Optimisation

2.1.1 Etat/individu

La métaheuristique utilisée est l'algorithme génétique. Le but est de créer des populations de solutions aléatoires puis les évaluées et sélectionner les plus aptes à résoudre notre problème, c'est-à-dire optimiser les zones du claviers en prenant en compte le placement des futurs lettres.

Question 1)

Nous allons créer une tableau 4x10 qui va représenter la disposition des lettres sur le clavier. La case représentera une touche du clavier avec à l'intérieur un caractère.

En fonction de l'algorithme génétique une solution serait comme un chromosome c'est-à-dire un tableau de caractères avec une disposition de touches aléatoires. Ces chromosomes seraient manipulés par des opérateurs génétiques pour générer de nouvelles solutions

Question 2)

La taille de l'espace de recherche serait de 40!.En effet il s'agit d'un cas de tirage sans remise étant donné que le clavier ne peut comporter qu'une seule fois une même lettre.

Question 3)

Pour chaque paire de lettres adjacentes dans notre disposition, on effectue un calcul de fréquence de bigramme correspondante depuis le fichier freqBigrammes.txt. On pondère chaque fréquence par le nombre de fois que ce bigramme apparaît dans le texte. On fait la somme de toutes ces fréquences pondérées pour toutes les paires de lettres adjacentes dans la disposition. Cette somme donne donc la valeur d'adaptation de la solution candidate.

Question 4)

La fonction d'adaptation proposée pour évaluer la qualité d'une disposition de touches pourrait être :
-Pour un doigt : Distance totale parcourue par ce doigt calculée à l'aide de la formule : distance totale = (nombre de caractères dans le texte -1) + nombre de touches survolées pour atteindre le prochain caractère * 1cm.
-Pour deux doigts : Distance totale parcourue par ces deux doigts calculée à l'aide de la formule :
distance total première main = (nombre de caractère dans le texte se trouvant dans la partie gauche -1) + nombre de touches survolées pour atteindre le prochain caractère * 1cm
distance total deuxième main = (nombre de caractère dans le texte se trouvant dans la partie droite - 1) + nombre de touches survolées pour atteindre le prochain caractère * 1cm

distance total parcouru par les deux mains = distance total première main + distance total deuxième main

Question 5 c)

1. L'initialisation des états de départ serait fait aléatoirement de façon à ce que chacun représente une disposition de touches de claviers différentes. Ce qui permet d'avoir une diversité plus importante nous permettant d'avoir un choix plus vaste sur notre sélection élitique.

2. La population serait de taille arbitraire, ici on choisit 1 040 (26*40). Le but étant que chaque lettre ai parcouru toutes les positions allant de 1 à 40 dans le tableau.

3. Grâce à la fonction d'adaptation nous allons choisir les parents qui retourne la plus grande valeur d'adaptation. Nous allons diviser notre population par deux avec d'un côté les individus avec la plus grande valeur et de l'autre côté des individus qui seront abandonnés.

4. L'opérateur de croisement utilisé est le simple enjambement. En effet cette solution permet par rapport au simple enjambement de minimiser le risque d'avoir des enfants similaires sur plusieurs croisements.

5. La permutation serait de changer deux touches hasard parmi les 40. La probabilité de mutation est de 0,1 à 0,01.

6. L'opérateur de remplacement utilisé serait le remplacement générationnel. En effet chaque ancienne génération serait remplacée par la nouvelle.

7. Le critère d'arrêt pourrait être un nombre d'itération prédéfini ou un temps d'exécution.

8. Le nombre de générations doit être suffisamment grand pour permettre d'obtenir une solution optimale, mais pas trop grand pour éviter un temps de calcul excessif. Un bon choix pourrait se situer entre 100 et 250 générations.

2.3 Rapport

Nous commençons par créer un fichier « Structure.h » qui contiendra les fonctions , les structures, les define etc… de nos individus et de ce qu'ils contiennent.

Dans Structure.h :

Des #define qui définira une lettre à un entier, par exemple : « A 1 ». Car A est la 1ère lettre de l'alphabet. Un autre qui définira l'addition de tous les entiers qui se trouvent dans le tableau d'occurrences situé dans le fichier « freqBigrammes.txt ».

Des structures comme la structure « touche » qui sera définie :

- Une touche d'un clavier par un entier ligne et colonne qui montrera sa position.
- Une lettre qui sera sur cette touche.

La structure « individu » définie par :

- Un tableau de 4 lignes et 10 colonnes « int** tab » qui représente un clavier de 40 touches.
- Un tableau de touches « touche_s* touches » qui représentera la position et les lettres qui se trouvent dans l'individu.
- Un entier « nb_vide » qui représente le nombre de cases vides.
- Un entier « nb_rempli » qui représente le nombre de cases remplies.
- Un double « note » qui représente la note d'évaluation (ratio de distance fait entre les touches, aussi équivaut à représenter l'efficacité de cette combinaison de touche).

