



Algorithmique avancée

Le voyageur de commerce

Rémy Lalande, Jean-Marie Le Yaouanc, Yvette Le Bras

Brest, 4 avril 2005

Enseignant responsable : Laurent Lemarchand

Département informatique

UFR Sciences et Techniques 6 avenue Le Gorgeu 29200 Brest

Master 1 informatique

Table des matières

1	L’algorithme génétique	1
2	Les différentes phases	3
2.1	La sélection	3
2.2	Le Cross-Over ou Recombinaison	3
2.3	La mutation	4
2.4	Les différents paramètres	5
2.4.1	Les paramètres de la population	5
2.4.2	Les paramètres des conditions d’arrêt	5
3	Nos résultats	6
4	Problèmes rencontrés	9
5	Optimisation possible	10
5.1	L’algorithme 2opt	10
5.2	Optimisation de la rapidité	10
A	Fichier d’entête	11
B	Code de l’algorithme génétique	13

Table des figures

2.1	Représentation graphique d'un Cross Over	4
3.1	Evolution des résultats avec des petites valeurs	7
3.2	Evolution des résultats avec des grandes valeurs	7
3.3	Evolution des résultats avec des valeurs correctes	8

Chapitre 1

L'algorithme génétique

Il s'agit de résoudre un problème d'optimisation d'une combinaison de paramètres dépendants les uns des autres. Ces paramètres peuvent prendre plusieurs valeurs prédéfinies, de la même manière que des gènes peuvent prendre plusieurs expressions. Une solution particulière au problème (une solution approchée) est représentée par l'ensemble des valeurs prises pour chaque paramètre, qui constitue en quelque sorte l'ADN de cette solution.

L'adaptation à l'environnement est représentée par une évaluation chiffrée indiquant la pertinence de la solution. Pour le PVC : un individu est un trajet, son évaluation est la longueur totale de ce trajet, un gène est la ville où l'on passe à une certaine position dans le parcours, et l'ADN est la liste des villes dans l'ordre du parcours.

Parmi un ensemble de solutions approchées (la population), on sélectionne alors des bonnes solutions et on les recombine pour en produire une nouvelle. D'autre part, on élimine les solutions les moins adaptées. En répétant ce processus, l'adaptation de la population augmente, et donc on converge vers la solution du problème.

De manière plus formelle, voici l'algorithme génétique utilisé pour la réalisation du PVC :

```
Pour chaque Individu dans Population
  Initialiser Individu
FinPour
Pour nombre_Itérations
  Si (hasard < pourcentage_Mutation) Alors
    Appliquer_une_mutation_à Fils;
  FinSi
  Si (hasard < pourcentage_CrossOver) Alors
    Parent A = Sélection_d'un_Individu (Population);
    Parent B = Sélection_d'un_Individu (Population);
    Fils = CrossOver (Parent A, Parent B);
    Evaluer fitness_Fils;
  Finsi
Faire une sélection;
FinPour
```

Chapitre 2

Les différentes phases

2.1 La sélection

Pour choisir les parents reproducteurs, on utilise la stratégie élitiste inspirée de la "roulette wheel" : On considère la sélection d'un individu comme le lancement d'une bille sur une roulette. Chaque individu possède plusieurs cases dans la roulette. Les meilleurs individus possèdent plus de cases et sont donc sélectionnés plus souvent.

Pour cela les individus sont classés selon l'inverse de leur fitness (individu ayant le plus petit nombre de cases= individu ayant la longueur de trajet la plus longue). On calcule la taille complète de la 'roulette' en fonction des fitness des individus puis on sélectionne un **winner**. Comme la fonction **rand()** renvoie un entier au hasard entre 1 et **tailleRoulette** il suffit de tirer un chiffre au hasard et de le comparer au chiffre du tableau des individus de la population pour connaître l'individu sélectionné.

Pour que chaque individu ait la même probabilité d'être sélectionné, il suffirait de positionner les chiffres du tableau à un intervalle de $\frac{\text{tailleRoulette}}{\text{Nombre_individus}}$.

