Rapport Projet RN40 A22

JOVENIN Eileen

Professeur : A. KOUKAM

Objectif du projet :

Ce projet a pour objectif la définition et la manipulation de types abstraits de données Individu et Population. Un Individu est représenté par une suite de bits et une Population est une suite d'Individus.

Il s'agit d'une version simplifiée d'algorithmes génétiques, constituant une des approches de résolution de problèmes d'optimisation.

Table des matières :

Description des choix de conception et d'implémentation relatifs aux structures de données utilisées et à la démarche adoptée.

Comme précisé dans l'énoncé du projet, les types Individu et Population sont des types abstraits. Ainsi j'ai décidé de créer un fichier nommé structures.h afin de les regrouper. Dans celui-ci se trouvent les structures créées (ElemIndiv, Individu, ElemPop, Population) ainsi que l'instanciation de Bit qui est, comme demandé, un « unsigned char ».

ElemIndiv et ElemPop sont des structures « intermédiaires » utilisées dans le cadre de listes chaînées afin de leur assigner chacun leur valeur (respectivement les Bit et l'Individu) ainsi qu'un pointeur sur l'élément suivant, comme nous avons pu le voir lors de travaux dirigés.

Pour les fonctions, j'ai décidé de créer trois fichiers séparés : main.c , individu.c et population.c ainsi que leurs header pour les deux derniers. Les fonctions sont donc triées selon sur quel type elles affectent : le type Individu dans individu.c et le reste dans population.c.

Ainsi le main.c contient seulement les appels et affichages de fonctions afin que cela soit plus clair et organisé. De plus il y a un « do… while » afin de répéter le programme autant de fois que nécessaire, notamment pour comparer les résultats.

Des explications sont disponibles dans le code C afin de mieux comprendre certaines parties du code mais pas dans les algorithmes afin de ne pas répéter ces dernières. Les algorithmes ci-après sont donc constitués de leur profil respectifs (données, résultat, lexique) ainsi que l'algorithme en lui-même de chaque sous-programme.

Algorithmes des sous-programmes.

Individu.c

Initialiser aléatoirement la liste de bits (version itérative).

```
*initialize_people_ite : fonction itérative pour donner des bits à un individu
    Retourne : un Individu
    Paramètres : un entier longindiv
    Lexique : l et p sont des Individu permettant de parcourir les bits

    Début
        *l = malloc(sizeof(Individu))
        *p = malloc(sizeof(Individu))
        vide(premier élément de l)
        vide(premier élément de p)
        tant que (longindiv > 0) faire
            premier élément de l = malloc(sizeof(ElemIndiv))
            longIndiv de l = longindiv
            le bit du premier élément de l = rand()%2
            suivant du premier élément de l = le premier élément de p
            premier élément de p = premier élément de l
            longindiv --
        fin tant que
        *initialize_people_ite(longindiv) = p
    Fin
```

Initialiser aléatoirement la liste de bits (version récursive).

```
*initialize_people_rec : fonction récursive pour donner des bits à un individu
    Retourne : un Individu
    Paramètres : un entier longindiv
    Lexique : rand() est la fonction pour obtenir un chiffre aléatoire
            insert_head_people(Individu *people, Bit value) est la fonction qui insert en tête dans people le Bit value
            insert_tail_people(Individu *people, Bit value) est la fonction qui insert en queue dans people le Bit value

    Début
        si longindiv == 0 alors
            insert_tail_people(longindiv) = insert_head_people(NULL,rand() % 2)
        fin si
        insert_tail_people(longindiv) = insert_tail_people(initialize_people_rec(longindiv - 1),rand() % 2)
    Fin
```

Décoder la liste de bits et donner la valeur entière correspondante.

```
calculate_value : fonction qui calcule la valeur décimale des bits d'un Individu
    Retourne : un entier
    Paramètres : un Individu *people
    Lexique : *actuel est un ElemIndiv
            pow(X,Y) est la fonction qui retourne Y à la puissance X
            value et i sont des entiers

    Début
        value = 0
        actuel = premier élément de people
        pour i allant de longIndiv de people à i > 0 avec un pas de -1 faire
            value = bit de actuel * pow(2,i-1) + value
            actuel = suivant de actuel
        fin pour
        calculate_value(*people) = value
    Fin
```

