Galluet Antoine

de Villeneuve la Colette Arnaud

**LO44**

**Projet d'algorithmique et programmation**

INFO 1

Automne 2012

**Sommaire**

Introduction 2

Structures de Données : 3

ListeBit : 3

Population : 3

Opérations élémentaires sur les structures de données : 4

ListeBit : 4

PrintList: 4

PrintPopQualite: 5

Ajouter\_Queue\_Pop: 5

DernierPop : 6

Avant\_Dernier\_Pop: 6

Foncpop : 7

Initialisation et Selection : 7

InitPopRecur 7

InitPopIter 7

PickIP 7

SelectP 8

coller\_liste 8

Croiser Population : 8

CroiserP 8

Fonctions necessaire au Quicksort : Qualite, Recopier, Coller\_Liste : 9

QualiteP 9

RecopierP 9

Quicksort : 10

Cas trivial: 10

Cas général: 10

Algorithme: 11

Difficultés rencontrées lors de la réalisation du projet : 12

Réalisation du MakeFile : 12

Comprehension du Quicksort : 12

Algorithme du tSelect et CroiserP : 12

Utilisation de srand : 12

Jeu de test du logiciel : 13

Tests sur le paramètre "tSelect": 13

Tests sur le paramètre "TaillePop": 13

Conclusion 15

# Introduction

Dans le cadre de notre projet de programmation et d’algorithmique nous avions à réaliser un programme. L’objectif de ce programme étant de créer une population d’individu générés aléatoirement, de les croiser et de les sélectionner selon leur qualité sur plusieurs générations afin d’obtenir le meilleur individu possible.

Pour la création de ce programme nous avons procédé par étape en découpant le problème ; dans un premier temps nous nous sommes intéressés à toutes les fonctions liées aux individus puis à celles liées aux populations et pour finir à la structure du programme et son interface.

Nous verrons donc d’abord la logique derrière notre programme puis les difficultés rencontrées et finalement les différents jeux d’essais.

# Structures de Données :

## ListeBit :

Nous avons définis le type de donnée ListeBit (ou individu) selon l'énoncé sous forme de liste chaînée.

typedef struct elem{

int valeur;

struct elem \* suivant;

}Bit;

typedef Bit \* ListeBit;

## Population :

Pour la structure de donnée Population nous l'avons créé sous forme de liste chaînée d'individus avec pour chaque individu sa qualité. Garder en mémoire la qualité de chaque individu nous permet de ne pas avoir à la recalculer à chaque étape, ceci accélérant le traitement de l'information.

typedef struct elemp{

ListeBit Indiv;

float qualite;

struct elemp \* suivant;

}element;

typedef element \* Population;

# Opérations élémentaires sur les structures de données :

## ListeBit :

Ajouter\_Queue\_Indiv:

Ajouter\_Queue\_Indiv : ListeBit A \* entier v -> ListeBit

Résultat: À la fin de la liste de bits A on a rajouté le bit de valeur v

Index: newel (bit que l'on va rajouter), p (liste utilisée pour parcourir A)

newel <- CreerI()

newel->valeur <- v

Si Vide(A) Alors

A <- newel

Sinon

p <- A

Tant que Vide(Reste(p)) Faire

p <- Reste(p)

Fait

p -> suivant <- newel

Fin Si

### PrintList:

Résultat: Imprime les valeurs d'une liste de bits en ligne

Index: A (liste de bits à imprimer), I (liste utilisée pour parcourir A)

Si Vide(A) Alors

Imprimer("La liste est vide")

Sinon

I <- A

Tant que NonVide(I) Faire

Imprimer (Valeur\_Tete(I))

I <- Reste(I)

Fait

Fin SI

Population :

PrintPop:

Résultat: imprime les valeurs des individus d'une liste d'individus en colonne

Index: A (population à imprimer)

Si Vide(A) Alors

Imprimer("La population est vide")

Sinon

Tant que NonVide(A) Faire

PrintList(A -> Indiv)

A <- Reste(A)