2.2 Le Cross-Over ou Recombinaison

C'est la fonction principale de l'algorithme génétique. Le Cross-Over consiste à créer un individu à partir de deux individus parents. *NB : la littérature précise qu'il existe presque autant d'algorithmes de Cross-Over que d'implémentations d'algorithmes génétiques...*

Un exemple de crossover est illustré à la figure [2.2](#) page [4](#).

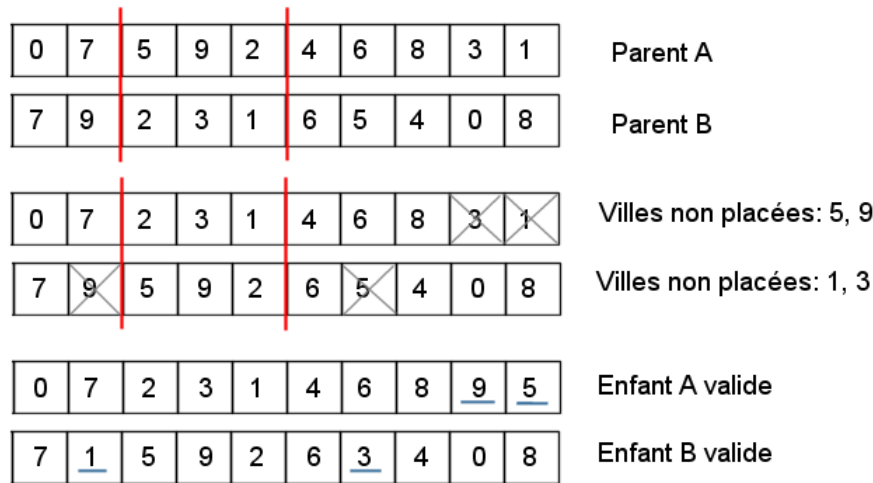


FIG. 2.1 – Représentation graphique d'un Cross Over

Nous allons ici décrire ici un des plus simples, le crossover à 2 points (c'est loin d'être le plus efficace mais il a le mérite d'être assez simple à coder).

Il s'agit de recopier dans le fils une partie du parent A, jusqu'à une première cassure, ensuite la partie correspondante du parent B jusqu'à la seconde cassure, et enfin la dernière partie du parent A. La difficulté réside dans le fait que les villes ne doivent pas être répétées. Si la ville a déjà été prise, on passe à la ville suivante dans l'ordre du parent B.

On détermine aléatoirement la cassure : ici entre le 2^{eme} et le 5^{eme} élément. Pour l'enfant A, le parent B n'est recopié qu'entre ces deux cassures, le parent A le reste du temps.

NB : Cet algorithme a un défaut pour le PVC, puisqu'il ne respecte pas toujours la règle suivante : si une arête (chemin entre 2 villes) se trouve dans les deux trajets parents, il faut qu'elle se trouve dans le trajet fils. Ce problème a été en partie résolu par le crossover OX (entres autres)

2.3 La mutation

Il s'agit d'une modification (plus ou moins aléatoire) du code génétique d'un individu. Cela permet de sortir des minimums locaux, grâce à une perturbation, un peu à la manière du recuit simulé. Le type de mutation implémenté est l'échange entre deux villes.

NB : Un autre choix plus efficace s'il est effectué plusieurs fois à la suite est le renversement de parcours entre deux villes

2.4 Les différents paramètres

2.4.1 Les paramètres de la population

- Probabilité de mutation : C'est la probabilité qu'un gène d'un individu de la population courante subisse une mutation aléatoire au cours d'une phase d'évolution.
- Probabilité de Cross-Over : C'est la probabilité d'effectuer un Cross Over aléatoire entre deux individus de la population courante au cours d'une phase d'évolution.
- Nombre d'individus de la population : ce nombre fixe le nombre d'individus générés aléatoirement au début pour former la population initiale. C'est aussi le nombre d'individus à la fin de chaque phase d'évolution. En effet, le nombre d'individus de la population est constant : il y a autant de morts que de naissances à chaque étape.

