Projet IA 2

Bensmail Kamelia & Ouarafana Badr

Avril 2023

Table des matières

1 Préambule

Ce tp est composé de deux programmes différents. Un pour l'algorithme génétique, et la colonie de fourmis.

Un Makefile est généré pour la compilation de programme.

On suppose que dans le graphe tous les sommets sont reliés entre eux, le graphe est représenté sous forme d'une matrice, ou chaque case i, j représente la distance entre la ville de l'indice i avec la ville j.

```
float distances[NUMBEROFCITIES][NUMBEROFCITIES] = {
      {0, 10, 20, 30, 40, 50, 60, 70, 80, 90},
      {10, 0, 15, 25, 35, 45, 55, 65, 75, 85},
      {20, 15, 0, 10, 20, 30, 40, 50, 60, 70},
      {30, 25, 10, 0, 15, 25, 35, 45, 55, 65},
      {40, 35, 20, 15, 0, 10, 20, 30, 40, 50},
      {50, 45, 30, 25, 10, 0, 15, 25, 35, 45},
      {60, 55, 40, 35, 20, 15, 0, 10, 20, 30},
      {70, 65, 50, 45, 30, 25, 10, 0, 15, 25},
      {80, 75, 60, 55, 40, 35, 20, 15, 0, 10},
      {90, 85, 70, 65, 50, 45, 30, 25, 10, 0}};
```

2 Algorithme génétique pour le problème TSP

2.1 Création de la population initiale

La fonction pour créer la population est la suivante :

```
void createPopulation(int **population){
  int cities[NUMBEROFCITIES];

  // les numeros des villes [0,1,2,3,..., n]
  for (int i = 0; i < NUMBEROFCITIES; i++){
    cities[i] = i;
  }

  // création d'un population aléatoire
  for (int i = 0; i < NUMBEROFPOPULATION; i++){
    shuffleArray(cities, NUMBEROFCITIES);
    copyArray(population[i], cities, NUMBEROFCITIES);
  }
}
```

On commence par créer la population initiale de la manière suivante :

- Stocker les indices des villes $[0, 1, \cdots n]$ dans un vecteur `int * cities` ou n = le nombre de villes souhaitées.
- Changer l'ordre des éléments d'une manière aléatoire avec la fonction `shuffleArray`.
- Stocker chaque chromosome dans un vecteur 2D `int ** population`

La population est modélisée sous forme d'une matrice 2D des indices de villes.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 0 \\ 2 & 1 & 3 & 0 & 4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 3 & 4 & 2 & 1 \end{bmatrix} \quad (N \text{ nombre de population} \times n \text{ nombre de villes })$$

Fonction pour créer un chromosome aléatoire.

```
void shuffleArray(int *T, int size) {
  int tmp;
  // changer l'ordre des villes aléatoirement
  for (int i = 0; i < size; i++){
    tmp = rand() % (size - 1) + 1;
    swap(&T[i], &T[tmp]);
  }
}
```

On itère sur tout le tableau et dans chaque itération, on change l'élément i avec un élément aléatoire `tmp`.

2.2 Calcul de la fonction f(n)

```
float calculatFitness(int *population, float graph[NUMBEROFCITIES][NUMBEROFCITIES]){
  float distance = 0;
  float sum = 0;
  // calculer la distance de chaque chromosome
  for (int j = 0; j < NUMBEROFCITIES; j++){
  // chercher les distance dans la matrice graph, population[j] represente la ville courante,
  //population[j+1] représente la ville suivante, le modulo pour revenir à la ville de départ
    distance = graph[population[j]][population[(j + 1) % NUMBEROFCITIES]];
    sum += distance;
  }
  return sum;
}
```

La fonction ci-dessus, calcule la fonction f(n) : la somme de distances pour un chemin donné d'un chromosome.

Le chromosome contient les indices du chemin, exemple : 1 0 5 2 4 3 et on cherche la distance dans notre matrice, ou `population[j]` est l'indice de la ville courante, et `population[(j + 1) % NUMBEROFCITIES]` est l'indice la ville suivante.

Le modulo, pour compter le chemin du retour du dernier sommet au sommet du départ.

2.3 Choix des parents

```
int *selectParents(int **population, float *fitness){
  int *parents = (int *)malloc(2 * sizeof(int));
  float cumulativeProbs[NUMBEROFPOPULATION];
  float total = 0;
  float randValue = 0.0f; // random value
  float sumFitness = sum(fitness, NUMBEROFPOPULATION);
  for (int i = 0; i < NUMBEROFPOPULATION; i++){
    total += fitness[i];
    cumulativeProbs[i] = total / sumFitness;
  }

  // choix des parents
  do{
    for (int j = 0; j < 2; j++){
      randValue = (float)rand() / RAND_MAX;
      for (int i = 0; i < NUMBEROFPOPULATION; i++){
        if (randValue < cumulativeProbs[i]){
        parents[j] = i;
        break;
        }
      }
    }
  } while (areTheSame(population[parents[0]], population[parents[1]], NUMBEROFCITIES));
  //verifier que les genomes ne sont pas pareils
  return parents;
}
```

On choisi les parents en utilisant la méthode "Selective wheel", ou un parent avec une valeur f(n) minimale a plus de chance d'être choisi

la fonction `int *selectParents(int **population, float *fitness)` renvoie un tableau de deux indices des parents dans la population qui vont représenter n et n'.