Calculer à partir de la valeur d'un individu sa qualité, en utilisant la fonction réelle f1

```
calculate_quality : fonction qui calcule la qualité d'un Individu
    Retourne : un nombre décimal quality
    Paramètres : un Individu *people
    Lexique : pow(X,Y) est la fonction qui retourne Y à la puissance X
              X est un nombre décimal utilisé pour calculer la qualité
              quality est un nombre décimal qui est la qualité à retourner
              valueA est une constante définie au préalable, égale à -1 pour f1
              valueB est une constante définie au préalable, égale à 1 pour f1

    Début
        si vide(people) alors
            X = valeur de people / longIndiv de people^2 * (valueB - valueA) + valueA
            quality = -X^2
            calculate_quality(*people) = quality
        sinon
            calculate_quality(*people) = 0
        fin si
    Fin
```

Autres fonctions « intermédiaires »

```
*insert_head_people : fonction qui insert en tête dans people le Bit value
    Retourne : un Individu
    Paramètres : un Individu *people et le Bit value
    Lexique : element est un ElemIndiv

    Début
        si vide(people) alors
            people = malloc(sizeof(Individu))
            vide(premier élément de people)
            longIndiv de people = 0
        sinon
            si (vide(premier élément de people)) alors
                *element = malloc(sizeof(ElemIndiv))
                le bit de element = value
                vide(suivant de element)
                le premier element de people = element
                longIndiv de people = 1
            sinon
                *element = malloc(sizeof(ElemIndiv))
                le bit de element = value
                suivant de element = premier élément de people
                le premier element de people = element
                longIndiv de people ++
            fin si
        fin si
        *insert_head_people(*people, value) = people
    Fin

*insert_tail_people : fonction qui insert en queue dans people le Bit value
    Retourne : un Individu
    Paramètres : un Individu *people et le Bit value
    Lexique : element et check sont des ElemIndiv

    Début
        si vide(people) alors
            people = malloc(sizeof(Individu))
            vide(premier élément de people)
            longIndiv de people = 0
        sinon
            si (vide(premier élément de people)) alors
                *element = malloc(sizeof(ElemIndiv))
                le bit de element = value
                vide(suivant de element)
                le premier element de people = element
                longIndiv de people = 1
            sinon
                *check = premier élément de people
                tant que !vide(suivant de check) faire
                    check = suivant de check
                fin tant que
                *element = malloc(sizeof(ElemIndiv))
                bit de element = value
                vide(suivant de element)
                suivant de check = element
                longIndiv de people ++
            fin si
        fin si
        *insert_tail_people(*people, value) = people
    Fin
```

```
*insert_tail_pop : fonction qui insert en queue dans pop l'Individu people
    Retourne : une Population
    Paramètres : une Population *pop et un Individu *people
    Lexique : element et check sont des ElemPop

    Début
        si vide(pop) alors
            pop = malloc(sizeof(Population))
            vide(premier élément de pop)
            popSize de people = 0
        sinon
            si (vide(premier élément de pop)) alors
                    *element = malloc(sizeof(ElemPop))
                    people de element = people
                    vide(suivant de element)
                    le premier element de pop = element
                    popSize de people = 1
            sinon
                *check = premier élément de pop
                tant que !vide(suivant de check) faire
                    check = suivant de check
                fin tant que
                *element = malloc(sizeof(ElemPop))
                people de element = people
                vide(suivant de element)
                suivant de check = element
                popSize de pop ++
            fin si
        fin si
        *insert_tail_pop(*pop, *people) = pop
    Fin
```

```
display_people : affiche un individu sous forme de bits
    Paramètres : un Individu individu
    Lexique : elem est un elemIndiv

    Début
        elem = tete(individu)
        tant que !vide(elem) faire
            afficher le bit de elem
            elem = suivant(elem)
        fin tant que
    Fin
```

Population.c

Initialiser de manière aléatoire la liste d'Individus.

```
*initialize_population : fonction qui permet d'initialiser une Population
    Retourne : une Population
    Paramètres : des entiers longIndiv et popSize
    Lexique : *pop est une Population
              *elem est un ElemPop

    Début
        *pop = malloc(sizeof(Population))
        popSize de pop = popSize
        premier élément de pop = malloc(sizeof(ElemPop))
        *elem = premier élément de pop
        people de elem = initialize_people_rec(longIndiv)
        value de people de elem = calculate_value(people de elem)
        quality de people de elem = calculate_quality(people de elem
        pour i de 1 à popSize avec un pas de 1 faire
            suivant de elem = malloc(sizeof(ElemPop))
            elem = suivant de elem
            people de elem = initialize_people_rec(longIndiv)
            value de people de elem = calculate_value(people de elem)
            quality de people de elem = calculate_quality(people de elem)
        fin pour
        *initialize_population(longIndiv, popSize) = pop
    Fin
```