2.4.2 Les paramètres des conditions d'arrêt

- Stagnation de la valeur d'adaptation du meilleur individu : Si la valeur d'adaptation du meilleur individu de la population stagne pendant un nombre important de générations, on peut raisonnablement penser que l'algorithme ne trouvera pas de meilleure solution au problème. Le paramètre est donc le nombre de générations de stagnation de la meilleure valeur d'adaptation à partir duquel on doit interrompre la recherche.
- Nombre maximum de génération : on peut supposer que l'on dispose d'un temps pour trouver une solution, on peut donc limiter l'exécution de l'algorithme à un temps maximal (d'utilisation du processeur) après lequel l'algorithme arrête sa recherche.

Chapitre 3

Nos résultats

Notre programme fonctionne correctement et retourne des résultats plutôt satisfaisants.

Le meilleur résultat obtenu pour le tour de la France est de 5384 Km pour un taux de mutation de 13% et de 1% pour le cross over alors que le résultat optimal ¹ est de 4809 Km. Pour l'Europe, nous obtenons 38591 Km alors que le résultat optimal est de 27416 Km.

Ces résultats sont illustrés à la figure 1 page 7 par différentes valeurs de mutation et cross over.

Ces valeurs de taux de mutation et cross over peuvent surprendre si on les compare aux valeurs trouvées dans la "littérature". Le site de Supelec propose d'autres valeurs du taux de mutation :

- 0.001% : On observe qu'à partir d'un moment, tous les individus sont identiques puisque le taux de mutation est trop petit. Il est alors absurde de poursuivre le mécanisme car les probabilités d'apparition d'une meilleure solution sont très faibles. Ici, comme il y a eu trop peu d'apparition de nouveaux caractères génétiques par mutation le meilleur individu a fini par s'imposer dans la population en se reproduisant à outrance.
- 10% : La valeur est trop grande et la propagation des bonnes solutions par reproduction ne peut se faire puisque les individus perdent très vite leur patrimoine génétique par mutation.
- 1% est une valeur correcte. L'apport en patrimoine génétique nouveau est suffisant mais pas important au point de gréver les avantages des croisements.

