Logo

Description automatically generated

projet rn40

Muna Ayuni BINTI MAHATHIR

Gabriella Ndjamba Batomen

Introduction

Le projet porte à la manipulation des types abstraits de donnée populations et individu dans le but de crée un algorithme génétique. Dans notre rapport nous allons lister nos algorithmes et les expliquer. Nous avons eu pour paramètres:

-Longueur d'un individu : longIndiv = 8

-Probabilité de croisement : pCroise = 0.5

-Taille de la Population : 20 ≤ TaillePop ≤ 200

-Taux de sélection : 10% ≤ tSelect ≤ 90%

-Nombre de générations : 20 ≤ nGen ≤ 200

Au depart nous avons définit un alias pour le type unsigned char appelé Bit,puis nous avons définit un nouveau type appelé BitList, qui est une structure contenant deux champs : bit : une valeur de type Bit (qui est un alias pour unsigned char), next : un pointeur vers un BitList (Le type BitList représente une liste chaînée de valeurs Bit). Le champ next de chaque élément BitList pointe vers l'élément suivant de la liste, ou est égal à NULL si l'élément est le dernier élément de la liste.

Individu

Avant d’ecrire notre sous programme pour initialiser une liste de bit nous avons ecrit un sous programme pour ajouter un bit au début de notre liste chaînée. Elle prend deux paramètres : bit1 : une valeur de type Bit, ind1 : une valeur de type Individu qui représente la liste chaînée existante.

Ce sous programme alloue de la mémoire pour Individu en utilisant la fonction malloc. Elle assigne la valeur de bit1 au champ bit et définit le champ next sur la valeur de ind1, qui est la liste chaînée existante.

1. Initialiser aléatoirement la liste de bits version itérative

Le sous-programme NewIndividuIter vas créer un Individu appelé new en l'initialisant à NULL, puis elle utilise une boucle pour itérer Longindiv fois et ajouter un nouvel élément à la tête d’Individu à chaque itération en utilisant la fonction ajout\_tete. L'argument bit1 passé à ajout\_tete est défini sur le résultat de rand() % 2, qui génère un entier aléatoire entre 0 et 1 (inclus) et l'assigne à bit1, le champ bit de chaque nouvel Individu ajouté à la liste chaînée sera défini sur un 0 ou un 1 aléatoire. La fonction renvoie l'objet Individu résultant.

Lexique :

R : une liste de bit générer aléatoirement

i: un entier

New: un Individu

Longindiv : Une constante qui représente la longueur d’un indivividu (8)

Résultat : R

NewIndividuIter()<- Individu

new : Null

Pour i de 0 à longindiv faire

New:= ajoute\_tete(random de 0 a 1, new)

Fin

NewIndividuIter():=new

1. Initialiser aléatoirement la liste de bits version récursive

Le sous-programme NewIndividuRecur crée une nouvelle instance d’Individu et la renvoie. Elle prend deux arguments : Individu appelé ind et un entier appelé taille. Le sous-programme commence par ajouter un nouvel élément à la tête de l’Individu en utilisant la fonction ajout\_tete, en passant ind en tant que second argument comme sur l’algorithme décrit plus hauts. Ensuite, la fonction vérifie si taille est différent de 1. Si c'est le cas, elle appelle elle-même récursivement en passant ind et taille - 1 en tant qu'arguments, et renvoie le résultat de cet appel. Sinon, elle renvoie simplement ind. Ce sous-programme crée récursivement une nouvelle instance d’Individu en ajoutant un bit à la tête de la liste chaînée à chaque appel récursif, jusqu'à ce que taille atteigne 1. Elle renvoie alors l’Individu résultante.

Lexique :

taille : un entier

ind : un individu

Résultat : une liste de bit générer aleatoirement

NewIndividurecur( taille: entier, new: Individu )<- Individu

Ind:= ajoute\_tete(ind, random 0 à 1)

Si !(taille= 1) alors NewIndividurecur (ind,taille-1)

Fin

NewIndividurecur():=ind

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

Le sous-programme Dec prend un Individu en argument et renvoyer un entier. En commençant par déclarer une variable res initialisée à 0. Ensuite, elle utilise une boucle while pour itérer tant que ind n'est pas NULL. À chaque itération, elle multiplie res par 2 et ajoute le champ bit de l'Individu courant. Ensuite, elle affecte à ind la valeur du champ next de l’Individu courant. Lorsque la boucle se termine, le sous-programme renvoie la valeur de res.