Le vecteur `int ** population` représente la population, et `int * fitness` représente la valeur f(n) pour chaque chromosome.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 0 \\ 2 & 1 & 3 & 0 & 4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 3 & 4 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 10 \\ 25 \\ \vdots \\ 31 \end{bmatrix} \longrightarrow f(n)$$

On commence par calculer la somme totale de f(n) pour toute la population, et créer le tableau contenant des sommes des probabilités cumulatives `float cumulativeProbs[NUMBEROFPOPULATION]` .

$$P(X \leq x) = \sum_{i=1}^{n} p_i \cdot \mathbf{1}_{\{x_i \leq x\}}$$

On choisi un nombre random entre $[0, 1]$ et si cette valeur est inférieure à un élément du tableau on le choisit.

Pour finir on s'assure que les parents sont différents avec la boucle `do {} while();`

2.4 Croisement

```
int *getGenome(int *parentA, int *parentB, int offset){
  int *genome = (int *)malloc(sizeof(int) * NUMBEROFCITIES);
  int i;
  bool isHere = false;

  for (i = 0; i < offset; i++){
    genome[i] = parentA[i];
  }
  for (int j = 0; j < NUMBEROFCITIES; j++, isHere = false){
    for (int k = 0; k < offset; k++){
      if (parentB[j] == parentA[k]){
      isHere = true;
      break;
      }
    }
    if (!isHere){
      genome[i++] = parentB[j];
    }
  }
  return genome;
}
```

Pour calculer le croisement, entre deux parents :

- On copie les éléments de l'un des parents jusqu'à un offset choisi dans le nouveau génome. Exemple avec offset = 3.

0	1	2	3	4	5
5	1	3	2	4	0
0	1	2	x	x	x

- On cherche les éléments dans le deuxième parent qui ne sont pas présents dans le nouveau génome et le premier parent, et on les ajoute dans les cellules vides.

0	1	2	3	4	5
5	1	3	2	4	0
0	1	2	5	3	4

Et retourne le nouveau génome n^* le résultat du croisement des deux parents, qui sera ajouté par la suite à la nouvelle population.

2.5 Mutation

```
void mutateGenome(int *genome){
  int i = rand() % NUMBEROFCITIES;
  int j = 0;

  do{
    j = rand() % NUMBEROFCITIES;
  } while (i == j);

  swap(&genome[i], &genome[j]);
}
```

Cette fonction choisie aléatoirement deux indices i et j tel que $i \neq j$ et effectue un permutation, ce qui represente un mutation.

2.6 Fonction main

On commence par allouer l'espace mémoire et créer la population initiale.

```
int **population = (int **)malloc(NUMBEROFPOPULATION * sizeof(int *));
  for (int i = 0; i < NUMBEROFPOPULATION; i++){
    population[i] = (int *)malloc(sizeof(int) * NUMBEROFCITIES);
  }
  // generer aléatoirement la population
  createPopulation(population);
```

Par la suite, on répète le processus suivant k nombre de fois :

On crée une nouvelle population

```
for (int k = 0; i<nbIteration; k++){
  // création de la nouvelle de la nouvelle population
  int **newPopulation = (int **)malloc(NUMBEROFPOPULATION * sizeof(int *));
  for (int i = 0; i < NUMBEROFPOPULATION; i++){
    newPopulation[i] = (int *)malloc(sizeof(int) * NUMBEROFCITIES);
  }
```

Et pour chaque chomosome dans la population précédente :

- On calcule le fonction fitness f(n).
- On choisi les parents avec une probabilité qui augmente quand f(n) diminue.
- On calcule le croisement des deux parents.
- Avec une probabilité faible on effectue une mutation au génome.
- On rajoute ce dernier à la nouvelle population.

```
for (int j = 0; j < NUMBEROFPOPULATION; j++){
  // calculer f(n) de chaque chromosome
  for (int i = 0; i < NUMBEROFPOPULATION; i++){
    fitnessValues[i] = calculatFitness(population[i], graph);
  }
  // choisir les parents avec une proba qui augmente avec f(n) ==> Wheel selection
  parents = selectParents(population, fitnessValues);

  // calculer le genome fils n*
  genome = getGenome(population[parents[0]], population[parents[1]], 2);

  // effectuer la mutation au genome avec une probabilité de 20%
  randValue = (float)rand() / RAND_MAX;
  if (randValue < 0.3){
    mutateGenome(genome);
  }
  copyArray(newPopulation[j], genome, NUMBEROFCITIES);
```

