MIGNEROT Grégori – PLANCHE Antoine

# LO21 – Automne 2019 : Projet

## Conception

Le principal choix d'implémentation des structures de données, a été celui d'utiliser exclusivement des listes simplement chaînées pour les types *Individu* et *Population* pour des raisons de simplicité. En effet, l'utilisation de structures plus complexes ne simplifiant réellement que très peu les algorithmes. L'utilisation de tableaux plutôt que de listes chaînées, voire simplement d'entier non signés pour les individus (par définitions, des suites de bits), aurait potentiellement pu simplifier encore certaines parties du projet, mais sort du cadre de cet exercice.

# Algorithmes des sous-programmes

*Dans tous les algorithmes ci-dessous, afin d'assurer une meilleure simplicité d'écriture et d'éviter la redondance de sections d'algorithme triviales :*

*On définit la fonction aléa() comme retournant un réel aléatoire entre 0 et 1.*

*On définit aléa_entier(min, max) comme retournant un entier aléatoire entre min et max inclus*

*On définit aléa_bit() comme retournant un bit aléatoire, raccourci pour arrondi(aléa())*

*On définit swap(variable1, variable2) comme échangeant les valeurs des deux variables*

*On écrira ci-dessous Erreur en cas d'erreur empêchant le sous-programme de continuer*

*On définit le type Bit comme une valeur de 0 ou 1*

*On définit le type Individu comme une liste de Bits*

*On définit le type Population comme une liste d'Individus*

*On définit les fonctions abstraites de base sur les listes : créer_liste(), ajouter_queue(liste, valeur), insérer(liste, index, valeur), taille(liste), valeur(liste, index), existe(liste, index)*

*On définit la fonction créer_individu(longueur) comme créer_individu_itératif ou créer_individu_récursif*

## Opérations sur les individus

### Créer_individu_itératif

Cette fonction construit aléatoirement un individu de façon itérative.

| | |
|---|---|
| `créer_individu_itératif(longueur : entier) -> Individu`<br><br>`Si longueur = 0 : Erreur`<br>`indiv <- créer_liste()`<br>`Pour i allant de 1 à longueur :`<br>`\|  ajouter_queue(indiv, alea_bit())`<br>`créer_individu_itératif <- indiv` | Données :<br>- longueur : Entier, nombre de bits composant l'individu |
| | Résultat : Individu aléatoire de la longueur donnée |
| | Lexique :<br>- indiv : L'Individu en cours de création<br>- i : Entier, compteur de boucle |

### Créer_individu_récursif

Cette fonction construit aléatoirement un individu de façon récursive

| | |
|---|---|
| `continuer_individu(indiv : Individu, longueur : entier)`<br><br>`Si longueur > 0 :`<br>`\|  ajouter_queue(indiv, alea_bit())`<br>`\|  continuer_individu(indiv, longueur−1)` | Données :<br>- indiv : Individu en cours de création<br>- longueur : Entier, nombre de bits composant l'individu |
| | Résultat : Individu aléatoire de la longueur donnée |
| | Lexique :<br>- indiv : L'Individu en cours de création<br>- i : Entier, compteur de boucle |

| | |
|---|---|
| `créer_individu_récursif(longueur : entier) -> Individu`<br><br>`Si longueur = 0 : Erreur`<br>`indiv <- créer_liste()`<br>`continuer_individu(indiv, longueur)`<br>`créer_individu_récursif <- indiv` | Données :<br>- longueur : Entier, nombre de bits composant l'individu<br>Résultat : Individu aléatoire de la longueur donnée<br>Lexique :<br>- indiv : Individu créé<br>- indiv : L'Individu en cours de création<br>- i : Entier, compteur de boucle |

### Copier_individu

Copie entièrement l'individu fourni, et renvoie la copie

| | |
|---|---|
| `copier_individu(base : Individu) -> Individu`<br><br>`indiv <- créer_liste()`<br>`i = 0`<br>`Tant que i <  taille(base):`<br>`\|   ajouter_queue(indiv, valeur(base, i))`<br>`copier_individu <- indiv` | Données :<br>- base : Individu à copier<br>Résultat : Copie de l'Individu donné<br>Lexique :<br>- indiv : L'Individu en cours de création<br>- i : Entier, compteur de boucler |