Lexique :

I: résultat d’une liste décoder pour obtenir sa valeur entier

res: un entier

Résultat :I

Dec(ind:individu)<- int

res:=0

Tant que !(vide(ind ))

Alors

res:=(res \*2) + bit(ind)

ind:=next(ind)

fin

Dec:=res

4. Croiser deux listes de bit

Dans cette fonction, ind1 et ind2 sont deux individus qui sont croisés pour produire un nouvel individu. La fonction génère d'abord une probabilité aléatoire prob entre 1 et 99. Si prob est inférieur ou égal à une probabilité fixe PCROISE fois 100, alors la fonction prend le champ bit de ind1 et le combine avec le résultat de l'appel récursif de la fonction sur les champs next de ind1 et ind2. Si prob est supérieur à PCROISE fois 100, alors la fonction prend le champ bit de ind2 et le combine avec le résultat de l'appel récursif de la fonction sur les champs next de ind1 et ind2. Finalement, la fonction retourne le résultat de l’appel du sous-programme ajout\_tete.

Lexique :

Prob : un entier

PCROISE : Probabilité de croisement (0.5)

ind1,ind2 :Individu

Résultat :

CroiserInd (ind1,ind2 :Individu)<-Individu

prob := 1+ random de 1 à 99

si (ind1=NULL ou ind2=NULL) alors CroiserInd (ind1,ind2 ) := NULL

fin

si (prob<=PCROISE\*100) alors ajout\_tete(ind1->bit, CroiserInd(ind1->next,ind2->next))

sinon ajout\_tete(ind2->bit, CroiserInd(ind1->next,ind2->next))

Fs

Fin

5. Calculer la qualité d’un individu

Ce sous-programme calcule la qualité d'un individu, val. La fonction convertit val en flottant en le divisant par 2^8. Elle calcule ensuite le carré de cette valeur et renvoie l'opposé de ce résultat.

Lexique :

X : flottant

Résultat :

Qualite (val :entier)<-flottant

X : =(val/pow(2,8))\*(B-A)+A

Qualite(val) := -(pow(x,2))

Population

Pour le type de donnée population nous avons créé une structure ind qui représente un élément d'une liste chaînée de structures Individu. Elle contient un pointeur vers une structure Individu et un pointeur vers le prochain élément de la liste. La structure Population est un type pointeur vers la structure ind.

Avant d’écrire l’algorithme pour initialiser une liste d’individu nous avons écrit une fonction ajoutert. Cette fonction est semblable à celle expliquer au débuts de la partie individu (ajoute\_tete). Elle ajoute un individu (ind) à une population (pop). Elle le fait en créant un nouvel objet population (temp) et en définissant le champ id sur ind. Elle définit ensuite le champ next de temp sur la population actuelle (pop) et renvoie la nouvelle population (temp).

1. Initialiser une liste d’individu

La fonction NewPopulation() crée une nouvelle instance de Population en initialisant une liste vide new, puis elle utilise une boucle pour itérer NB\_IND fois en ajoutant un nouvel élément à la tête de population à chaque itération en utilisant la fonction ajoutert et NewIndividuIter().

Lexique :

New : une population

I :un entier

NB\_IND :taille de la population

Résultat :

NewPopulation() <-population

New :=NULL

Pour i de 0 à NB\_IND

New :=ajoutert(NewIndividuIter(),new)

Fin

NewPopulation() := new

2. Triée la liste d’individu par qualité décroissante

Pour pouvoir triée notre liste d’individu nous aurons besoin des trois sous programmes dont last\_node, dividepop et quick\_sort. Le sous-programme last\_node est utilisée pour obtenir un pointeur vers le dernier élément de Population, elle fait cela en parcourant la liste ind et en retournant un pointeur vers l'élément qui a un champ next nul. Le sous-programme DividePop est utilisée pour diviser Population en deux sous-populations à un point donné dans la liste. Elle fait cela en sélectionnant un élément pivot et en parcourant la liste, en échangeant les valeurs des éléments qui sont plus grands que le pivot avec celles des éléments qui sont plus petits. La fonction échange ensuite la valeur de l'élément pivot avec la valeur du dernier élément de la liste et retourne un pointeur vers le pivot. Le sous-programme quick\_sort est utilisée pour trier les éléments de Population. Elle le fait en divisant récursivement la liste en sous-listes à l'aide de la fonction DividePop et en triant les sous-listes. La fonction se termine lorsqu'elle atteint une sous-liste avec un seul élément, qui est déjà trié.