Et pour finir, on enregistre la solution optimale estimée et on libère la mémoire.

```
sumFitness = calculatFitness(genome, graph);
if (sumFitness < optimalValue){
  optimalValue = sumFitness;
  copyArray(optimalPath, genome, NUMBEROFCITIES);
}

free(genome);
free(parents);
genome = NULL;
parents = NULL;
}

// liberer m'éspace mémoire
for (int i = 0; i < NUMBEROFPOPULATION; i++){
  free(population[i]);
}
free(population);
population = newPopulation;
newPopulation = NULL;
}
```

3 Colonie de Fourmis

3.1 Modélisation

Pour cette implémentation, nous avons utilisé la programmation orientée objet pour représenter un Sommet, et un Arête.

un Sommet est caractérisé par : un indice, un vecteur de voisins et un booléen pour marquer le sommet visité.

```
class Sommet {
  protected :
  std::vector <Arete *> listeArete;
  int indice ;
  bool estVisite ;

  public :
  Sommet (int) ;
  ~Sommet () ;
  int getIndice() {return this->indice ;}
  std::vector <Arete *> getNeighbors () {return this->listeArete;}
  bool getEstVisite(){
  return this->estVisite;}
  void setEstVisite(bool b){
  this->estVisite=b;}
  void afficherVoisins();
  void ajouterVoisin(Arete *);
  void marquer();
};
```

La classe Arete représente l'arête entre un sommet A et un sommet B avec la distance et le niveau de phéromone déposé par une fourmi.

```
class Arete{
private:
Sommet * end ;
float distance;
float phermone;
public :
Arete ( Sommet * , float,float) ;
~Arete ();
Sommet * getEnd() ;
float getDistance(){return this->distance;}
void setDistance(float d){this->distance=d;}
float getPhermone(){return this->phermone;}
void setPhermone(float p) {this->phermone=p;}
};
```

3.2 Calcul de probabilité

La fonction pour calculer la probabilité d'une fourmi pour choisir une ville "règle aléatoire de transition proportionnelle" :

$$p_{i,j} = \frac{[\tau_{i,j}(t)]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{k \in N'_i(t)}[\tau_{i,k}(t)]^{\alpha}[\eta_{i,k}]^{\beta}}$$

```
float calculerProbabilite(Arete *arete, std::vector<Arete *> voisins, int alpha, int beta){
  float somme = 0;
  float phermone = 0;
  float distance = 0;
  for (int i = 0; i < (int)voisins.size(); i++){
    if (voisins.at(i)->getEnd()->getEstVisite()) continue; // ville déjà visitée
    phermone = voisins.at(i)->getPhermone();
    distance = voisins.at(i)->getDistance();
    somme += pow(phermone, alpha) * pow((1 / distance), beta);
  }
  return ((pow(arete->getPhermone(), alpha)) * (pow(1 / arete->getDistance(), beta))) / somme;
}
```

3.3 Probabilité cumulative

```
Sommet *selectionWheel(std::vector<float> probabilites, std::vector<Arete *> voisinsAchoisir){
  std::vector<float> cumulativeProbabilities;
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<> dis(0.0, 1.0);
  float randomNum = dis(gen); // générer un nombre aléatoire de 0 à 1.
  float sum = 0;
  int j = 0;

  for (float i : probabilites){
    sum += i;
    cumulativeProbabilities.push_back(sum);
  }

  for (; j < (int)cumulativeProbabilities.size(); j++){
    if (randomNum <= cumulativeProbabilities[j]){
      return voisinsAchoisir.at(j)->getEnd();
    }
  }
  // j sera l'indice la ville suivante dans le cas ou il reste une seule ville
  return voisinsAchoisir.at(j - 1)->getEnd();
}
```

Cette fonction suit le même principe que l'algorithme précédent :

On crée un vecteur pour stocker les probabilités cumulatives, on génère une valeur aléatoire r entre $[0, 1]$, et on choisi la première valeur dans le vecteur inférieur ou égale à r, et on renvoie l'arête associée à cette valeur qui sera le voisin suivant.

3.4 Déplacement d'une fourmi

On commence par poser une fourmi dans une ville aléatoire, et tant qu'on a pas visité tous les sommets en effectue les étape suivante :

- On calcule les probabilités de déplacement d'une fourmi pour chacun des voisins de la ville de départ.

```
// on calcule les probas pour chaque arete
for (Arete *a : villeDepart->getNeighbors()){
  if (a->getEnd()->getEstVisite()){
    continue;
  }
  else{
    voisinsAchoisir.push_back(a);
  }
  probabilites.push_back(calculerProbabilite(a, villeDepart->getNeighbors(), alpha, beta));
}
```

- On choisi la ville suivante par la méthode du "Selective wheel selection"
- On calcule la distance du chemin de chaque ville parcouru, et on renvoie une structure à la fin qui regroupe le chemin et la distance.

```
distance += aretes[path.at(0)->getIndice()][path.at(path.size() - 1)->getIndice()]->getDistance();
path.push_back(path.at(0));
return {path, distance};
```

3.5 Algorithme

Concernant l'algorithme pour chaque itération, on effectue ces étapes :

- On calcul le chemin parcouru par toutes les fourmis et on les stocks dans un vecteur.
- On met à jour le niveau de phéromone (vaporisation).
- On ajoute le phéromone pour chaque passage des fourmis.