Trier la liste par Qualité décroissante des Individus au moyen de Quicksort.

```
*quicksort : fonction qui permet de trier une Population
   Retourne : une Population
   Paramètres : une Population *pop
   Lexique : popSize et supp sont des entiers
             *elem et *compare sont des ElemPop, le 1er créé pour garder en mémoire le meilleur et le 2e créé pour parcourir la Population
             *best et *worst sont des Population
             quality1 et quality2 sont des nombres décimaux qui sont les qualités à comparer

   Début
       si popsize de pop == 1 OU vide(pop) OU vide(premier élément de pop) alors
           *quicksort(*pop) = pop
       fin si
       popSize = popSize de pop
       *elem = premier élément de pop
       *best = malloc(sizeof(Population))
       *worst = malloc(sizeof(Population))
       popSize de best = 0
       vide(premier élément de best)
       popSize de worst = 0
       vide(premier élément de worst)
       tant que !vide(elem) faire
           supp = 0
           *compare = premier élément de pop
           tant que !vide(compare) faire
               quality1 = quality de people de elem
               quality2 = quality de people de compare
               si quality1 >= quality2 alors
                   supp++
               fin si
               compare = suivant de compare
           fin tant que
           si supp >= popSize / 2 ET popSize de best <= popSize / 2 alors
               best = insert_tail_pop(best,people de elem)
           sinon
               worst = insert_tail_pop(worst, people de elem)
           fin si
           elem = suivant de elem
       fin tant que
       si popSize de best == 0 alors
           *quicksort(*pop) = worst
       sinon
           si popSize de worst == 0
               *quicksort(*pop) = best
           sinon
               *quicksort(*pop) = merge_pop(quicksort(best),quicksort(worst))
           fin si
       fin si
   Fin
```

Sélectionner les meilleurs Individus de la Population en tronquant la liste et en la complétant par recopie des selection premiers éléments.

```
*tSelect : fonction qui permet de sélectionner les meilleurs Individus de la Population
    Retourne : une Population
    Paramètres : une Population *pop et un entier selection
    Lexique : *selected est une Population qui correspond aux individus sélectionnés
              *elem et *forReplace sont des ElemPop

    Début
        *selected = malloc(sizeof(Population))
        vide(premier élément de selected)
        popSize de selected = 0
        *elem = premier élément de pop
        pour i allant de 0 à selection avec un pas de 1 faire
            insert_tail_pop(selected, people de elem)
            elem = suivant de elem
        fin pour
        *forReplace = premier élément de selected
        tant que !vide(elem) faire
            si vide(forReplace) alors
                forReplace = premier élément de selected
            fin si
            people de elem = people de forReplace
            elem = suivant de elem
            forReplace = suivant de forReplace
        fin tant que
        *tSelect(*pop, selction) = pop
    Fin
```

Croiser la Population, c'est à dire à partir d'une Population P1, créer une seconde Population P2, constituée d'Individus sélectionnés aléatoirement deux à deux dans P1 et croisés entre eux.

```
*cross_population : fonction qui permet de croiser deux Individu
    Retourne : une Population
    Paramètres : une Population *pop et un nombre décimal pCroise
    Lexique : *P1, *P2 et *P3 sont des Individu
              *newPop est la nouvelle Population créée
              *bitP1 et *bitP2 sont des ElemIndiv
              getRandomPeople(pop) permet de sélectionner un Individu aléatoirement dans une Population pop
              insert_tail_people(Individu *people, Bit value) est la fonction qui insert en queue dans people le Bit value
              calculate_value(P3) permet de calculer la valeur décimale des bits d'un Individu P3
              calculate_quality(P3) permet de calculer la qualité d'un Individu P3

    Début
        *newPop = malloc(sizeof(Population))
        popSize de newPop = 0
        vide(premier élément de newPop)
        tant que popSize de newPop < popSize de pop faire
            *P3 = malloc(sizeof(Individu))
            vide(premier élément de P3)
            longIndiv de P3 = 0
            P1 = getRandomPeople(pop)
            répéter
                P2 = getRandomPeople(pop)
            tant que P1==P2
            bitP1 = premier élément de P1
            bitP2 = premier élément de P2
            tant que !vide(bitP1) et !vide(bitP2) faire
                si rand()%2 < pCroise alors
                    insert_tail_people(P3,bitP1->bit)
                sinon
                    insert_tail_people(P3,bitP2->bit)
                fin si
                bitP1 = suivant de bitP1
                bitP2 = suivant de bitP2
            fin tant que
            value de P3 = calculate_value(P3)
            quality de P3 = calculate_quality(P3)
            insert_tail_pop(newPop,P3)
        fin tant que
        *cross_population(*pop, pCroise) = newPop
    Fin
```