Fait

Fin SI

### PrintPopQualite:

Résultat: Imprime les valeurs des individus et leur qualité d'une liste d'individus en colonne

Index: A (population à imprimer)

Si Vide(A) Alors

Imprimer("La population est vide")

Sinon

Tant que NonVide(A) Faire

PrintList(A -> Indiv)

Imprimer(A -> qualité)

A <- Reste(A)

Fait

Fin SI

### Ajouter\_Queue\_Pop:

Résultat: Ajoute à une population 'A' un individu supplémentaire 'indiv' en fin de liste

Index: A (population à laquelle on va rajouter un individu), indiv (liste de bits que l'on va rajouter en queue de A), LongIndiv (entier donnant la longueur des individus utilisé pour calculer leur qualité)

newel <- CreerP()

newel -> Indiv <- indiv

newel -> qualite <- QualiteI(indiv, LongIndiv)

Si Vide(A) Alors

A <- newel

Sinon

p <- A

Tant que Vide(Reste(p)) Faire

p <- Reste(p)

Fait

p -> suivant <- newel

Fin Si

### DernierPop :

DernierPop: Population P -> Population

Algorithme récursif

Résultat: Parcours une liste d'individus et en renvoie le dernier

Index: P (population à parcourir)

Si VideP(P) Ou VideP(ResteP(P)) Alors

Retourner P

Sinon

DernierPop <- DernierPop(ResteP(P))

Retourner P

Fin Si

### Avant\_Dernier\_Pop:

Avant\_Dernier\_Pop: Population P -> Population

Algorithme récursif

Résultat: Parcours une liste d'individus et en renvoie l'avant-dernier

Index: P (population à parcourir)

Si VideP(P) Ou VideP(ResteP(P)) Ou VideP(ResteP(ResteP(P))) Alors

Retourner P

Sinon

DernierPop <- DernierPop(ResteP(P))

Retourner P

Fin Si

## Foncpop :

Algorithmes des fonctions majeures sur population.

### Initialisation et Selection :

#### InitPopRecur

Résultat : Créer de manière récursive une population d'individu aléatoire suivant une probabilité donnée et suivant une taille de pop donnée.

InitPopRecur (Population p, int TaillePop, int LongIndiv, int prob) -> Population

Si TaillePop = 0 alors // TaillePop correspond au compteur pour compter le nombre d'individu à ajouter à la population

p -> Null // On marque la fin de la population par l'ajouter de l'element Null

Sinon

InitPopRecur ( p, TaillePop, LongIndiv, prob) = ajouter\_queue\_pop( InitPopRecur( p, TaillePop -1, LongIndiv, prob), InitialiserIndivRecursif(LongIndiv, prob), Longindiv)

// Definition récursive de InitPopRecur

#### InitPopIter

Résultat : Créer de manière itérative une population d'individu aléatoirement suivant une probabilité donnée et suivant une taille de population donnée.

InitPopIter (Population p, int TaillePop, int LongIndiv, int prob) -> Population p

p = creerP()

Pour i de 0 à TaillePop faire

p = ajouter\_queue\_pop(p, InitialiserIndivRecursif (LongIndiv, prob), LongIndiv)

Fait

#### PickIP

Résultat : Sélection d’un Individu au hasard dans une population

PickIP (Population, p, int TaillePop) -> Indiv

n = RandomN(TaillePop) // On choisi aléatoirement un nombre compris entre 1 et TaillePop

Tant que i < n faire

p = ResteP(p) // On passe au suivant pour trouver l'individu voulu

i= i+1

fait

#### SelectP

Résultat : Population p avec en fin de liste les tSelect premiers individus recopié a la fin de la liste

SelectP( Population p, int tSelect, int taillePop, int LongIndiv) -> Population p

tSelect = (taillePop\*tSelect)/100

Population tamp = p

i = 1

Tant que i < tSelect faire

tamp = ResteP(tamp)

i = i+1

Fait

Troncature de la liste :

tamp -> suivant = Null

Recopie des éléments pour completer la liste :