### Valeur_décimale

Convertit un Individu en un nombre entier représentant la valeur décimale de la liste de bits qui le composent

| | |
|---|---|
| `valeur_décimale(indiv : Individu) -> Entier`<br><br>`dec <- 0`<br>`Pour i allant de 1 à taille(indiv) :`<br>`\|   dec <- dec + valeur(indiv, i) * 2`$^{i}$<br>`valeur_décimale <- dec` | Données :<br>- indiv : Individu traité<br>Résultat : Valeur de la liste de Bits<br>Lexique :<br>- dec : Entier, valeur actuelle du résultat<br>- i : Entier, compteur de boucle et puissance de 2 courante |

### Qualité_individu

Donne la qualité d'un individu, étant donné une fonction et les paramètres de la mise à l'échelle

| | |
|---|---|
| `qualité_individu(indiv : Individu, A : Réel, B : Réel, F(x) : fonction) -> Individu`<br><br>`dec <- valeur_décimale(indiv)`<br>`X <- (dec / 2`$^{taille(indiv)}$`) * (B – A) + A`<br>`qualité_individu <- F(X)` | Données :<br>- indiv : Individu dont la qualité est déterminée<br>- A, B : Réels, paramètres de la mise à l'échelle<br>- F(x) : fonction de qualité<br>Résultat : Qualité de l'Individu pour la fonction donnée<br>Lexique :<br>- dec : valeur décimale de l'individu<br>- X : Réel, valeur de l'individu mise à l'échelle |

## Croiser_individus

Echange aléatoirement certains bits des individus, selon une certaine probabilité.

```
croiser_individus(indiv1 : Individu,
indiv2 : Individu, probabilité : Réel)

Pour i allant de 0 à taille(indiv1) :
|   tirage = aléa()
|   Si tirage < probabilité :
|   |   swap(valeur(indiv1, i),
valeur(indiv2, i))
```

Données :

- indiv1, indiv2 : Individus à croiser, que l'on admet de longueurs identiques
- probabilité : Réel, probabilité que chaque bit des deux individus soit échangé, entre 0 et 1

Résultat : Les deux individus sont modifiés sur place

Lexique :

- i : Entier, compteur de boucle
- tirage : nombre aléatoire généré

## Opérations sur les populations

### Créer_population

Crée une population aléatoire de la longueur donnée

```
créer_population(longueur : entier, longIndiv : entier) -> Individu

Si longueur = 0 : Erreur
P <- créer_liste()
Pour i allant de 1 à longueur :
|   ajouter_queue(P, créer_individu(longIndiv))
créer_population <- P
```

Données :
- longueur : Entier, nombre d'Individus composant la population
- longIndiv : Entier, taille d'un individu de la population

Résultat : Population aléatoire de la longueur donnée

Lexique :
- P : La Population en cours de création
- i : Entier, compteur de boucle

### Quicksort_population

Trie les individus de la population par qualité décroissante.

Ce sous-programme utilise une variante de l'algorithme *Quicksort*, un algorithme de tri connu pour son efficacité. La principale difficulté d'implémentation a été de l'adapter à des listes simplement chaînées – ce qui a nécessité de prendre quelques libertés par rapport à l'algorithme original, sans quoi l'implémentation était soit extrêmement difficile, soit très inefficace.

Le principe de cet algorithme est de sélectionner un « pivot ». On place ensuite toutes les valeurs de la liste supérieures au pivot avant celui-ci, et toutes les valeurs inférieures après le pivot (pour trier par valeurs décroissantes). On trie ensuite récursivement les deux parties de la liste par ce même algorithme.

Ici, on utilise le premier élément de la liste comme pivot : ainsi, il suffit ensuite de passer tous les éléments de la liste supérieurs au pivot avant celui-ci (ici, simplement au début de la liste), simplifiant de beaucoup le traitement.

```
quicksort_population(P : population, pos : Entier, longueur : Entier)

Si longueur > 1 :
|   gauche <- 0
|   i <- 0
|   pivot = valeur(P, 0)
|   Pour i allant de pos à pos + longueur :
|   |   indiv <- valeur(P, i)
|   |   Si qualité_individu(indiv) > qualité_individu(pivot) :
|   |   |   supprimer(P, i)
|   |   |   insérer(P, pos, indiv)
|   |   |   gauche <- gauche + 1
|   quicksort_population(P, pos, gauche)
|   quicksort_population(P, pos + gauche + 1, longueur – gauche - 1)
```

Données :
- P : Population à trier
- pos : Début de la section à trier
- longueur : Taille de la section à trier

Résultat : La Population est triée sur place

Lexique :
- gauche : Entier, taille de la partie de la liste avant le pivot
- i : Compteur de boucle
- indiv : Individu courant

## Sélectionner_population

Sélectionne les premiers individus de la population et les répliquent en remplaçant tous les autres individus.