Lexique :

Head, temp : une liste d’individu

Résultat :

Last\_node(head : ind) <-ind

temp:= head

tant que !(temp :=Null) et !(next(temp :=Null))

temp :=next(temp)

Fin

Last\_node(head) := temp

Lexique :

Pivot, first, current, temp, last : une liste d’individu

Résultat :

Dividepop (first, last :ind) <-ind

Pivot :=first

Current :=first

tant que !(current :=Null) et !(current :=last)

si Qualite(Dec(id(current)))> Qualite(Dec(id(last)))

Pivot :=first

Temp := id(first)

Id(first) :=id(current)

id(current) := temp

first :=next(first)

fs

current :=next(current)

fs

temp :=id(first)

id(first) :=id(last)

id(last) :=temp

Fin

Dividepop(first, last) := pivot

Lexique :

First, last, pivot : une liste d’individu

Résultat :

Quick\_sort(first, last :ind) <-rien

Si (first=last) alors rien

fin

Pivot := Dividepop(first, last)

si !(pivot :=Null) et !(next(pivot :=Null))

quick\_sort(next(pivot),last)

Fin

Si !(pivot :=Null) et !(first := pivot)

Quick\_sort(first, pivot)

Fin

3. Sélectionner le meilleur individu de la population

Le sous-programme meilleur prend une Population en entrée et retourne une Population contenant les T\_SELECT individus les plus performants de la population d'origine. Elle le fait en parcourant ind de Population et en sélectionnant les T\_SELECT individus qui ont la meilleure qualité, qui est mesurée en utilisant la fonction Qualite. La fonction Qualite calcule la qualité d'un individu en fonction de sa valeur décodée (obtenue à l'aide de la fonction Dec) et renvoie une valeur flottante représentant la qualité de l'individu.

Lexique :

Head, chosen, stop : une liste d’individu

P : une population

I :un entier

T\_SELECT : taux de sélection

Résultat :

Meilleur (p : population) <-population

Head : = p

Chosen :=p

Stop :=p

Pour i de 0 à T\_SELECT

Head :=next(head)

Stop :=next(stop)

Fin

Tant que !(head :=Null)

Id(head) :=id(chosen)

Head :=next(head)

Si !(next(chosen :=stop))

Alors chosen :=next(chosen)

Fs

Sinon chosen :=p

Fs

Meilleur (p) :=p

Fin

4. Croiser la population

Le sous-programme CroiserPop est utilisée pour créer une nouvelle population en croisant les individus de la population d'origine. Elle prend une Population en entrée et retourne une Population contenant les individus croisés.

Lexique :

Popcroise : une population

I, j, k, random1, random2 :un entier

NB\_IND :taille de la population

Pointer1, pointer2 : une liste d’individu

Résultat :

Croiserpop (p :population) <-population

Popcroise :=NULL

Pour i de 0 à NB\_IND

Random1 := génère un nombre aléatoire entre 1 et NB\_IND-1

Random2 := génère un nombre aléatoire entre 1 et NB\_IND-1

Pointer1 :=p

Pointer2 :=p

Pour j de 1 à random1

Pointer1 :=next(pointer1)

Fin

Pour k de 1 à random2

Pointer2 :=next(pointer2)

Fin

Popcroise := ajoutert(CroiserInd(id(pointer1),id(pointer2),popcroise)

Fin

Croiserpop (p) :=popCroise

Fin

Jeux d’essais

Avec le sous-programme Qualite originale les individus ayant des valeurs plus élevées ont une qualité inférieure et les individus ayant des valeurs plus faibles ont une qualité supérieure.

Le nouveau sous-programme Qualite, les individus ayant des valeurs plus élevées ont une qualité inférieure et les individus ayant des valeurs plus faibles auront une qualité supérieure. Nous pensons que cela signifie que l'algorithme sera plus enclin à sélectionner et à reproduire les individus ayant des valeurs plus faibles, car ils seront perçus comme étant de qualité supérieure.

Résultat

La première étape du code consiste à créer une nouvelle population d'individus aléatoires en utilisant la fonction NewPopulation. Ensuite, le code entre dans une boucle qui itère NGEN fois. À chaque itération de la boucle, le code crée une nouvelle population en croisant des individus de la population précédente en utilisant la fonction CroiserPop. La nouvelle population est ensuite triée à l'aide de la fonction quick\_sort. Le code sélectionne ensuite les T\_SELECT meilleurs individus de la population triée en utilisant la fonction meilleur. Ce processus est répété pour NGEN itérations. Après la fin de la boucle, le code affiche le meilleur individu de la population finale en appelant la fonction PrintIndividu. Cet individu est choisi dans la population triée à l'aide de la fonction meilleur.