¹Les résultats optimaux sont obtenus avec le logiciel Soplex

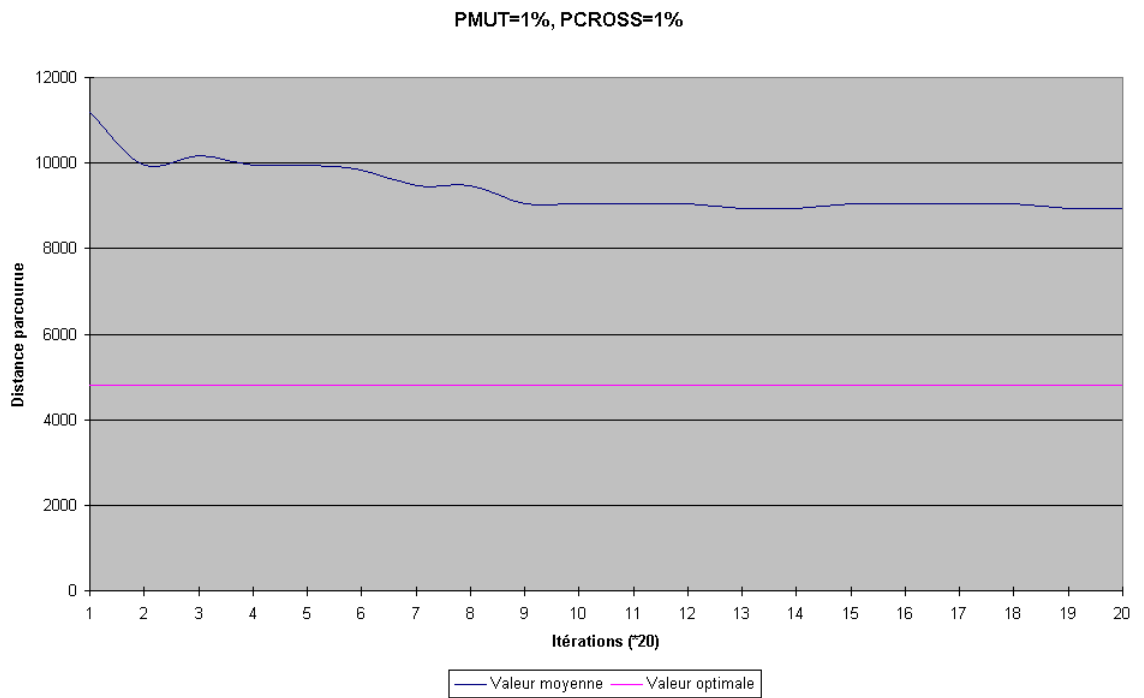


FIG. 3.1 – Evolution des résultats avec des petites valeurs

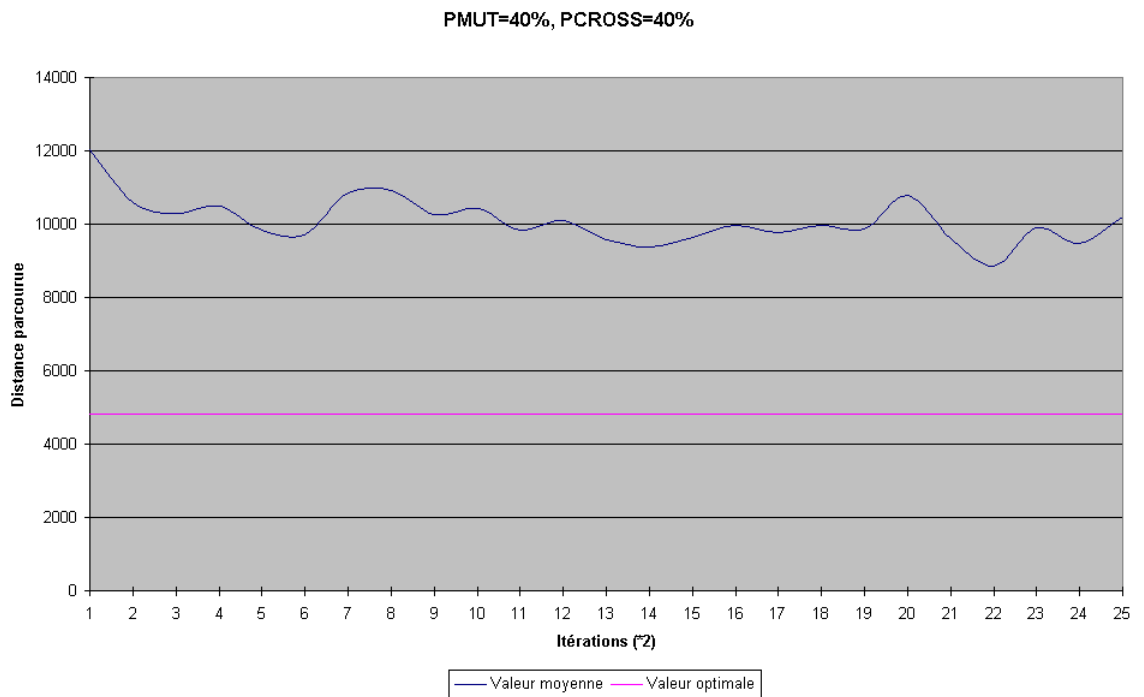


FIG. 3.2 – Evolution des résultats avec des grandes valeurs

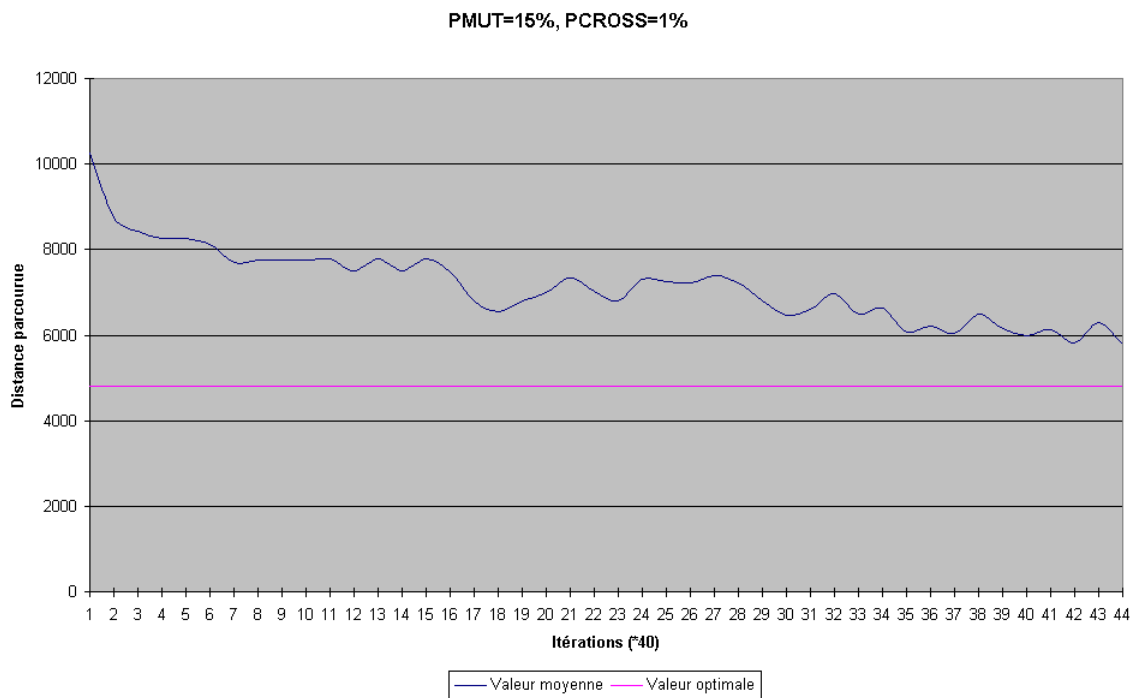


FIG. 3.3 – Evolution des résultats avec des valeurs correctes

Chapitre 4

Problèmes rencontrés

Ce projet a été développé en utilisant la version v3.3.2. de gcc (Mandrake 10.0). L'exécution du programme sur une machine de l'**UBO** génère l'erreur *Segmentation Fault*.

Chapitre 5

Optimisation possible

5.1 L'algorithme 2opt

On peut utiliser une méthode de recherche locale (2opt ou 3opt), appliquée à chaque individu lors de sa création. Le principe est le suivant :

- On génère un trajet initial aléatoirement.
- On applique à ce trajet une transformation simple (mutation par exemple).
- Si la transformation améliore le trajet, on garde cette transformation, sinon on l'annule.
- On répète l'opération jusqu'à ce que l'on ne puisse plus améliorer le trajet.

5.2 Optimisation de la rapidité

- Au lieu de calculer la longueur du trajet à chaque fois, il est plus rapide de calculer le *gain*, c'est à dire la différence de coût entre la solution initiale et la solution avec la portion de trajet retournée.
- Une autre possibilité est d'utiliser les architectures parallèles.

Annexe A

Fichier d'entête

```
#ifndef __DATA_H_
#define __DATA_H_

#define POPSIZE 100
#define TOWNS 24
#define PMUT 15
#define PCROSS 1
#define MAXI 15000
#define AMIN 100

struct individu {
    int codage[TOWNS];
    int fitness;
};

struct population {
    struct individu tabIndividu[POPSIZE];
    struct individu * best;
};

void initPop(struct population *pp, struct individu *pi);
void fitness(struct individu *pi);
void mutation(struct individu *pi);
void crossOver(struct individu *pi1, struct individu *pi2, struct individu *newI1,
    struct individu *newI2);
void correct(struct individu *pi1, struct individu *pi2);
void select(struct population *pp, struct population *new);
void readDistances(char *filename, int towns);
void bestPop(struct population *ptPop);
```

```
void affichP(struct population *ptPop);  
void affichI(struct individu *ptInd);  
  
#endif
```