Ce sous-programme ne fait que copier les premiers individus de la liste et les mettre à la place des autres, et ne sélectionne pas les meilleurs individus à proprement parler. La Population est donnée triée par ordre de qualité décroissante, ce qui donne finalement que les éléments répliqués sont effectivement ceux à la qualité la plus élevée.

| | |
|---|---|
| `sélectionner_population(P : Population, nbSelect : Entier)`<br><br>`base <- 1`<br>`Pour i allant de nbSelect + 1 à taille(P) :`<br>`\|  valeur(P, i) <- copier_individu(valeur(P, base))`<br>`\|  base <- base + 1`<br>`\|  Si base > nbSelect :`<br>`\|  \|  base = 1` | Données :<br>- P : Population à traiter<br>- nbSelect : Nombre d'individus recopiés<br><br>Résultat : La population donnée est traitée sur place et ne contient plus que les nbSelect premiers individus en boucle<br><br>Lexique :<br>- base : Entier, index de l'individu à copier<br>- i : Entier, index courant |

## Croiser_population

Croise aléatoirement les individus de la population deux à deux.

Pour cela on utilise un tableau d'attribution : pour chaque individu, on choisit un index aléatoire dans un tableau de la même taille que la population, puis on place l'individu à cette place où à la prochaine place libre depuis cet index. Il suffit ensuite de prendre les individus dans le tableau deux par deux et d'appeler croiser_individus.

| | |
|---|---|
| `croiser_population(P : Population, probabilité : Réel)`<br><br>`Pour i allant de 1 à taille(P) :`<br>`\|  index <- aléa_entier(1, taille(P))`<br>`\|  Tant que existe(attribution, index) :`<br>`\|  \|  index <- (index + 1) mod taille(P)`<br>`\|  valeur(attribution, index) <- valeur(P, i)`<br><br>`Pour i allant de 1 à taille(P) par pas de 2 :`<br>`\|  croiser_individus(valeur(attribution, i), valeur(attribution, i + 1), probabilité)` | Données :<br>- P : Population à traiter<br>- probabilité : Réel, probabilité que chaque bit des individus soit échangé (transmis tel quel à croiser_individus)<br><br>Résultat : La population donnée est traitée sur place<br><br>Lexique :<br>- attribution : liste d'Individus de la même taille que la population<br>- i : Entier, compteur de boucle<br>- index : Entier, index sélectionné dans le tableau d'attribution |

## Meilleur_individu

Sélectionne simplement l'individu à la qualité la plus élevée dans la Population. S'apparente à une simple fonction max()

| | |
|---|---|
| `meilleur_individu (P : Population) -> Individu`<br><br>`meilleur <- valeur(P, 1)`<br>`Pour i allant de 1 à taille(P) :`<br>`\|  Si qualité_individu(valeur(P, i)) > qualité_individu(meilleur) :`<br>`\|  \|  meilleur <- valeur(P, i)`<br>`meilleur_individu <- meilleur` | Données :<br>- P : Population à traiter<br>Résultat : L'Individu à la meilleure qualité dans la Population<br>Lexique :<br>- meilleur : Individu à la meilleure qualité trouvé |

## Commentaires

Dans le programme, plusieurs fonctions de qualité données sont implémentées, et donnent des résultats différents. Cependant, bien que les résultats soient différents, on retrouve globalement certaines similarités. Premièrement, on voit bien que quelques soient les paramètres dans les intervalles donnés, le meilleur individu tend toujours vers une qualité maximale : -1 pour $f_{1(x)} = -x^2$, ln(10) pour $f_2(x) = -\ln(x)$ et 1 pour $f_{3(x)} = -\cos(x)$. On remarque aussi que le résultat final est moins variable pour une population plus grande ou pour un plus grand nombre de génération, et que plus la probabilité est éloignée de 0.5, plus il faut de générations pour tendre vers la qualité maximale.