Calculer le chemin pour chaque fourmi

```
for (int j = 0; j < nbrFourmis; j++){
  cheminDesFourmis.push_back(calculerCheminParUneFourmie(graph, aretes, alpha, beta));
}
```

La fonction prend en paramètre un vecteur de sommets : graph, une matrice des arêtes pour récupérer les informations nécessaires (distance entre deux sommets, niveau de pheromone ...) et les paramètres de réglages.

Calcule le chemin emprunté far chaque fourmi, et l'ajoute dans le vecteur `std::vector<Solution> cheminDesFourmis`

Nota : la structure Solution contient un chemin "Un vecteur de Sommet" et la distance appropriée.

```
for (int i = 0; i < NUMBEROFCITIES; i++){
  for (int j = 0; j < NUMBEROFCITIES; j++){
    if (j == i)
      continue;
    aretes[i][j]->setPhermone((1 - gho) * aretes[i][j]->getPhermone());
  }
}
```

On parcourt la matrice des arêtes, et on modifie la valeur du pheromone par $\frac{dP}{dt} = -\gamma_{h,o}P$

```
for (Solution s : cheminDesFourmis){
  for (int i = 0; i < (int)s.chemin.size() - 1; i++){
    int cur = s.chemin.at(i)->getIndice();
    int next = s.chemin.at(i + 1)->getIndice();
    aretes[cur][next]->setPhermone(aretes[cur][next]->getPhermone() + (1 / s.distance));
  }
  // verifier la solution optimale
  if (s.distance < solutionOptimale.distance){
    solutionOptimale = s;
  }
}
```

Dernière étape de l'algorithme, c'est de rajouter le niveau de phéromone déposé par chaque fourmi en passant par un chemin.

Nota : on suppose que le niveau de phermone sur toutes les arêtes est de 1 .

4 Comparaison des algorithmes

En général, nous avons remarqué que le deuxième algorithme est plus efficace et rapide en trouvant le chemin minimal, et ça marche plutôt bien avec des grandes matrices, en revanche, l'algorithme génétique semble limité quand on dépasse 10 sommets.

Comparaison 5 sommets :

On remarque que pour un graphe de 5 sommets, les résultats sont les mêmes et l'implémentation est assez rapide même pour un grand nombre de tours.

```
→ ALGO_GENETIC git:(master) ✗ time ./algoGenetic
       [0]     [1]     [2]     [3]     [4]
[0]    0.00    281.00  176.00  243.00  106.00
[1]    281.00  0.00    25.00   192.00  65.00
[2]    176.00  25.00   0.00    193.00  17.00
[3]    243.00  192.00  193.00  0.00    138.00
[4]    106.00  65.00   17.00   138.00  0.00
================================================
Bien verifier les paramètres !!
Nombre de tours : 100000
Nombre de populations : 50
Nombre de sommets : 5
offset pour croisement : 3
================================================
Estimated Optimal value is 583.00
3 -> 0 -> 4 -> 2 ->  -> 1
{3 ==> 0 = 243.00}
{0 ==> 4 = 106.00}
{4 ==> 2 = 17.00}
{2 ==> 1 = 25.00}
{1 ==> 3 = 192.00}
./algoGenetic  6,43s user 0,01s system 99% cpu 6,439 total
→ ALGO_GENETIC git:(master) ✗
```

