Compléments d'informatique

Projet 1 : Algorithmes génétiques

26 octobre 2021

Dans ce projet, on vous propose d'implémenter une petite librairie mettant en œuvre les algorithmes génétiques, qui sont des algorithmes heuristiques permettant de résoudre des problèmes d'optimisation pour lesquels il n'existe pas de solution efficace. Ces algorithmes ne permettent pas d'obtenir une solution optimale à ces problèmes mais permettent pour certains problèmes d'atteindre une solution proche de l'optimal en un temps raisonnable. On appliquera cette librairie pour traiter le problème du voyageur de commerce. Du point de vue de la programmation, l'objectif est de vous apprendre à organiser un programme réparti en plusieurs fichiers, en vous basant sur la plupart des concepts vus au cours (organisation de programmes, type opaque, pointeurs de fonctions, etc.), ainsi que de vous faire faire un peu d'algorithmique pour écrire un code efficace.

Ce projet est à réaliser par **groupe de deux étudiants maximum**. La date limite de remise du projet est indiquée dans Ecampus.

1 Algorithmes génétiques

Les algorithmes génétiques sont des algorithmes d'optimisation basés sur l'idée de sélection naturelle. Ils font partie de la famille des algorithmes évolutionnaires ou encore des algorithmes bioinspirés.

Motivation. Soit une fonction $f : \mathcal{X} \to \mathbb{R}^+$, représentant la fonction à optimiser sur un ensemble $\mathcal{X}$, généralement supposé fini. On cherche donc à trouver $x^* = \arg\max_{x \in \mathcal{X}} f(x)$. L'idée générale des algorithmes génétiques est de faire évoluer une ensemble de solutions $P = \{x_1, x_2, \ldots, x_n\}$ de manière à ce que cet ensemble contienne des solutions de qualités croissantes (mesurées par la valeur de f). On appellera f la fonction d'évaluation (ou de *fitness* en anglais), l'ensemble de solutions P la *population* et chaque solution candidate x_i de la population un *individu*. Chaque individu est supposé codé par une séquence finie d'entiers de longueur quelconque mais fixe. Par analogie avec la génétique, on appelle cette

séquence le *génotype* et chaque valeur de cette séquence un *gène*. Par exemple, si l'ensemble $\mathcal{X}$ est l'ensemble des entiers de 0 à 31, on codera les individus par des séquences binaires de longueur 5.

Principe général. La forme générale d'un algorithme génétique est la suivante :

1. Générer une population *initiale* de n individus, $P = \{x_1, \ldots, x_n\}$.
2. Répéter t fois :
   (a) Générer une nouvelle population $P' = \{x'_1, \ldots, x'_n\}$ à partir de P.
   (b) $P \leftarrow P'$

Il existe énormément de manières de générer une nouvelle population à partir de la précédente (étape 2.(a)). Dans ce projet, on vous demande d'implémenter la procédure suivante en deux étapes :

1. Sélection et recombinaison :
   (a) $x'_1, \ldots, x'_k$ sont les k individus de P de valeurs de la fonction f les plus élevées, avec x'_1 l'individu de fitness le plus élevé de P [1].
   (b) Chaque individu x'_i, avec $k < i \leq n$, est obtenu en appliquant un opérateur de *recombinaison* à deux individus, appelés les parents, *sélectionnés* aléatoirement dans la population selon leurs valeurs de fitness.
2. Mutation : chaque individu, sauf le premier x'_1 [2], est modifié aléatoirement en lui appliquant une opération de *mutation*.

Une instanciation de l'algorithme requiert la définition de 4 opérateurs pour (1) la création des individus de la population initiale, (2) la sélection des parents, (3) la recombinaison entre ces parents et (4) la mutation. Nous décrivons ci-dessous l'opération de sélection (qui est générique), ainsi que les trois autres opérations dans le cas où les individus sont représentés par des séquences d'entiers non contraintes. Le cas du voyageur de commerce sera traité dans la section 2.