Annexe B

Code de l'algorithme génétique

```
/*=====
  Nom          : Rémy Lalande, Jean-Marie Le Yaouanc, Yvette Le Bras
  -----
  Description : Fichier data.c
  =====*/

#include <stdio.h>
#include <time.h>
#include "data.h"

int matrice[TOWNS][TOWNS];

/*****
  Nom de la fonction : initPop
  Type               : void
  Parametres d'entree : struct population * pp, struct individu * pi
  -----
  Description : Initialise une population
  *****/
void initPop(struct population *pp, struct individu *pi){
  int num, j, k, l, a, b, c;
  int tmp=1;

  for(j=0; j< POPSIZE; j++){
    for(k=0; k< TOWNS; k++){
      while(tmp==1){
tmp=0;
```

```

num= rand()%TOWNS;
for(l=0; l< k; l++){
    if( pi->codage[l] == num) tmp=1;
}
if(tmp==0) pi->codage[k]= num;
    } tmp=1;
    } fitness(pi);

    for(c=0; c<TOWNS;c++){
        pp->tabIndividu[j].codage[c]=pi->codage[c];
    }
pp->tabIndividu[j].fitness= pi->fitness;
}
bestPop(pp);
}

/*****
Nom de la fonction : fitness
Type : void
Parametres d'entree : struct individu * pi
-----
Description : Calcule le fitness d'un individu
*****/
void fitness(struct individu *pi){
    int distance=0;
    int m, a, b;
    for(m=0; m<TOWNS; m++){
        a=pi->codage[m];
        b=pi->codage[m+1];
        distance= distance + matrice[a][b];
    }
    pi->fitness=distance;
}

/*****
Nom de la fonction : mutation
Type : void
Parametres d'entree : struct individu * pi
-----
Description : Effectue une mutation de 2 genes d'un individu
*****/
void mutation(struct individu *pi){

```

```

int ville1,ville2, tmp;
ville1= rand()%TOWNS;
ville2= rand()%TOWNS;
if(ville2==ville1){ville2=(ville2+1)%TOWNS;}
tmp= pi->codage[ville1];

pi->codage[ville1]= pi->codage[ville2];
pi->codage[ville2]= tmp;
}

/*****
Nom de la fonction : crossOver
Type : void
Parametres d'entree : struct individu * pi1, struct individu * pi2,
                    struct individu * newI1, struct individu * newI2
-----
Description : Operation principale de l'algorithme genetique. Utilise
              deux individus parents pour donner deux individus enfants
              en utilisant le cross over à deux points.
*****/
void crossOver(struct individu *pi1, struct individu *pi2, struct individu *
newI1, struct individu *newI2){

int coupure1, coupure2, tmpC, n, p, flag, erreur;

coupure1= rand()%TOWNS;
coupure2= rand()%TOWNS; if(coupure2==coupure1){coupure2=(coupure2+1)%TOWNS;}

//coupure1 < coupure2
if(coupure1 > coupure2){
    tmpC= coupure2;
    coupure2=coupure1;
    coupure1= tmpC;
}

for(n=0; n< coupure1; n++){
    newI1->codage[n]= pi1->codage[n];
    newI2->codage[n]= pi2->codage[n];
}

for(n=coupure1; n<coupure2; n++){
    newI2->codage[n]= pi1->codage[n];
    newI1->codage[n]= pi2->codage[n];
}