Autres fonctions « intermédiaires »

```
*getBestPeople : fonction qui permet de déterminer le meilleur Individu d'une Population
    Retourne : un Individu
    Paramètres : une Population *pop
    Lexique : *best et *elem sont des ElemPop

    Début
        best = premier élément de pop
        elem = premier élément de pop
        tant que !vide(elem) faire
            si quality de people de elem < quality de people de best faire
                best = elem
            fin si
            elem = suivant de elem
        fin tant que
        *getBestPeople(*pop) = people de best
    Fin

*getRandomPeople : fonction qui permet de sélectionner un Individu aléatoirement dans une Population
    Retourne : un Individu
    Paramètres : une Population *pop
    Lexique : *elem est un ElemPop
            random est un entier

    Début
        elem = premier élément de pop
        random = rand()% (popSize de pop)
        Pour i allant de 0 à random avec un pas de 1 faire
            elem = suivant de elem
        fin pour
        *getRandomPeople(*pop) = people de elem
    Fin
```

```
*merge_pop : fonction qui permet de fusionner deux Population
    Retourne : une Population
    Paramètres : des Population *big et *small
    Lexique : *smallElem est un ElemPop

    Début
        si vide(big) faire
            big = malloc(sizeof(Population))
            vide(premier élément de big)
            popSize de big = 0
        fin si
        tant que !vide(smallElem) faire
            big = insert_tail_pop(big, people de smallElem)
            smallElem = suivant de smallElem
        fin tant que
        *merge_pop(*big, *small) = big
    Fin
```

Jeux d'essais.

Fonction F1

1er test

```
Bienvenue dans une simplification d'algorithmes genetiques !

Nombre de bit.s par individu : 8
Probabilite de croisement : 0.500
Taille population : 134
Nombre de selection.s : 37
Nombre de generation.s : 185

Le meilleur individu a pour liste de bits : 10000000
Soit 128 en decimal
Et pour qualite -0.00000000
```

3e test

```
Bienvenue dans une simplification d'algorithmes genetiques !

Nombre de bit.s par individu : 8
Probabilite de croisement : 0.500
Taille population : 55
Nombre de selection.s : 11
Nombre de generation.s : 146

Le meilleur individu a pour liste de bits : 01111111
Soit 127 en decimal
Et pour qualite -0.00006104
```

4e test

```
Bienvenue dans une simplification d'algorithmes genetiques !

Nombre de bit.s par individu : 8
Probabilite de croisement : 0.500
Taille population : 159
Nombre de selection.s : 34
Nombre de generation.s : 186

Le meilleur individu a pour liste de bits : 01111101
Soit 125 en decimal
Et pour qualite -0.00054932
```

7e test

```
Taille population : 30
Nombre de selection.s : 3
Nombre de generation.s : 77

Le meilleur individu a pour liste de bits : 10010000
Soit 144 en decimal
Et pour qualite -0.01562500
```

Fonction F2

1er test

```
Bienvenue dans une simplification d'algorithmes genetiques !

Nombre de bit.s par individu : 16
Probabilite de croisement : 0.500
Taille population : 46
Nombre de selection.s : 10
Nombre de generation.s : 45

Le meilleur individu a pour liste de bits : 0000000010011011
Soit 155 en decimal
Et pour qualite 2.19293237
```

2e test

```
Nombre de bit.s par individu : 16
Probabilite de croisement : 0.500
Taille population : 150
Nombre de selection.s : 120
Nombre de generation.s : 22

Le meilleur individu a pour liste de bits : 0000000000000000
Soit 0 en decimal
Et pour qualite 2.30258512
```

4e test

```
Taille population : 141
Nombre de selection.s : 31
Nombre de generation.s : 48

Le meilleur individu a pour liste de bits : 0000000000000001
Soit 1 en decimal
Et pour qualite 2.30183768
```

Commentaires sur les résultats et conclusion.