Sélection des parents. L'idée de la sélection, comme la sélection naturelle, est de favoriser les parents de valeurs de f les plus élevées. L'approche la plus classique est de sélectionner chaque parent avec une probabilité proportionnelle à sa valeur de fitness. Plus précisément, la probabilité de tirer x_i comme parent est donnée par $\frac{f(x_i)}{\sum_{k=1}^{n} f(x_k)}$. Pour implémenter ce tirage, on procédera comme suit : on tire aléatoirement un nombre réel r uniformémement dans $[0,0; 1,0[$. L'individu sélectionné est alors l'individu $i \in \{1, \ldots, n\}$ tel que $g_{i-1} \leq r < g_i$ avec $g_i = \sum_{j=1}^{i} f(x_j) / \sum_{k=1}^{n} f(x_k)$ ($g_0 = 0$). Seule cette approche de sélection sera utilisée dans ce projet, même si d'autres techniques existent.

1. L'ordre des autres éléments n'a pas d'importance.
2. Cela assure que le meilleur individu est toujours reproduit tel quel dans la nouvelle population.

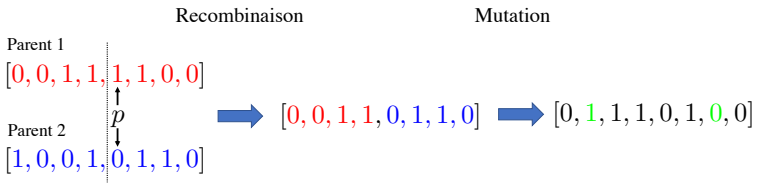

FIGURE 1 – Illustration de la recombinaison et de la mutation dans le cas d'un génotype non contraint.

Initialisation. S'il n'y a pas de contraintes sur les séquences représentant les individus, la manière la plus simple de générer la population initiale est de générer chaque individu comme une séquence aléatoire d'entiers de la longueur désirée.

Recombinaison. La recombinaison vise à déterminer le génotype d'un individu fils comme la combinaison des génotypes de ses deux parents, de manière à ce qu'il hérite d'une partie de leurs gènes. Les parents étant choisis sur base de leur valeur de fitness, l'espoir est de créer ainsi un fils de valeur potentiellement supérieure à ses parents. Soit l la longueur de génotype des individus et soient les génotypes de deux parents $[s_1^1, s_2^1, \ldots, s_l^1]$ et $[s_1^2, s_2^2, \ldots, s_l^2]$ et du fils $[s_1, s_2, \ldots, s_l]$. L'opérateur le plus classique de recombinaison consiste à déterminer une position aléatoire p dans $[1, \ldots, l+1]$ et à attribuer au fils la valeur du génotype de son premier parent jusqu'à la position $p-1$ et la valeur du génotype de son deuxième parent à partir de la position p, c'est-à-dire $s_i = s_i^1$ pour $0 \leq i \leq p-1$ et $s_i = s_i^2$ pour $p \leq i \leq l$.

Mutation. L'opération de mutation consiste à introduire des modifications aléatoires ponctuelles sur les génotypes de manière à introduire plus de diversité dans la population et ainsi empêcher l'algorithme de converger trop vite vers un maximum local. L'opérateur le plus classique dans le cas de séquences non contraintes consiste à changer chaque gène de la séquence pour une valeur aléatoire avec une probabilité de mutation p_m, généralement assez faible.

Une séquence de recombinaison et de mutation est illustrée à la figure 1 dans le cas de séquences binaires.

2 Problème du voyage de commerce

Le problème du voyage de commerce est un problème d'optimisation très populaire et aux applications nombreuses. Malgré cela, personne n'a encore trouvé d'algorithme efficace (c'est-à-dire de complexité non exponentielle) pour le résoudre et il y a peu de chance qu'on

y parvienne. Les algorithmes génétiques constituent donc une approche intéressante pour trouver une solution approchée en un temps raisonnable.

Formalisation du problème. Soit une liste $\{v_1, \ldots, v_N\}$ de N villes représentées par leurs coordonnées (x_j, y_j) (avec $j \in \{1, \ldots, N\}$). On aimerait trouver un ordre de visite de ces villes qui minimise la longueur du trajet total, en supposant qu'on puisse se déplacer en ligne droite entre deux villes et qu'on souhaite revenir à sa ville de départ une fois la dernière ville visitée. Soit $(v_{i_1}, \ldots, v_{i_N})$ un tel ordre avec $(i_1, i_2, \ldots, i_N)$ une permutation de $(1, 2, \ldots, N)$. La longueur du tour sera :

$$\sum_{k=1}^{N-1} \sqrt{(x_{i_k} - x_{i_{k+1}})^2 + (y_{i_k} - y_{i_{k+1}})^2} + \sqrt{(x_{i_N} - x_{i_1})^2 + (y_{i_N} - y_{i_1})^2}, \tag{1}$$

où les deux derniers termes correspondent à la longueur du chemin pour revenir à la première ville.