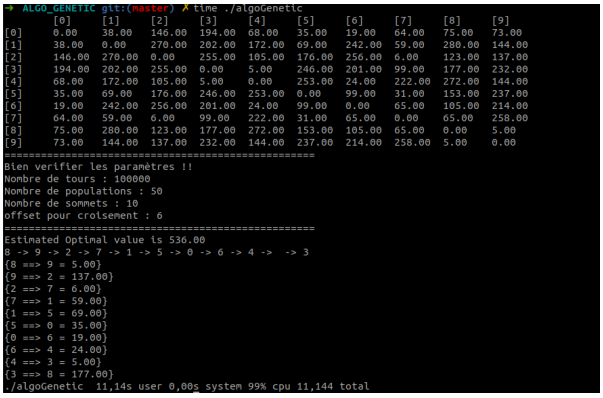
(a) Algorithme génétique

```
→ ANT_COLONY git:(master) ✗ make
g++ -c ant_colony.cpp -Wall -std=c++2a -g
g++ ant_colony.o  Arete.o Sommet.o -o ant_colony -g
→ ANT_COLONY git:(master) ✗ time ./ant_colony
       [0]     [1]     [2]     [3]     [4]
[0]    0.00    281.00  176.00  243.00  106.00
[1]    281.00  0.00    25.00   192.00  65.00
[2]    176.00  25.00   0.00    193.00  17.00
[3]    243.00  192.00  193.00  0.00    138.00
[4]    106.00  65.00   17.00   138.00  0.00
================================================
Bien verifier les paramètres !!
Nombre de tours : 1000
Nombre de fourmis : 50
Nombre de sommets : 5
Alpha 1, Beta 1, Gho 0.50
================================================
================================================
Estimated Optimal value is: 583
0 -->  4 -->  2 -->  1 -->  3 -->  0
================================================
{0 -> 4} : 106
{4 -> 2} : 17
{2 -> 1} : 25
{1 -> 3} : 192
{3 -> 0} : 243
./ant_colony  7,07s user 0,00s system 99% cpu 7,075 total
→ ANT_COLONY git:(master) ✗
```

(b) Colonie de fourmis

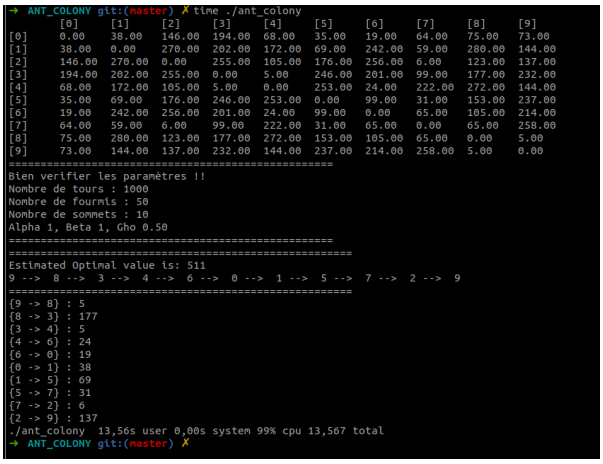
FIGURE 1

Pour 10 sommets, on remarque que les deux algorithmes prennent un peu plus de temps, or tempas consomé par l'algorithme de colonie de fourmis est inferieur est il donne un meilleur résultat.



```
→ ALGO_GENETIC git:(master) ✗ time ./algoGenetic
      [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]     [8]     [9]
[0]   0.00    38.00   146.00  194.00  68.00   35.00   19.00   64.00   75.00   73.00
[1]   38.00   0.00    270.00  202.00  172.00  69.00   242.00  59.00   280.00  144.00
[2]   146.00  270.00  0.00    255.00  105.00  176.00  256.00  6.00    123.00  137.00
[3]   194.00  202.00  255.00  0.00    5.00    246.00  201.00  99.00   177.00  232.00
[4]   68.00   172.00  105.00  5.00    0.00    253.00  24.00   222.00  272.00  144.00
[5]   35.00   69.00   176.00  246.00  253.00  0.00    99.00   31.00   153.00  237.00
[6]   19.00   242.00  256.00  201.00  24.00   99.00   0.00    65.00   105.00  214.00
[7]   64.00   59.00   6.00    99.00   222.00  31.00   65.00   0.00    65.00   258.00
[8]   75.00   280.00  123.00  177.00  272.00  153.00  105.00  65.00   0.00    5.00
[9]   73.00   144.00  137.00  232.00  144.00  237.00  214.00  258.00  5.00    0.00
==================================================
Bien verifier les paramètres !!
Nombre de tours : 100000
Nombre de populations : 50
Nombre de sommets : 10
offset pour croisement : 6
==================================================
Estimated Optimal value is 536.00
8 -> 9 -> 2 -> 7 -> 1 -> 5 -> 0 -> 6 -> 4 ->  -> 3
{8 ==> 9 = 5.00}
{9 ==> 2 = 137.00}
{2 ==> 7 = 6.00}
{7 ==> 1 = 59.00}
{1 ==> 5 = 69.00}
{5 ==> 0 = 35.00}
{0 ==> 6 = 19.00}
{6 ==> 4 = 24.00}
{4 ==> 3 = 5.00}
{3 ==> 8 = 177.00}
./algoGenetic  11,14s user 0,00s system 99% cpu 11,144 total
```