Des prototypes du fichier « Structure.c ».

Dans Structure.c :

- Un tableau de taille 26 par 26 qui représente les valeurs qui se trouvent dans le fichier « freqBigrammes.txt », chaque ligne et chaque colonne représente une lettre de l'alphabet.
- Des fonctions :

void remplir_tableau(individu_s* individu) : // fonction de remplissage d'un individu

Tant que le nombre de cases remplies est différent de 26 (= nombre de lettres) Alors

On créé deux nombre aléatoires qui représenteront les coordonnées d'une touche,

On test si cette touche a déjà été remplie (= 0) si non alors on remplit la touche, etc…

bool estVide(individu_s individu) : // fonction qui permet de savoir si l'individu est vide

void calcul_note(individu_s* individu) : //fonction qui permet d'évaluer un individu en fonction de la distance entre chaque touches.

Pour chaque combinaison de lettre et si la somme des deux occurrences != 0 alors

On fait le calcul d'un ratio qui est calculé en fonction de la distance entre les touches et de la pertinence de cette distance entre les touches (Plus c'est pertinent, plus la distance est courte mieux c'est).

Si = 0 on ne le fait pas car il n'y aura presque jamais une pression sur ces deux touches successivement.

On divise la note par 18000000

individu_s init_Individu() : //Fonction qui initialise un individu

Allocation de la mémoire pour un individu de taille de la structure individu.

Allocation de la mémoire pour un tableau de touches de taille 26.

Cases vides instanciées à 40 de base.

Cases remplies instanciées à 0 de base.

Allocation de la mémoire pour un tableau qui représente un clavier, un tableau

A double entrée de taille 4 par 10.

On instancie et alloue de la mémoire dans ce tableau et instancie les cases à 0

On complète le tableau

On calcul les notes de chaque individu

individu_s* initialisation_pop (int x) : // fonction d'initialisation d'une population

On alloue de la mémoire pour un tableau d'individus de taille n donné en paramètre.

Ensuite, on le parcours et on initialise chaque individu.

On tri de façon croissante ce tableau d'individus.

void print_touches(individu_s individu) : //fonction qui permet d'afficher sur la sortie standard les diverses informations sur les touches de l'individu.

void print_info_individu(individu_s individu) : //fonction qui permet d'afficher sur la sotie standard les informations relative à l'individu.

int length(individu_s* individus) : //calcul le nombre d'individus dans notre échantillon

individu_s* select_parent(individu_s* individus) : // fonction de création de la prochaine génération ; cette fonction représente l'opérateur de sélection des parents.

On alloue de la mémoire pour un tableau d'individus de taille du tableau d'individus donné en paramètre divisé par 2.

On définie une variable position à la taille / 2

On parcours individus en partant de la fin jusqu'à sa moitié pour récupérer la moitié supérieur et la mettre dans le nouveau tableau d'individus qui est parent.

Car dans la moitié supérieur, il se trouve les individus les plus intéressant (trié par fusion en fonction de leur note, plus elle est proche de 0, mieux c'est).

void fusion(individu_s arr[], int left, int middle, int right) : //fonction qui fusionne les 2 tableaux triés.

On va créer deux tableaux temporaires.

Copier les données dans les tableaux temporaires.

Fusionner les tableaux temporaires dans le tableau d'individus en regardant les marques des individus (si la note de l'array2 est >= à celle de l'array3 alors on va choisir de rentrer celui de l'array2.

void mergeSort(individu_s arr[], int left, int right) : // fonction de tri fusion

void simpleEnjambement(individu_s* enfants , individu_s pere, individu_s mere) : // Effectue le croisement à simple enjambement entre le père et la mère. Retourne un tableau avec les deux enfants s'ils sont valide.

Création de deux individus.

On vérifie que le nombre de cases vides est suffisant pour faire le croisement

On parcours les 4 lignes du « clavier ».

On mets la partie gauche du père dans le fils et la partie gauche de la mère dans la fille en parcourant les 5 premières colonnes.

On mets la partie droite du père dans le fils et la partie droite de la mère dans la fille

en parcourant les 5 dernières colonnes.

Ensuite, on passe à une partie vérification qui va parcourir le clavier et vérifier si dans le

Clavier du fils il y'a 26 lettres et qu'1 fois la même.

Pareil pour la fille.

Pour finir, on va ajouter des enfants dans le tableau d'enfants s'ils sont aptes.

Et on va trier par fusion ce tableau d'enfants.

unsigned long long facto(int n) : // fonction qui calcule la factorielle de n

void permutation(individu_s individu) : // fonction qui permet de permuter deux touches de l'individu.

Coordonnées prisent au hasard des deux touches qui vont être permutés.

void freeIndividu(individu_s* individu) : //Fonction qui libère la mémoire allouée à l'individu.

void freePopulation(individu_s* pop) : //Fonction qui libère la mémoire allouée à une population d’individus.