Fonction fitness et génotypes. Le problème tel que formalisé ci-dessus correspond à un problème de minimisation d'une fonction (la longueur du chemin) sur l'ensemble de toutes les permutations possible des N villes. Pour aborder ce problème via les algorithmes génétiques, il est nécessaire de définir une fonction fitness positive à maximiser et de déterminer une manière d'encoder les solutions sous la forme de séquences d'entiers. Pour la fonction fitness, nous utiliserons simplement l'inverse de la longueur du chemin.

Il semble naturel d'utiliser comme génotype pour nos individus directement une permutation $(v_{i_1}, \ldots, v_{i_N})$ des N villes. Si on dénote chaque ville par un entier entre 0 et $N-1$, cet encodage correspond bien à des séquences d'entiers de longueur $l = N$ fixe. Cependant, cette séquence est maintenant contrainte et si on applique les opérateurs de recombinaison et de mutation tels que décrits ci-dessus, on obtiendra des individus dont le génotype pourrait contenir plusieurs fois la même ville et ne correspondrait donc plus à une permutation des villes de départ. Pour appliquer les algorithmes génétiques à ce problème, il est donc nécessaire de redéfinir les opérateurs pour que les génotypes générés correspondent toujours à des permutations des entiers de 0 à $N-1$. L'instanciation particulière de ces opérateurs que nous vous demandons d'implémenter est décrite ci-dessous, mais de nombreux autres choix auraient été possibles.

Initialisation. Les génotypes des individus de la première population devront correspondre à des permutations aléatoires des entiers de 0 à $N-1$.

Recombinaison. L'opérateur qu'on vous demande d'implémenter s'appelle le "partial matching crossover" (PMC). L'idée est de choisir deux positions au hasard p_1 et p_2 ($\geq p_1$) entre 0 et $N-1$. Le génotype du fils est d'abord recopié entièrement de celui du premier

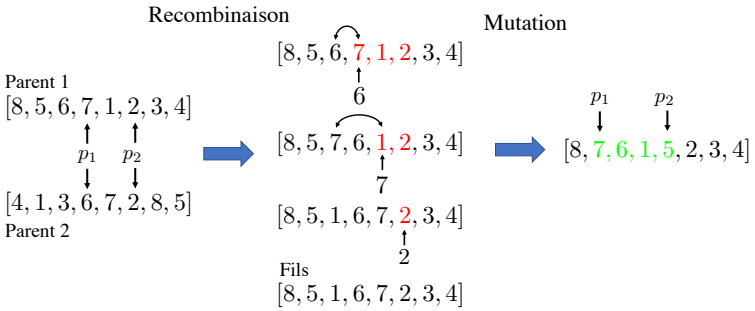

FIGURE 2 – Illustration de la recombinaison et de la mutation dans le cas d'un génotype de type permutation. contraint.

parent. Ensuite, les gènes entre les positions p_1 et p_2 (comprises) sont recopiés, un par un et de gauche à droite, du génotype du second parent. Pour maintenir la contrainte de permutation, la ville à la position p qui doit être écrasée par la nouvelle ville c venant du parent 2 est placée à la position de la ville c dans le génotype actuel du fils. Cette procédure est illustrée à la figure 2.

Mutation. L'opération de mutation consiste à prendre deux positions p_1 et p_2 ($\geq p_1$) dans le génotype au hasard entre 0 et $N-1$ (comme pour la recombinaison) et ensuite à renverser la liste des gènes (villes) entre ces deux positions. On n'effectuera cette mutation sur chaque individu qu'avec une probabilité p_m.

