceleration coefficient for every distance */
vector<double> accelCoef;
uniform_real_distribution<> acc(-1, +1);
for (int j = 0; j < distanceSize; j++) {
    accelCoef.push_back(accel * acc(gen));
}

/* define a new bee */
Bee newBee;
/* get the new bee position */
for (int j = 0; j < distanceSize; j++) {
    /* new bee position is equal to current bee's position + (current bee's position - random bee's position) * acceleration
      coefficient */
    newBee.pos.push_back(beesVector[i].pos[j] + (beesVector[i].pos[j] - beesVector[randomBeesIndex].pos[j]) * accelCoef[j]);
}
/* calculate the new cost */
newBee.cost = optimizableFunction.getResult(newBee.pos);

/* if the new cost is better */
if (newBee.cost <= beesVector[i].cost) {
    /* replace the old cost with the better one */
    beesVector[i] = newBee;
}
/* else abandon it */
else {
    abandonedBees[i] += 1;
}
```

Faza albinelor scout în care dacă albina abandonată este peste limita de abandonare, trebuie căutat din nou, se calculează cel mai bun cost, se salvează într-un vector.

```
/* scout bees phase */
for (int itScout = 0; itScout < beesScout; itScout++) {
    for (int i = 0; i < beesEmployed; i++) {
        /* if the abandoned bee is over the abandonment limit, make it search again */
        if (abandonedBees[i] >= limit) {
            uniform_real_distribution<> distance(distanceMin, distanceMax);
            for (int j = 0; j < distanceSize; j++) {
                beesVector[i].pos[j] = distance(gen);
            }
            beesVector[i].cost = optimizableFunction.getResult(beesVector[i].pos);
            abandonedBees[i] = 0;
        }
    }
}
```

Se calculează costul mediu, varianța, costul tuturor deviațiilor standard și se afișează.

Date experimentale

Pentru fiecare funcție [1](pg. 464-467) s-au calculat folosind 30 de experimente și 1500 de iterații următoarele:

f	Alg	ABC	
	Dim	Mean	SD
f1	10	0,178219	0,0617469
	20	0,0053709	0,008312
	30	0,154695	0,0746836
f2	10	15,7613	3,00867
	20	90,1958	10,7816
	30	204,964	10,7425
f3	10	5,1369	0,368844
	20	20,2929	9,69067
	30	738,148	374,381
f4	10	20,3358	0,050324
	20	20,7583	0,0743063
	30	20,9653	0,0479011
f5	10	5,56389e-309	0
	20	5,56325e-309	0
	30	5,56307e-309	0

Concluzii

În urma analizei datelor obținute din tabel și a datelor din tabelul din articol se poate observa că:

- s-au obținut rezultate notabil mai bune decât cele din articol pentru funcția f5 atât la Mean cât și la SD pentru toate cele 3 dimensiuni;
- s-au obținut rezultate aproximativ bune, nu chiar ca cele din articol pentru funcția f4 la SD pentru toate cele 3 dimensiuni;
- s-au obținut rezultate similare cu cele din articol pentru funcția f1 atât la Mean cât și la SD pentru cele 3 dimensiuni;
- s-au obținut rezultate aproximativ bune, nu chiar ca cele din articol pentru funcția f3 la SD pentru dimensiunea 10;
- în rest se pot observa diferențe semnificative în ceea ce privește valorile cu mențiunea că în cadrul tabelului din articol au fost dobândite rezultate mai bune.

Bibliografie

[1] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm," in *Journal of Global Optimization*, Springer, 2007, pp. 459-471.

[2] Wikipedia, the free encyclopedia, "Fitness proportionate selection - Wikipedia," 25 December 2019. [Online]. Available: https://en.wikipedia.org/wiki/Fitness_proportionate_selection. [Accessed April 2020].

[3] Yarpiz, "Artificial Bee Colony (ABC) in MATLAB," 11 September 2015. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/52966-artificial-bee-colony-abc-in-matlab. [Accessed April 2020].