MIRA **SIRVENT**

Compte-rendu de projet- LO21

Choix de conception

```c
typedef unsigned char Bit; // définition d'un bit compris entre 0 et 1

typedef struct Elemb
{
    Bit value;
    struct Elemb* next;
} ElemBit; // Element de la liste de bits

typedef struct ListeBit
{
    ElemBit* head;
    int longIndiv; // Taille de la liste de bits
} Individu; // La liste de bits
```

```c
    Individu* value;
    struct Elemi* precedent;
    struct Elemi* next;
}ElemIndiv; // Element de la liste d'individu

typedef struct ListeIndiv
{
    ElemIndiv* head;
    ElemIndiv* tail;
    int taillePop; // Taille de la liste d'indiv
}Population;  // Une population est une liste d'individu
```

Nous avons décidé de créer une structure « ElemBit » et « Individu ». En temps normal, la structure « ElemBit » aurait suffi afin de pouvoir manipuler des Individus. Cependant, nous avions besoin d'intégrer le paramètre « longIndiv » (un entier) afin de pouvoir rendre la taille des Individus variable, d'où l'existence de la structure « Individu ».

La même démarche s'est appliquée à « ElemIndiv » et « Population » avec en plus l'ajout d'un pointeur vers l'élément précédent de la liste qui permettra de rendre les manipulations plus faciles (notamment avec le QuickSort)

Algorithmes des sous-programmes

Fonction initialisation (ITERATIVE)

Lexique	
Données	entier i, Individu l
Résultat	Individu l initialisé avec des bits aléatoires

Individu Initialisation (Individu l)

Pour i de 0 à valeur_longIndiv(l) de 1 en 1

ajouter_tete (l, entier aléatoire entre 0 et 1)

Fin Pour

Initialisation (Individu l)←l

Fin

Fonction initialisation (RECURSIVE)

Lexique	
Données	entier i, Individu l, entier longIndiv, Elembit p
Résultat	Individu l initialisé avec des bits aléatoires

Individu initialisation (Individu l, entier longIndiv)

Si (longIndiv > 0) Alors

p ← tete(l)

longIndiv ← longIndiv – 1

l ← ajouter_tete (l, entier aléatoire entre 0 et 1)

p ← suivant(p)

initialisation (l, longIndiv)

Fin Si

initialisation (Individu l, entier longIndiv)←l

Fin

Fonction décodage

Lexique	
Données	Element temp, entier i, entier LONGINDIV
Résultat	Entier valIndiv

```
Entier decodage (Individu I)
|   valIndiv ← 0
|   i ← LONGINDIV - 1
|   temp = tete(I)
|   Tant que(suivant(temp) != INDEFINI)
|   |   valIndiv ← valIndiv + (valeur(temp))*pow(2,i)
|   |   i←i-1
|   |   temp ← suivant(temp)
|   Fait
|   decodage (Individu I)← valIndiv
Fin
```

Fonction Initialisation de Population :

Lexique	
Données	Individu indiv, entier i, entier LONGINDIV
Résultat	Population Pop

```
Population init_pop(Population pop, Entier taillePop)
Début
|       Pour i allant de  0 à taillePop
|       |       Indiv ← nouveau(Individu)
|       |       Indiv←init_indiv(Indiv, LONGINDIV)
|       |       ajouterT(pop, indiv)
|       fait
Population init_pop(Population pop, Entier taillePop)←pop
FIN
```

Sélectionner population :

Lexique	
Données	elemPop elemCopy, entier i, entier LONGINDIV
Résultat	Population Pop

```
Population select_pop(Population Pop, Entier tSelect)
Début
    Si (non(vide(Pop)))
        elemCopy ←tete(Pop)
        Pour i de 0 à TAILLEPOP -tselect
            supprimerQ(pop)
        fait
        pour i de 0 à TAILLEPOP - tselect
            ajouterQ(pop, value(elemCopy))
            elemcopy←suivant(elemcCopy)
        fait
    Fin Si
Population select_pop(Population pop, Entier tSelect)←Pop
Fin
```

Trier Population :

Lexique	
Données	elemPop droite, elemPop gauche, Réel pivot
Résultat	Population Pop

```
Trier_pop(Population Pop, elemPop début, elemPop fin)
    Si (non(vide(pop)))
        Si(début != fin)
            Pivot ← qualité(début)
            Gauche ← début
            Droite ← fin
```

```
                    Tantque( droite != gauche)
                        tantque (qualité(Droite) < pivot)
                            droite← prec(droite)
                        fait
                        tantque (qualité(gauche) > pivot)
                            gauche← suivant(gauche)
                        fait
                        si(qualité(droite) = qualité(gauche) & droite != gauche)
                            gauche = suiv(gauche)
                        sinon(   si(qualité(droite) > qualité(gauche)))
                            echange(value(droite),value(gauche))
                            fin Si
                        fin Si
                        Si(gauche != début)
                            Pop← Trier_pop(Pop, début, prec(gauche))
                        Fin Si
                        Si(droite != fin)
                            Pop← Trier_pop(Pop, suiv(droite),fin)
                        Fin Si
                    Fait
                Fin Si
            Fin Si
Trier_pop(Population Pop, elemPop début, elemPop fin)←Pop
Fin
```