3 Implémentation

Votre implémentation demandera la création de trois modules :

- Un module constitué des fichiers `individual.c` et `individual.h` implémentant la structure et les fonctions nécessaires à représenter et manipuler des individus et leur génotype.
- Un module constitué des fichiers `population.c` et `population.h` implémentant la structure et les fonctions nécessaires à la manipulation d'une population.
- Un module constitué des fichiers `tsp.c` et `tsp.h` implémentant l'algorithme de résolution du voyageur de commerce sur base de l'algorithme génétique.

Nous vous fournissons également un fichier `testga.c` permettant d'effectuer différents tests de votre implémentation.

3.1 Fichiers `individual.h` et `individual.c`

Ce module doit définir une structure abstraite de type `Individual` qui servira à contenir les informations utiles pour représenter un individu (principalement son génotype). Les

opérateurs d'initialisation, de recombinaison et de mutation seront à implémenter dans ce fichier.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `individual.c` sont les suivantes :

`Individual *individualCreate(int length, int nbVal)` : crée un individu dont le génotype aura une longueur `length` et sera constitué d'entiers de valeurs comprises entre 0 et `nbVal`-1.

`void individualFree(Individual *ind)` : libère la mémoire prise par l'individu.

`int individualGetLength(Individual *ind)` : renvoie la longueur du génotype de l'individu.

`int individualGetGene(Individual *ind, int i)` : renvoie la valeur du gène à la position `i` de l'individu.

`void individualSetGene(Individual *ind, int i, int val)` : met la valeur du gène à la position `i` à `val`.

`void individualRandomInit(Individual *ind)` : Initialise le génotype de l'individu à une valeur aléatoire (voir la section 1).

`void individualRandomPermInit(Individual *ind)` : Initialise le génotype de l'individu à une permutation aléatoire des valeurs entre 0 et `nbVal-1` (voir la section 2).

`Individual *individualCopy(Individual *ind)` : crée et renvoie un nouvel individu qui est une copie parfaite de l'individu `ind`.

`void individualPrint(FILE *fp, Individual *ind)` : affiche le génotype de l'individu dans le fichier `fp` (fonction `fprintf`). Le format attendu est une liste des gènes (entiers) séparées par des virgules, sans retour à la ligne.

`void individualSeqMutation(Individual *ind, float pm)` : implémente l'opérateur de mutation décrit dans la section 1 pour une séquence non contrainte.

`Individual *individualSeqCrossOver(Individual *parent1, Individual *parent2)`: implémente l'opérateur de recombinaison décrit dans la section 1 pour une séquence non contrainte.

`void individualPermMutation(Individual *ind, float pm)` : implémente l'opérateur de mutation décrit dans la section 2 pour une permutation d'entiers.

`Individual *individualPermCrossOver(Individual *parent1, Individual *parent2)`: implémente l'opérateur de recombinaison décrit dans la section 2 pour une permutation d'entiers.

Contraintes de complexité. Pour obtenir tous les points, la complexité des fonctions de mutation et de recombinaison ne doit pas dépasser $O(l)$ dans le pire cas, où l est la longueur des génotypes.

3.2 Fichiers `population.h` et `population.c`

Ce module définit le type de données abstrait `Population` utilisé pour définir une population d'individus et implémente les fonctions de manipulation de cette population, qui représente le cœur de l'algorithme génétique.

Fonctions de l'interface. Les fonctions à implémenter dans le fichier `population.c` sont les suivantes :

```
Population *populationInit(int length, int nbVal, int size,
                           double (*fitness)(Individual *, void *),
                           void *paramFitness,
                           void (*init)(Individual *),
                           int (*mutation)(Individual *, float),
                           float pmutation,
                           Individual* (*crossover)(Individual *, Individual *),
                           int eliteSize)}:
```

crée une population de `size` individus avec des génotypes de longueur `length` et de valeurs de gènes comprises entre 0 et `nbVal`-1 initialisés par la fonction `init` fournie en argument. Cette population sera amenée à évoluer ensuite en utilisant les opérateurs de mutations définis par les fonctions `mutation` et `crossover` fournies en arguments, avec une probabilité de mutation `pmutation` et le paramètre `eliteSize` (on supposera qu'il sera toujours ≥ 1). `paramFitness` est un pointeur `void` permettant de passer des paramètres à la fonction `fitness`.