(a) Algorithme génétique

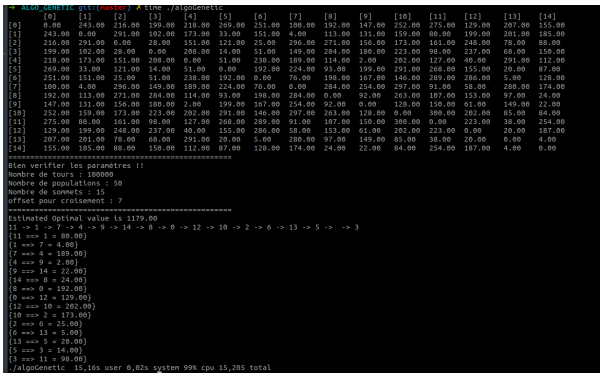


```
→ ANT_COLONY git:(master) ✗ time ./ant_colony
      [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]     [8]     [9]
[0]   0.00    38.00   146.00  194.00  68.00   35.00   19.00   64.00   75.00   73.00
[1]   38.00   0.00    270.00  202.00  172.00  69.00   242.00  59.00   280.00  144.00
[2]   146.00  270.00  0.00    255.00  105.00  176.00  256.00  6.00    123.00  137.00
[3]   194.00  202.00  255.00  0.00    5.00    246.00  201.00  99.00   177.00  232.00
[4]   68.00   172.00  105.00  5.00    0.00    253.00  24.00   222.00  272.00  144.00
[5]   35.00   69.00   176.00  246.00  253.00  0.00    99.00   31.00   153.00  237.00
[6]   19.00   242.00  256.00  201.00  24.00   99.00   0.00    65.00   105.00  214.00
[7]   64.00   59.00   6.00    99.00   222.00  31.00   65.00   0.00    65.00   258.00
[8]   75.00   280.00  123.00  177.00  272.00  153.00  105.00  65.00   0.00    5.00
[9]   73.00   144.00  137.00  232.00  144.00  237.00  214.00  258.00  5.00    0.00
==================================================
Bien verifier les paramètres !!
Nombre de tours : 1000
Nombre de fourmis : 50
Nombre de sommets : 10
Alpha 1, Beta 1, Gho 0.50
==================================================
==================================================
Estimated Optimal value is: 511
9 -->  8 -->  3 -->  4 -->  6 -->  0 -->  1 -->  5 -->  7 -->  2 -->  9
==================================================
{9 -> 8} : 5
{8 -> 3} : 177
{3 -> 4} : 5
{4 -> 6} : 24
{6 -> 0} : 19
{0 -> 1} : 38
{1 -> 5} : 69
{5 -> 7} : 31
{7 -> 2} : 6
{2 -> 9} : 137
./ant_colony  13,56s user 0,00s system 99% cpu 13,567 total
→ ANT_COLONY git:(master) ✗
```