Pour i de 1 à taillePop/tSelect faire

RecopierP (tamp, p, tSelect, LongIndiv)

Fait

RecopierP (tamp, p, tSelect, LongIndiv) // On fait appel une derniere fois à RecopierP au cas ou tSelect ne soit pas un multiple de TaillePop

#### coller\_liste

Résultat : Population qui est le resultat du collage de deux populations

coller\_liste( Population A, Population B) -> Population p

Population temp = A

Tant que (temp ->suivant != Null) faire

temp = temp-> suivant

Faire

temp -> suivant = B

Fait

### Croiser Population :

#### CroiserP

Résultat : Croisement d'une population

CroiserP ( Population p1, int TaillePop, int pCroise, int LongIndiv) -> Population p2

Initialisation :

Population p2 = CreerP()

Population tamp1 = CreerP()

Population tamp2 = CreerP()

Ajout des 2 individus a p2 :

Tant que i <= TaillePop faire

// On choisit 2 individu au hasard dans p1 pour les mettre dans p2

p2 = ajouter-queue-pop ( p2, PickIP(p1, TaillePop), LongIndiv)

p2 = ajouter-queue-pop ( p2, PickIP(p1, TaillePop), LongIndiv)

// Croisement des 2 individus ajoutés :

tamp1 = p2

tamp2 = p2

tamp1 = Avant\_Dernier\_Pop(p2)

tamp2 = Dernier\_Pop(p2)

CroiserI (tamp1->Indiv, tamp2->Indiv, pCroise)

i = i+1

i= i+1

// Recalcul des qualites

QualiteP(p2, LongIndiv)

Fait

### Fonctions necessaire au Quicksort : Qualite, Recopier, Coller\_Liste :

#### QualiteP

Résultat : Calcul des qualités des individus d'une population

QualiteP(Population p, int LongIndiv) -> Population P // Recalcule la qualité des individus de la population

Population P = p

Tant que (NonVide(p)) faire

P -> qualite = QualiteI( p -> Indiv, LongIndiv) // On ajoute la qualite a la population

P = ResteP(P)

Fait

#### RecopierP

Résultat : Population p2 avec nbindiv de p1 recopié à la ifn de p2

RecopierP( Population p2, Population p1, int nbindiv, int LongIndiv ) -> Population p2

Si nbindiv = 0 alors

Rien

Sinon

p2 = ajouter\_queu\_pop ( p2, p1 ->Indiv, LongIndiv)

RecopierP ( p2, ResteP(p1), nbindiv-1, LongIndiv)

FinSi

### Quicksort :

Résultat : Population avec des individus triés par qualité croissante

Parametrage : quicksort: population A -> population

#### Cas trivial:

Si vide(A) Alors

quicksort=A

Fin si

#### Cas général:

Creer(S1)

Creer(S2)

Creer(P)

quicksort <- élément\_tête(A))

A <- Reste(A)

Tant que non\_vide(A) Faire

Si qualite(élément\_tête((quicksort))) < qualite(élément\_tête(A))) Faire

S1 <- ajouter\_queue(element\_tete(A))

Sinon

S2 <- ajouter\_queue(element\_tete(A))

Fin Si

A <- Reste(A)

Fait

Si non\_vide (S1) Alors

quicksort <- coller\_liste(quicksort(S1), quicksort)

Fin Si

Si non\_vide (S2) Alors

quicksort <- coller\_liste(quicksort, quicksort(S2))

Fin Si

#### Algorithme:

Si vide(A) Alors

quicksort=A

Sinon

Creer(S1)

Creer(S2)

quicksort <- élément\_tête(A))

A <- Reste(A)

Tant que non\_vide(A) Faire

Si qualite(élément\_tête((quicksort))) < qualite(élément\_tête(A))) Faire

S1 <- ajouter\_queue(element\_tete(A))

Sinon

S2 <- ajouter\_queue(element\_tete(A))

Fin Si

A <- Reste(A)

Fait

Si non\_vide (S1) Alors

quicksort <- coller\_liste(quicksort(S1), quicksort)

Fin Si

Si non\_vide (S2) Alors

quicksort <- coller\_liste(quicksort, quicksort(S2))

Fin Si

Fin Si

# Difficultés rencontrées lors de la réalisation du projet :

## Réalisation du MakeFile :

Après moult essaies infructueux cette étapes fut abandonnée. La compilation se passe bien lorsqu'elle est réalisée à la main ou sous un IDE, mais de nombreuses erreurs liées aux "includes" ressortent lorsque l'on utilise le MakeFile.