`double populationGetMaxFitness(Population *pop)` : renvoie le fitness maximum parmi tous les individus de la population.
`double populationGetMinFitness(Population *pop)` : renvoie le fitness minimum parmi tous les individus de la population.
`double populationGetAvgFitness(Population *pop)` : renvoie le fitness moyen parmi

tous les individus de la population.

`Individual *populationGetBestIndividual(Population *pop)` : renvoie le meilleur individu de la population.

`void populationFree(Population *pop)` : libère la mémoire prise par la population.

`Individual *populationSelection(Population *pop)` : tire un individu au hasard dans la population selon les valeurs de fitness (voir section 1).

`void populationEvolve(Population *pop)` : effectue une itération d'évolution de la population selon l'algorithme de la section 1. Cette fonction doit appeler `populationSelection` pour la sélection des parents pour la recombinaison.

Contraintes de complexité. Pour obtenir tous les points, la complexité de la fonction `populationSelection` ne doit pas excéder $O(\log(n))$ dans le pire cas et la fonction `populationEvolve` ne doit pas excéder $O(n \log(n))$, où n est la taille de la population.

3.3 Fichiers `tsp.h` et `tsp.c`

Ce module contient l'implémentation des fonctions permettant d'appliquer les algorithmes génétiques pour résoudre le problème du voyageur de commerce. Un type de données de type `Map` est défini dans le fichier `tsp.c` pour representer les coordonnées des villes. Cette structure contient le nombre de villes et les coordonnées x et y de chacune des villes. Une fonction vous est fournie pour charger les coordonnées dans un fichier et pour visualiser graphiquement un tour.

Fonctions de l'interface. Les fonctions suivantes vous sont fournies :

`Map *tspLoadMapFromFile(char *filename, int nbTowns)` : charge un fichier csv, appelé `filename`, contenant les coordonnées d'un ensemble de villes. Seules les `nbTowns` premières villes sont chargées du fichier. Renvoie un pointeur vers une structure de type `Map` contenant les coordonnées des villes chargées du fichier.

`void tspFreeMap(Map *map)` : libére la mémoire prise par la structure `map`.

`void tspTourToGIF(int *tour, Map *map, const char *gifName, int size)` : crée un fichier au format d'image `gif` appelé `gifName` contenant une représentation du tour donné en argument en se servant des coordonnées de la structure `map`. `tour` est un simple tableau d'entiers contenant les indices de villes dans l'ordre du tour et `size` est la largeur de l'image générée en nombre de pixels.

Dans ce fichier, vous devrez implémenter vous même les deux fonctions suivantes :

`double tspGetTourLength(int *tour, Map *map)` : renvoie la longueur du tour donné en argument selon les coordonnées de villes dans la structure `map`.

`int *tspOptimizeByGA(Map *map, int nbIterations, int sizePopulation, int eliteSize, float pmutation, int verbose)` : renvoie le meilleur tour déterminé par algorithme génétique selon le schéma expliqué dans la section 1 avec les opérateurs définis dans la section 2. L'algorithme doit être exécuté pendant `nbIterations` (t), avec une population de taille `sizePopulation` (n), en conservant à chaque évolution les `eliteSize` meilleurs individus (k) et avec une probabilité de mutation `pmutation` (p_m). Si le paramètre `verbose` est égal à 0, aucun affichage ne doit être fait à l'écran. S'il est égal à 1, vous êtes libres d'afficher ce que vous souhaitez à l'écran pour votre débogage (par exemple, l'évolution de la longueur minimale du tour au cours des itérations).

3.4 Fichier `testga.c`

Ce fichier vous permet de tester votre implémentation. Trois applications sont implémentées dans le fichier :

- La recherche de l'optimum d'une fonction fitness comptant le nombre de 1 dans une séquence binaire de longueur l. Cette application est purement artificielle et permet de vérifier que votre algorithme fonctionne correctement.
- La recherche de l'optimum de la fonction $f(x) = -\frac{x^2}{10} + 3x + 4$ où x est une valeur entière entre 0 et 31, codé par une séquence de 5 bits.
- Le problème du voyageur de commerce.