Fonction Croiser population

Lexique	
Données	elemPop elemCroise, Entier i, Entier j, Entier rnd1, Entier rnd2, individu indiv1, individu indiv2, Population P2
Résultat	Population P2

```
Population croiser pop(population P1)

Début

Si(non(vide(P1)))
    Pour i de 0 à TAILLEPOP/2
        Rnd1← alea(1, TAILLEPOP)
        elemCroise←tete(P1)
        pour j de 0 à rnd1
            elemCroise←suiv(elemCroise)
        fait
        indiv1←value(elemCroise)

        faire
            rnd2 ← alea(1, TAILLEPOP)
        tanque(rnd1 = rnd2)
        elemCroise←tete(P1)
        pour j de 0 à rnd2
            elemCroise←suiv(elemCroise)
        fait
        indiv2←value(elemCroise)

        croiser_indiv(indiv1, indiv2)
        ajouterT(P2, indiv1)
        ajouterT(P2, indiv2)
    fait
    si(impair(TAILLEPOP))
        Rnd1← alea(1, TAILLEPOP)
```

```
        elemCroise←tete(P1)
        pour j de 0 à rnd1
            elemCroise←suiv(elemCroise)
        fait
        indiv1←value(elemCroise)

        faire
            rnd2 ← alea(1, TAILLEPOP)
        tanque(rnd1 = rnd2)
        elemCroise←tete(P1)
        pour j de 0 à rnd2
            elemCroise←suiv(elemCroise)
        fait
        indiv2←value(elemCroise)

        croiser_indiv(indiv1, indiv2)
        ajouterT(P2, indiv1)
    fin SI
fin Si
Population croiser pop(population P1)←P2
Fin
```

Jeux d'essais et commentaires

Une fois le programme finalisé, nous retrouvons un résultat comme celui-ci-dessous ;

```
Le meilleur individu de cette population est :
[ 0  1  1  1  1  1  1  1 ]


Qualite : -0.000244
```

Nous allons donc tester maintenant les résultats qu'on trouve en utilisant les différentes fonctions qu'on a écrite (les valeurs décimales étant la qualité du meilleur individu)

Fonction 1 : TaillePopulation = 20/%deSelection = 10/Ngenerations = 20/LongIndiv = 8

N° Essais	1	2	3	4	5	6	7	8	9	10
Qualité	-0,000244	0	-0,000244	-0,019775	-0,002197	-0,011963	-0,019775	-0,000244	0	-0,002197

TaillePopulation = 20/%deSelection = 90/Ngenerations = 200/LongIndiv = 8

N° Essais	1	2	3	4	5	6	7	8	9	10
Qualité	0	0	0	0	0	0	0	0	0	0

Fonction 2 : TaillePopulation = 20/%deSelection = 10/Ngenerations = 20/LongIndiv = 16

N° Essais	1	2	3	4	5	6	7	8	9	10
Qualité	1,866072	1,630393	2,198981	1,705691	2,207103	1,848826	2,252993	0,893452	2,100212	2,080858

TaillePopulation = 200/%deSelection = 90/Ngenerations = 200/LongIndiv = 16

N° Essais	1	2	3	4	5	6	7	8	9	10
Qualité	2,302585	2,301091	2,302585	2,302585	2,302585	2,302585	2,302585	2,302585	2,302585	2,302585

Fonction 3 : TaillePopulation = 20/%deSelection = 10/Ngenerations = 20/LongIndiv = 32

N° Essais	1	2	3	4	5	6	7	8	9	10
Qualité	0,994792	0,706001	1	1	0,977654	0,998751	0,999999	0,999998	0,999998	0,999978

TaillePopulation = 200/%deSelection = 90/Ngenerations = 200/LongIndiv = 32

N° Essais	1	2	3	4	5	6	7	8	9	10
Qualité	1	1	1	1	1	1	1	1	1	1

Pour la fonction 1, on observe que plus on augmente le nombre de répétitions et la dureté de la sélection, plus la qualité du meilleur individu se rapproche de 0. Et cela fait sens car la fonction f1 étant $f1(X) = -X^2$ donc un nombre qui sera toujours inférieur ou égal à 0, le résultat le plus grand sera forcément 0. Notre programme semble donc efficace.

Pour la fonction 2, les résultats tendent vers une qualité de 2,3025825 lorsque que l'on cherche les meilleurs individus avec les paramètres optimaux. Le minimum de cette fonction étant atteint lorsque $x = 0$ et donc $X = 0.1$ est $-ln(0.1) = 2.30259$. Les résultats obtenus semblent cohérents étant donné que la probabilité d'obtenir un individu "0000000000000000" est très faible. En effet si l'on a pu obtenir des résultats semblables au maximum pour f1, c'est à cause de la longueur des individus qui était bien plus faible et donc permettait une possibilité d'obtention de l'individu "00000000" beaucoup plus importante.

Pour la fonction 3, les résultats tendent tous vers 1 avec des paramètres permettant de trouver un individu optimal. Cela s'explique car la plus grande valeur que peut atteindre la fonction f3 est atteinte en $X = -\pi[2\pi]$ et est donc $f3(-\pi) = 1$. Ce qui explique que ce résultat soit obtenu beaucoup plus souvent c'est le modulo 2π qui permet d'avoir plusieurs maximums, ce qui est une propriété de la fonction cosinus alors que le logarithme népérien lui n'a qu'une limite qui n'est jamais atteinte.

Conclusion :

On peut donc en conclure que notre programme fonctionne correctement et permet effectivement l'obtention d'une population avec des individus d'une qualité de plus en plus proche de leur maximal si les paramètres choisis permettent un grand nombre de répétition, d'une sélection importante ainsi qu'un nombre d'individus élevé. La longueur des individus joue également beaucoup sur la probabilité d'obtenir le meilleur individu possible, en effet plus un individu est complexe plus il est rare d'obtenir l'individu idéal que l'on recherche et le dernier facteur est bien évidemment la fonction qui détermine la qualité d'un individu, si elle a plusieurs antécédents à ses valeurs, il sera plus facile d'obtenir un individu avec une bonne qualité et inversement. Cela permet donc de mettre en avant des effets réels que l'on peut constater en génétique.