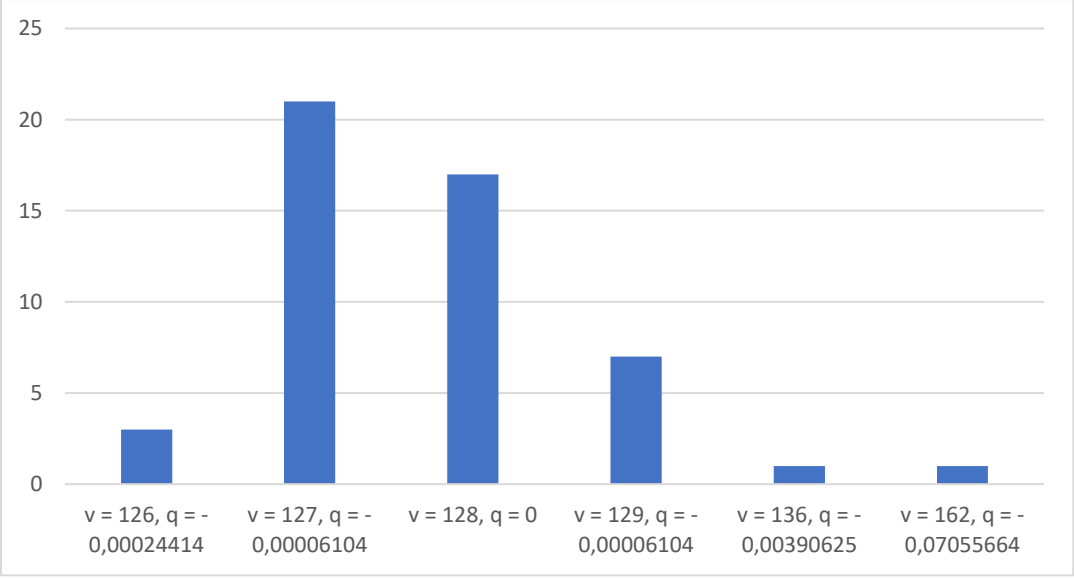
Ainsi grâce aux captures d'écran précédentes, nous pouvons voir qu'avec la fonction F1, la qualité est un nombre négatif, dont le but est de se rapprocher de 0. De plus, elle est atteinte lorsque l'individu a pour valeur décimale 128.

Quant à la fonction F2, il s'agit de qualités positives, dont le maximum est atteint avec un individu de valeur 0 et de qualité d'environ 2,30.

Lorsque l'on répète l'algorithme un grand nombre de fois, on se rend compte que le meilleur individu est souvent l'individu avec la qualité maximale, énoncé précédemment. Et lorsque cela n'est pas le cas, alors il s'agit d'un individu ayant une valeur proche du meilleur.

Cependant il arrive que la qualité ne soit pas très optimale, comme par exemple le 7^{e} test avec la fonction F1, mais cela reste rare puisque chaque paramètre (taille de population, nombre de sélections et nombre de générations) est proportionnel à la qualité du meilleur individu. Ainsi pour avoir une qualité médiocre, il faut une petite population, peu de sélections et peu de générations, ce qui est donc peu probable au vue de l'aléatoire sur ces dernières.

Voici un exemple avec une répétition de 50 fois de l'algorithme avec la fonction F1.



Ainsi nous pouvons constater que le meilleur individu est donc très souvent avec une qualité parfaite (individu 128) ou presque (individus 127 et 129) et que l'on a seulement 5 cas où l'individu a une qualité plus faible (126, 136 et 162) ce qui représente donc seulement 10% des répétitions.

Pour conclure, ce programme est un bon condensé de ce que nous avons pu voir en cours tout au long du semestre. Certaines parties furent plus complexes que d'autres (quicksort par exemple) mais globalement je l'ai trouvé très abordable sachant qu'il s'agit d'algorithme génétique, ce qui reste une notion complexe.