(b) Colonie de fourmis

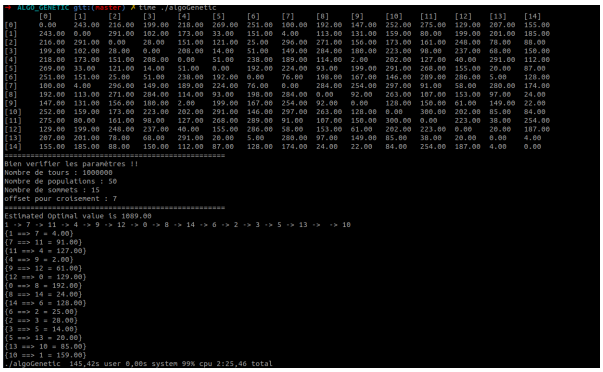
FIGURE 2

Et là, on met le nombre de sommets à 15, et on remarque qu'on même augmentant le nombre d'itérations, et changer les réglages(nombre de population, offset ...etc), le deuxième algorithme est plus performant, et en moins laps de temps.



```
→ ALGO_GENETIC git:(master) ✗ time ./algoGenetic
      [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]     [8]     [9]     [10]    [11]    [12]    [13]    [14]
[0]   0.00    243.00  216.00  199.00  218.00  269.00  251.00  100.00  192.00  147.00  252.00  275.00  129.00  207.00  155.00
[1]   243.00  0.00    291.00  102.00  173.00  33.00   151.00  4.00    113.00  131.00  159.00  88.00   199.00  201.00  185.00
[2]   216.00  291.00  0.00    28.00   151.00  121.00  25.00   296.00  271.00  156.00  173.00  161.00  248.00  78.00   88.00
[3]   199.00  102.00  28.00   0.00    208.00  14.00   51.00   149.00  284.00  180.00  223.00  98.00   237.00  68.00   150.00
[4]   218.00  173.00  151.00  208.00  0.00    51.00   238.00  189.00  114.00  2.00    202.00  127.00  40.00   291.00  112.00
[5]   269.00  33.00   121.00  14.00   51.00   0.00    192.00  224.00  93.00   199.00  291.00  268.00  155.00  26.00   87.00
[6]   251.00  151.00  25.00   51.00   238.00  192.00  0.00    76.00   198.00  167.00  146.00  289.00  286.00  5.00    128.00
[7]   100.00  4.00    296.00  149.00  189.00  224.00  76.00   0.00    284.00  254.00  297.00  91.00   58.00   285.00  174.00
[8]   192.00  113.00  271.00  284.00  114.00  93.00   198.00  284.00  0.00    92.00   263.00  107.00  153.00  97.00   24.00
[9]   147.00  131.00  156.00  180.00  2.00    199.00  167.00  254.00  92.00   0.00    128.00  150.00  61.00   149.00  22.00
[10]  252.00  159.00  173.00  223.00  202.00  291.00  146.00  297.00  263.00  128.00  0.00    300.00  202.00  85.00   84.00
[11]  275.00  80.00   161.00  98.00   127.00  268.00  289.00  91.00   107.00  150.00  300.00  0.00    223.00  38.00   254.00
[12]  129.00  199.00  248.00  237.00  40.00   155.00  286.00  58.00   153.00  61.00   202.00  223.00  0.00    20.00   187.00
[13]  207.00  201.00  78.00   68.00   291.00  20.00   5.00    280.00  97.00   149.00  85.00   38.00   20.00   0.00    4.00
[14]  155.00  185.00  88.00   150.00  112.00  87.00   128.00  174.00  24.00   22.00   84.00   254.00  187.00  4.00    0.00
==================================================
Bien verifier les paramètres !!
Nombre de tours : 100000
Nombre de populations : 50
Nombre de sommets : 15
offset pour croisement : 7
==================================================
Estimated Optimal value is 1179.00
11 -> 1 -> 7 -> 4 -> 9 -> 14 -> 8 -> 0 -> 12 -> 10 -> 2 -> 6 -> 13 -> 5 ->  -> 3
{11 ==> 1 = 88.00}
{1 ==> 7 = 4.00}
{7 ==> 4 = 189.00}
{4 ==> 9 = 2.00}
{9 ==> 14 = 22.00}
{14 ==> 8 = 24.00}
{8 ==> 0 = 192.00}
{0 ==> 12 = 129.00}
{12 ==> 10 = 202.00}
{10 ==> 2 = 173.00}
{2 ==> 6 = 25.00}
{6 ==> 13 = 5.00}
{13 ==> 5 = 20.00}
{5 ==> 3 = 14.00}
{3 ==> 11 = 98.00}
./algoGenetic  15,16s user 0,02s system 99% cpu 15,205 total
```

(a) Algorithme génétique avec 100000 itération



```
→ ALGO_GENETIC git:(master) ✗ time ./algoGenetic
      [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]     [8]     [9]     [10]    [11]    [12]    [13]    [14]
[0]   0.00    243.00  216.00  199.00  218.00  269.00  251.00  100.00  192.00  147.00  252.00  275.00  129.00  207.00  155.00
[1]   243.00  0.00    291.00  102.00  173.00  33.00   151.00  4.00    113.00  131.00  159.00  88.00   199.00  201.00  185.00
[2]   216.00  291.00  0.00    28.00   151.00  121.00  25.00   296.00  271.00  156.00  173.00  161.00  248.00  78.00   88.00
[3]   199.00  102.00  28.00   0.00    208.00  14.00   51.00   149.00  284.00  180.00  223.00  98.00   237.00  68.00   150.00
[4]   218.00  173.00  151.00  208.00  0.00    51.00   238.00  189.00  114.00  2.00    202.00  127.00  40.00   291.00  112.00
[5]   269.00  33.00   121.00  14.00   51.00   0.00    192.00  224.00  93.00   199.00  291.00  268.00  155.00  26.00   87.00
[6]   251.00  151.00  25.00   51.00   238.00  192.00  0.00    76.00   198.00  167.00  146.00  289.00  286.00  5.00    128.00
[7]   100.00  4.00    296.00  149.00  189.00  224.00  76.00   0.00    284.00  254.00  297.00  91.00   58.00   285.00  174.00
[8]   192.00  113.00  271.00  284.00  114.00  93.00   198.00  284.00  0.00    92.00   263.00  107.00  153.00  97.00   24.00
[9]   147.00  131.00  156.00  180.00  2.00    199.00  167.00  254.00  92.00   0.00    128.00  150.00  61.00   149.00  22.00
[10]  252.00  159.00  173.00  223.00  202.00  291.00  146.00  297.00  263.00  128.00  0.00    300.00  202.00  85.00   84.00
[11]  275.00  80.00   161.00  98.00   127.00  268.00  289.00  91.00   107.00  150.00  300.00  0.00    223.00  38.00   254.00
[12]  129.00  199.00  248.00  237.00  40.00   155.00  286.00  58.00   153.00  61.00   202.00  223.00  0.00    20.00   187.00
[13]  207.00  201.00  78.00   68.00   291.00  20.00   5.00    280.00  97.00   149.00  85.00   38.00   20.00   0.00    4.00
[14]  155.00  185.00  88.00   150.00  112.00  87.00   128.00  174.00  24.00   22.00   84.00   254.00  187.00  4.00    0.00
==================================================
Bien verifier les paramètres !!
Nombre de tours : 1000000
Nombre de populations : 50
Nombre de sommets : 15
offset pour croisement : 7
==================================================
Estimated Optimal value is 1089.00
1 -> 7 -> 11 -> 4 -> 9 -> 12 -> 0 -> 8 -> 14 -> 6 -> 2 -> 3 -> 5 -> 13 ->  -> 10
{1 ==> 7 = 4.00}
{7 ==> 11 = 91.00}
{11 ==> 4 = 127.00}
{4 ==> 9 = 2.00}
{9 ==> 12 = 61.00}
{12 ==> 0 = 129.00}
{0 ==> 8 = 192.00}
{8 ==> 14 = 24.00}
{14 ==> 6 = 128.00}
{6 ==> 2 = 25.00}
{2 ==> 3 = 28.00}
{3 ==> 5 = 14.00}
{5 ==> 13 = 20.00}
{13 ==> 10 = 85.00}
{10 ==> 1 = 159.00}
./algoGenetic  145,42s user 0,00s system 99% cpu 2:25,46 total
```