Cela n'a pas d'influence sur le fonctionnement du logiciel mais nous n'avons pas réussi a comprendre nos erreur sur le MakeFile.

## Comprehension du Quicksort :

L'énoncé a été difficile à comprendre pour un des membres du binôme. C'est l'esprit de l'algorithme qui n'est pas forcement explicite à la lecture de l'énoncé. En effet celui ci avait compris que l'algorithme devait couper la liste en un point précis pour donner deux listes bien ordonnées.

Hors cela est impossible et il faut créer les deux listes en y plaçant les éléments de la liste de base en fonction de leur qualité.

## Algorithme du tSelect et CroiserP :

Nous avons également rencontré des difficultés dans la compréhension de la partie "sélection" des individus ; en effet nous avons au départ compris que la fonction avait pour but de remplacer les 'tSelect' derniers individus d'une population par les premiers individus.

## Utilisation de srand :

Nous avons constaté qu'au bout d'un certain nombre de génération, on n'obtenait plus qu'un seul type d’individu dans la population. Nous pensons que cela est du au manque de précision de notre fonction random.

Nous avons cherché d'autre fonction que srand pour obtenir de l'aléatoire mais il est explicitement demandé dans le sujet d'utiliser rand, srand et time. Nous n'avons donc pas trouvé comment résoudre le problème.

Le reste du projet n'a pas posé de difficultés majeures.

# Jeu de test du logiciel :

Nous avons testé notre programme dans différentes conditions:

Les résultats complets sont situés en annexes dans le répertoire "Jeux d'essais".

## Tests sur le paramètre "tSelect":

Nous avons fait une première série de tests sur le paramètre tSelect en lui donnant 3 valeurs différentes: 10%, 50% et 90% pour les trois longueurs d'individus différentes.

Les autres paramètres utilisés sont les suivants:

pCroise = 50%

TaillePop = 200 individus

nGen = 200 générations

Le tableau ci-dessous donne la génération à partir de laquelle il n'y a plus qu'un seul type d'individu dans chaque population.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 8 bits | 16 bits | 32 bits |
| tSelect = 10% | 3 | 3 | 3 |
| tSelect = 50% | 8 | 6 | 7 |
| tSelect = 90% | 68 | 45 | 46 |

Nous avons constaté qu'à tSelect=90% les résultats sont très variables, la valeur donnée ici est donc une moyenne de ces résultats.

Nous pouvons donc en déduire que la longueur des individus et donc la formule de qualité n'ont pas d'influence sur le nombre de générations nécessaires à l'obtention d'une population à une seule sorte d'individu.

## Tests sur le paramètre "TaillePop":

Nous avons ensuite réalisé une série de tests similaires sur le paramètre TaillePop en lui donnant là aussi 3 valeurs différents: 20, 110 et 200 individus pour les 3 longueurs d'individus différentes:

Les autres paramètres utilisés sont les suivants:

pCroise = 50%

tSelect = 50%

nGen = 200 générations

De même, le tableau ci-dessous donne la génération à partir de laquelle il n'y a plus qu'un seul type d'individu dans chaque population.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 8 bits | 16 bits | 32 bits |
| TaillePop = 20 | 5 | 3 | 5 |
| TaillePop = 110 | 6 | 5 | 6 |
| TaillePop = 200 | 68 | 45 | 46 |

Nous pouvons donc en déduire que la taille de la population n'a pas d'influence sur le nombre de génération nécessaire à l'obtention d'une population à une seule sorte d'individu et que seul le taux de sélection influe.

# Conclusion

Fefe