```

```

}

for(n=coupure2; n<TOWNS; n++){
    newI1->codage[n]= pi1->codage[n];
    newI2->codage[n]= pi2->codage[n];
}

for(n=0; n<TOWNS; n++){
    for(p=0; p<TOWNS; p++){
        if(newI1->codage[p]==n){
if(flag==1) {erreur=1;} else {flag=1;}
        }
    }
}
if(erreur==1) {correct(newI1, newI2);}
}

/*****
Nom de la fonction : correct
Type : void
Parametres d'entree : struct individu * pi1, struct individu * pi2
-----
Description : Corrige les enfants issus du CrossOver. Parcours
un tableau de flags et verifie s'il y a des doublons
parmi les genes. Si flag==0, une ville n'est pas là,
si flag==2, la ville est présente 2 fois. Remplace
la ville présente 2 fois par la ville absente.
*****/
void correct(struct individu *pi1, struct individu *pi2){
    int tabFlag1[TOWNS];
    int tabFlag2[TOWNS];
    int i,j,k,l;
    int flag0,flag2;
    int latent, var=1;

    for(i=0;i<TOWNS; i++){
        tabFlag1[i]=0; tabFlag2[i]=0;
    }

    for(j=0; j< TOWNS;j++){
        if(tabFlag1[pi1->codage[j]]== 0) {tabFlag1[pi1->codage[j]]=1;}
        else {tabFlag1[pi1->codage[j]]=2;}
    }
}

```

```

for(j=0; j< TOWNS;j++){
    if(tabFlag2[pi2->codage[j]]== 0) {tabFlag1[pi2->codage[j]]=1;}
    else {tabFlag2[pi2->codage[j]]=2;}
}

latent = 1; i=0; flag0=-1; flag2=-1;
while (latent) {
    while ((flag0 ==-1 || flag2 ==-1) && i<TOWNS){
        if (tabFlag1[i]==2 && flag2 ==-1){
flag2 = i;
        }

        if (tabFlag1[i]==0 && flag0 ==-1){
flag0 = i;
        }
        i++;
        if (i==TOWNS) {latent =0;}
    }

    while(i<TOWNS && var){
        if(pi1->codage[i] == flag2){
            pi1->codage[i] == flag0;
            var=0;
        } var=1;
    }
}
latent = 1; i=0; flag0=-1; flag2=-1;
while (latent) {
    while ((flag0 ==-1 || flag2 ==-1) && i<TOWNS){

if (tabFlag2[i]==2 && flag2 ==-1){
    flag2 = i;
}

if (tabFlag2[i]==0 && flag0 ==-1){
    flag0 = i;
}
i++;
if (i==TOWNS) {latent =0;}
    }

    while(i<TOWNS && var){
if(pi2->codage[i] == flag2){

```

```

    pi2->codage[i] == flag0;
    var=0;
}
var=1;
    }
}
}

/*****
Nom de la fonction : select
Type                : void
Parametres d'entree : struct individu * pp, struct individu * new
-----
Description : Parmi une population initiale, selectionne les meilleurs
              individus pour créer une seconde population pouvant
              contenir plusieurs fois le même individu.
*****/
void select(struct population *pp, struct population *new){
    int win, j;
    int tailleRoulette =0; int partRoulette =0; int selected=0; int i=0; int max=-1;

    for(i=0; i< POPSIZE; i++){
        tailleRoulette = tailleRoulette + MAXI - pp->tabIndividu[i].fitness ;
    }
    i =0 ;
    for(j=0; j<POPSIZE; j++){
        win= rand()%tailleRoulette;
        while (selected == 0){
            partRoulette = partRoulette + MAXI - pp->tabIndividu[i].fitness ;
            if(win < partRoulette){new->tabIndividu[j]= pp->tabIndividu[i]; selected=1;}
            i++;
        } partRoulette = 0; selected=0; i=0;
    }
}

/*****
Nom de la fonction : readDistances
Type                : void
Parametres d'entree : char *filename, int towns
-----
Description : Lit les valeurs d'un fichier
*****/
void readDistances(char *filename, int towns){
    int i, j;