Vous pouvez exécuter l'un ou l'autre de cette application en ajoutant comme premier argument sur la ligne de commande 1, 2 ou 3. Dans le cas du problème du voyageur de commerce, vous devez ajouter sur la ligne de commande les arguments suivants :

```
file ntowns size pm elitesize niterations
```

où `file` est un fichier contenant des coordonnées de villes, `ntowns` est le nombre de villes récupérées dans le fichier, `size` est la taille de la population, `pm` la probabilité de mutation et `elitesize` le nombre k des meilleurs individus conservés à chaque évolution, et `niterations` le nombre d'évolution à effectuer. A chaque exécution un fichier `tour.gif` est créé (qui écrase le précédent) reprenant le meilleur tour obtenu à l'issue de l'exécution de l'algorithme génétique. Deux fichiers de villes vous sont fournis :

- `xy-random.csv` : contenant des coordonnées de 1000 villes placées aléatoirement.
- `xy-belgium.csv` : contenant les coordonnées de 589 villes belges.

4 Conseils d'implémentation

Commencez par implémenter les fonctions du fichier `individual.c`, excepté les fonctions `IndividualPermCrossOver` et `IndividualPermMutation`. Les autres fonctions ne devraient pas vous poser de problème particulier. Testez votre implémentation en vous servant de la fonction `individualPrint` permettant d'afficher les génotypes.

Implémentez ensuite les fonctions du fichier `population.c`. Les fonctions les plus complexes de ce fichier sont les fonctions `populationSelection` et `populationEvolve`. Pour obtenir au pire cas $O(\log n)$ pour la première fonction, vous devez imaginer une solution basée sur le diviser pour régner et également vous arranger pour précalculer les sommes cumulées des valeurs de fitness des individus de la population. La principale difficulté de `populationEvolve` est d'identifier les k individus de la population ayant les valeurs de fitness les plus élevées. Une manière de le faire est de trier au préalable les valeurs de fitness. Vous pouvez pour cela utiliser la fonction `qsort` définie dans `stdlib.h`. Renseignez vous sur la manière de l'utiliser. Une fois l'implémentation de ces fonctions terminées, vous pouvez tester votre implémentation pour résoudre les problèmes 1 et 2 du fichier `testga.c`.

Implémentez ensuite les fonctions `individualPermCrossOver` et `individualPermMutation` et puis les fonctions du fichier `tsp.c`. `individualPermMutation` et les fonctions de `tsp.c` ne devraient pas vous poser problème mais il n'est pas trivial d'obtenir une complexité $O(l)$ pour `individualPermCrossOver`. Vous devez trouver un mécanisme pour ne pas devoir faire une recherche linéaire pour déterminer les villes à échanger lors du recopiage des gènes du deuxième parent.

5 Rapport et expériences

Le fichier `rapport.txt` fourni contient des questions courtes auxquelles on vous demande de répondre directement dans le fichier (sans modifier la mise en page). En particulier, on vous demande de préciser les complexités de certaines fonctions. Pour vous encourager à utiliser votre programme, on vous invite également à l'utiliser pour essayer de déterminer le meilleur tour possible des 200 premières villes du fichier `xy-belgium.txt`. Testez différentes combinaisons des paramètres de l'algorithme de votre choix et rapportez dans `rapport.txt` la longueur minimale obtenue et les valeurs de paramètres vous ayant permis d'obtenir cette solution. Enregistrez également le tour obtenu dans un fichier `tour.txt` (en utilisant la fonction `individualPrint`) que vous soumettrez avec votre projet.

6 Soumission

Le projet doit être soumis via la plateforme de soumission sous la forme d'une archive au format `zip` contenant les fichiers suivants :

— `individual.c`

— `population.c`
— `tsp.c`
— `rapport.txt`
— `tour.txt`

Vos fichiers seront compilés et testés sur la plateforme de soumission en utilisant le fichier `Makefile` fourni via la commande `make all` ou de manière équivalente en utilisant la commande suivante :

```
gcc -o testga testga.c individual.c population.c tsp.c easyppm.c --std=c99 -lm
```

En outre, nous utiliserons les flags de compilation habituels (`-pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter négativement la cote.

Important : Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, l'étudiant (ou le groupe) se verra affecter une cote nulle à l'ensemble des projets.

Bon travail !