Rapport TD Optimisation

Métaheuristique choisie : algorithme évolutionnaire

Q1 :

Un état pourrait être défini par une matrice de lettres de 40 cases (4 lignes et 10 colonnes, la transposée ne change en rien la solution). L’implémentation pourra se faire par un tableau linéaire ou à deux dimensions, avec une norme associée qui reste à déterminer. On pourra par exemple utiliser la norme associée à la distance de Manhattan, ou même redéfinir une distance sur ⟦ 1, 10 ⟧ x ⟦ 1, 4 ⟧.

Q2 :

Le tableau possède 40 cases. Parmi ces 40 cases, 26 seront occupée par une lettre (toutes différentes). On a donc un arrangement de 26 parmi 40, soit 40! / (40 - 26)! combinaisons possibles.

La taille de l’espace est donc environ 9.36 × 10^36 (Je n’ai pas vérifié, faisons confiance à Wolfram Mathematica).

Q3 :

On peut poser une norme sur le clavier associée à la distance de Manhattan entre les touches.

Alors, pour tout clavier C, ||C|| = somme(i,j)(poids(bigramme(i,j) \* distance(i, j)).

La fonction d’adaptation serait alors la minimisation de cette norme, soit min(||C||) ou max(1/||C||).

Q4 :

Lorsqu’on tape avec un doigt, la formule reste la même. Lorsqu’on tape avec plusieurs doigts, on peut diviser le clavier en autant de zones qu’on utilise de doigts, en considérant que la distanc eentre deux touches de zones distinctes est 0. La formule ne change donc pas, seule la distance se trouve modifiée.

Q5 :

Décrit dans le rapport page suivante.

Rapport détaillé

Nous avons choisi d’utiliser le C++ pour implémenter cet algorithme génétique, car il s’agit d’un langage rapide et permettant l’utilisation d’objets (ce qui est préférable). La rapidité était un critère majeur pour le traitement de grandes populations, car nous n’avons pas mis en place de parallélisation.

Avant toute chose, le code ainsi que les instructions de compilation (linux/macOS) sont disponibles sur mon GitHub : <https://github.com/FumedSaumonSauvage/ClavierGenetique>.

Clonez le dossier, compilez et lancez sans préciser d’options pour avoir une démo rapide (quelques paramètres qui avaient l’air de fonctionner, retrouvables en haut du Main).

Pour l’implémentation de la structure de données, j’ai choisi d’utiliser une classe Clavier décrivant un clavier (un tableau de lettres, essentiellement), ainsi qu’une Tribu dérant la population de claviers. Cette Tribu possède toutes les fonctions propres à être appliquées sur la population, et le Clavier possède toutes les fonctions applicables sur un seul individu à la fois.

La fonction de score utilisée pour vérifier la qualité d’un clavier (*fitness*) est basique, elle renvoie la somme des distances entre les touches pondérées par le poids des bigrammes.

Nous avons inversé le résultat pour que meilleur sera le clavier, plus haut sera le score (pour pouvoir parler de *highscore*).

Passons à l’exécution : en ce qui concerne l’initialisation, la population est initialisée aléatoirement : tous les claviers sont dotés de 26 touches et d’emplacements vides, de manière à générer des claviers corrects et complets. Cette initialisation aléatoire a posé quelques problèmes avec le générateur aléatoire de nombre de stlib.h, ici comme par la suite nous avons dû utiliser quelques astuces pour augmenter l’entropie et éviter des claviers tous identiques.

Ensuite vient la boucle principale de l’algorithme. Dans cette boucle (qui itèrera 1000 fois au maximum) se trouvent les étapes de modification des claviers :

* Sélection des meilleurs : la tribu est triée par ordre de score décroissant (donc, le meilleur en premier).
* Croisements : l’heuristique de croisements est ici assez simple mais semble efficace, elle est inspirée de la biologie (comme le principe général de l’algoritme). Lors du croisement de deux parents, nous générons deux enfants par principe de remplacement de la population. Ces deux enfants sont composés comme suit : si une lettre est à la même place chez ses deux parents, elle est considérée comme dominante et sera placée au même endroit chez la descendance. Pour le reste des lettres (les lettres dont les positions ne sont pas communes aux deux parents), nous choisissons pour chaque enfant *différemment* une partie des lettres au hasard qui viendra du premier parent, et une partie qui viendra du second parent.

Cette méthode de croisement entraine souvent des erreurs : les enfants ne sont généralement ni corrects ni complets, car dans le processus de récupération d’une lettre sur deux d’un parent et de l’autre, certaines seront en double et d’autre n’apparaitront pas.

Nous avons donc introduit un mécanisme de correction d’erreurs : les claviers descendants sont vérifiés intégralement et corrigés comme suit *avant même leur introduction dans la tribu.* S’il manque une lettre, elle est ajoutée dans un espace vide au hasard sur le clavier, et si une lettre est en double une de ses occurrences est supprimée.

Cette méthode de correction entraine un pau plus de mutations que prévu, mais évite des incompatibilités grossières entre les parents lors du croisement.

* Mutation : lors de cette étape, nous sélectionnons une partie de la population (paramètre variable), et nous les considérons comme apte à la mutation. Parmis ces individus sélectionnés, nous définissons pour chacun d’eux un nombre de mutations entre 0 et le nombre de mutations consécutives maximum (lui aussi paramètre variable).

Nous faisons ainsi muter les individus concernés, ce qui revient à permuter deux touches. Nous avons là aussi dû utiliser des astuces pour augmenter l’entropie du générateur d’aléa, sans quoi la même mutation risquait d’apparaître sur toute la population.

* Limitation de la consanguinité : La consanguinité fut un problème récurrent, auquel il a fallu remédier car le taux de mutation ne permettait pas d’en sortir simplement.

Dans notre modèle, des que la portion de la population autorisée à se reproduire était trop haute, elle finissait par se stabiliser en des « champions locaux » bloqués dans un maxima local. Ces champions étaient les seuls à pouvoir se reproduire, et finissaient par devenir exactement les mêmes. Le pool génétique finissait donc par se réduire sans être particulièrement performant.

Pour contrer ce problème, une limite de la proportion de consanguins a été fixée (paramètre variable), et tout clavier strictement identique à un autre en dehors de cette limite se verra muter au moins une fois de manière à le rendre différent.

La limitation des consanguins ne se fait qu’au niveau de la descendance, car ce sont les plus propices à en être atteint (les autres ne se reproduisent pas.

Cette mesure n’est actuellement qu’à moitié satisfaisante, nous trouverons peut être un meilleur moyen de l’implémenter.

Pour constater l’état du système en temps réel, un affichage rudimentaire sur le teminal a été construit.

Fonction d’arrêt : voir Dan.

Finalement, l’exécution rend un résultat assez prometteur : même si les individus sortants sont un peu meilleurs avec une population de 500 qu’avec 50, la différence n’est pas notable et le format général optimisé est respecté (un paquet de touches au centre du clavier).