```

```

    int distance;

FILE *fp = fopen(filename, "r");

for(j=1; j<towns; j++){
    for(i=0; i<j; i++) {
        fscanf(fp, "%d", &distance);
        matrice[i][j] = matrice[j][i] = distance;
    }
}
fclose(fp);
}

/*****
Nom de la fonction : bestPop
Type                : void
Parametres d'entree : struct population *ptPop
-----
Description : Permet d'obtenir le 'best' individu d'une population
*****/
void bestPop(struct population *ptPop){
    int i,f;
    f=MAXI;
    struct individu *ptInd;

    for(i=0; i<POPSIZE; i++){
        if((ptPop->tabIndividu[i].fitness)<f){
            ptInd= &(ptPop->tabIndividu[i]); f=ptInd->fitness;
        }
    }
    ptPop->best = ptInd;
}

/*****
Nom de la fonction : affichP
Type                : void
Parametres d'entree : struct population *ptPop
-----
Description : Affiche l'ensemble d'une population
*****/
void affichP(struct population *ptPop){
    int i,j;
    struct individu *ptInd;
    for(i=0; i<POPSIZE; i++){

```

```

    ptInd = &(ptPop->tabIndividu[i]);

    printf("Codage: ");
    for(j=0; j<TOWNS; j++){
        printf(" %d ", ptInd->codage[j]);
    }
    printf("\t Fitness=%d\n", ptInd->fitness);
}
printf("Best Individu=");
for(j=0; j<TOWNS; j++){
    printf(" %d ", ptPop->best->codage[j]);
}
printf("%d ", ptPop->best->fitness);
}

/*****
Nom de la fonction : affichI
Type                : void
Parametres d'entree : struct individu *ptInd
-----
Description : Affiche le codage d'un individu ainsi que sa fitness
*****/
void affichI(struct individu *ptInd){
    int i,j;
    printf("\n");
    printf("Codage: ");
    for(j=0; j<TOWNS; j++){
        printf(" %d ", ptInd->codage[j]);
    }
    printf("\t Fitness=%d\n", ptInd->fitness);
}

/*****
Nom de la fonction : main
Type                : int
Parametres d'entree : Aucun
-----
Description : Execute les fonctions
*****/
int main(){

    struct population *ptPop;
    struct population pop, pop2;

```

```

struct individu ind1, ind11, ind2;

int i,j,k,r,amelioration, tmpI1, tmpI2;
int c=0, cpt=0;
amelioration=MAXI;

srand(time(0));
printf("PMUT=%d, PCROSS=%d", PMUT, PCROSS);
printf("\nlecture\n");
readDistances("../Data/france.txt", 24);

ptPop=&pop;
printf("Init de la population\n");
initPop(ptPop, &ind1);
//affichP(ptPop);

while(clock()<(5*60) || c<30){
    //mutation
    printf("\n\t mutation");

    for (i=0; i<POPSIZE; i++){
        r=rand()%100; //%
        if(r<PMUT){
mutation(&(ptPop->tabIndividu[i]));
fitness(&(ptPop->tabIndividu[i]));
        }
    }

    affichP(ptPop);
    //crossOver
    printf("\n\t crossOver");
    while(r<PCROSS)
{
    tmpI1= rand()%POPSIZE;
    tmpI2= rand()%POPSIZE;
    if(tmpI2==tmpI1) tmpI2= (tmpI2+1)%POPSIZE;
    crossOver(&(ptPop->tabIndividu[tmpI1]), &(ptPop->tabIndividu[tmpI2]), &ind11, &ind2);

    ptPop->tabIndividu[tmpI1]=ind1; ptPop->tabIndividu[tmpI2]=ind2;
    fitness(&(ptPop->tabIndividu[tmpI1]));
    fitness(&(ptPop->tabIndividu[tmpI2]));
}

    affichP(ptPop);

```

```
        //selection
        printf("\n\t selection");

        select(ptPop, &pop2);
        bestPop(&pop2);
        amelioration=abs(pop.best->fitness - pop2.best->fitness);
        ((amelioration > 1000) ? (c++) : (c=0));

        pop=pop2;

        //Un affichage tous les 40 pas
        if(cpt%40==0){ affichP(ptPop);}
        cpt++;
        affichP(ptPop);
    }
}
```