(b) Algorithme génétique avec 1000000 itération

FIGURE 3

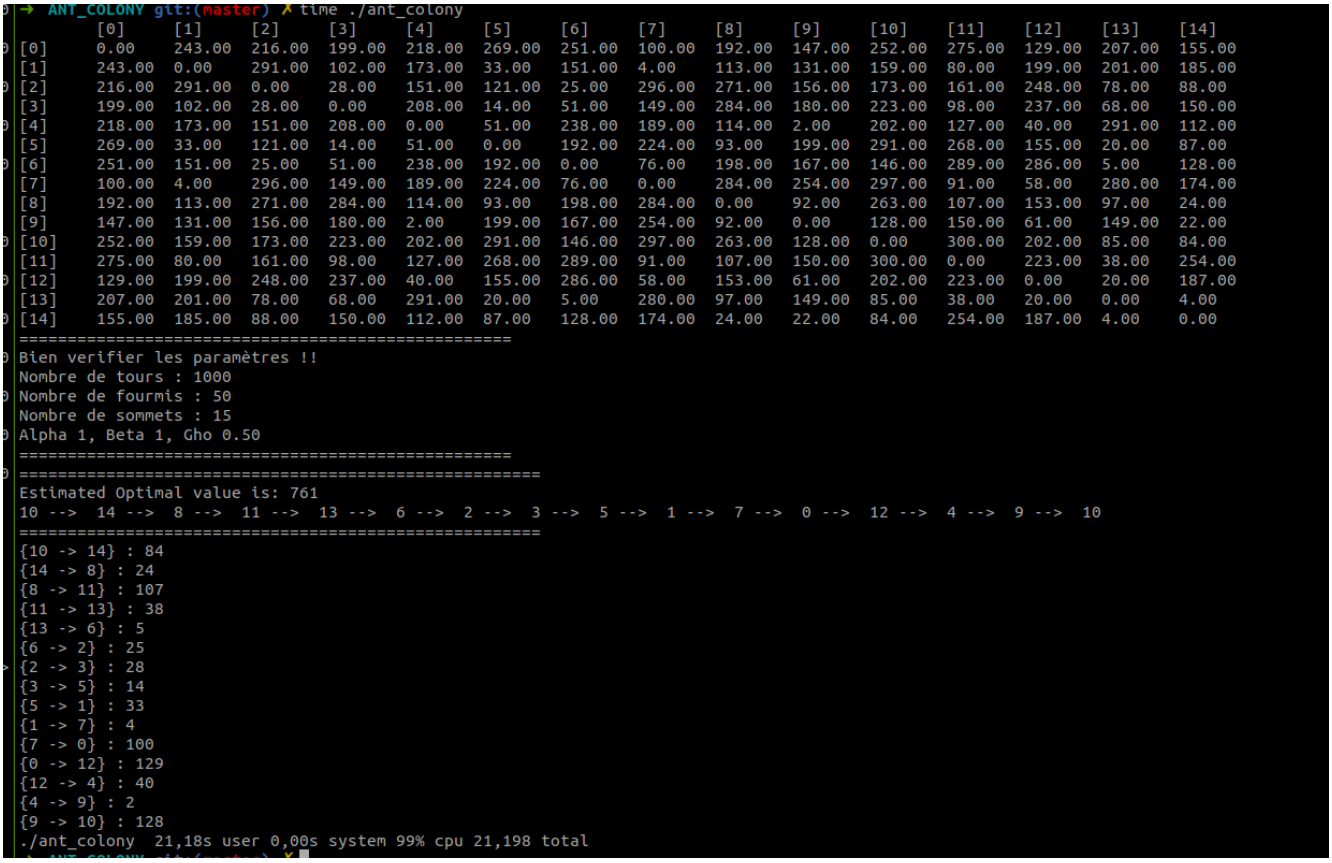


FIGURE 4 – Algorithme Colonie de fourmis avec 15 sommets