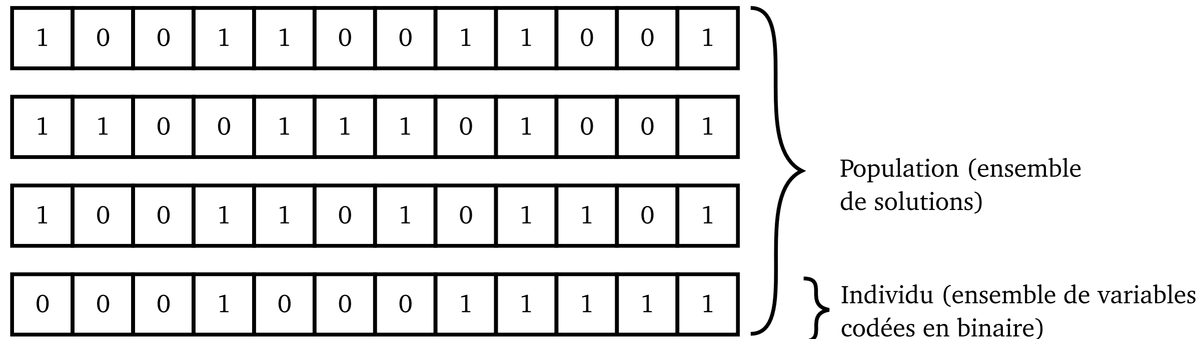
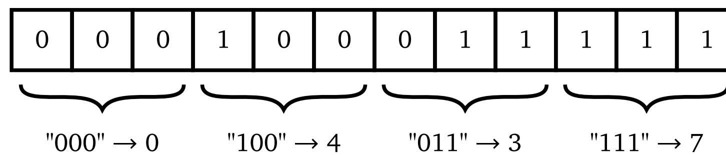
Problème 1. Algorithmes génétiques

Les algorithmes génétiques (AG) sont une méthode d'optimisation imaginée par John Henry Holland dans les années 70. Ils appartiennent à la famille des méthodes d'optimisation communément appelées méta heuristiques dont le principe fondamental est « mieux vaut une solution proche de l'optimum obtenu dans un temps raisonnable que l'optimum obtenu dans un temps irréaliste ». Leur fonctionnement des AG dans leur forme canonique est relativement simple et se base sur le principe de sélection naturelle énoncée par théorie de l'évolution du naturaliste Charles Darwin.

La première étape de l'algorithme consiste à générer de manière aléatoire un ensemble de solutions décrites sous forme de chaînes binaires. Chacune d'elles décrit une solution potentielle au problème posé. Un exemple est donné dans la figure ci-dessous.



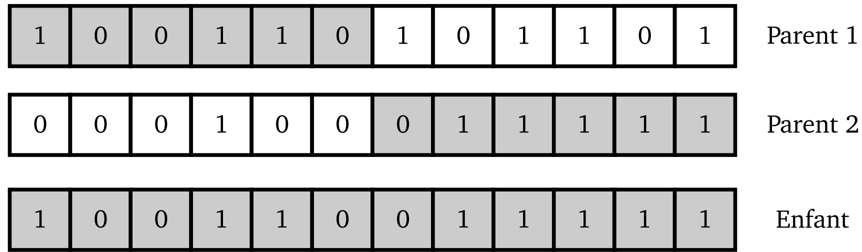
Elles sont tour à tour décodées, évaluées puis classées en fonction de la qualité de leur réponse au problème comme cela est présenté dans l'exemple ci-dessous avec une solution comportant quatre variables définies sur l'intervalle [0,7].



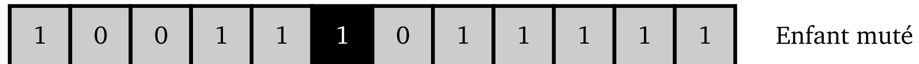
Les meilleures solutions (celles dont l'évaluation par la fonction d'évaluation $f(x)$ renvoie la plus petite valeur dans le cas dans processus de minimisation d'erreur) sont conservées en vue de produire de nouvelles solutions qui vont remplacer les anciennes.

Descriptions des solutions												Évaluation	Sélection
1	0	0	1	1	0	0	1	1	0	0	1	$f(x) = 2$	OUI
1	1	0	0	1	1	1	0	1	0	0	1	$f(x) = 3$	NON
1	0	0	1	1	0	1	0	1	1	0	1	$f(x) = 1$	OUI
0	0	0	1	0	0	0	1	1	1	1	1	$f(x) = 8$	NON

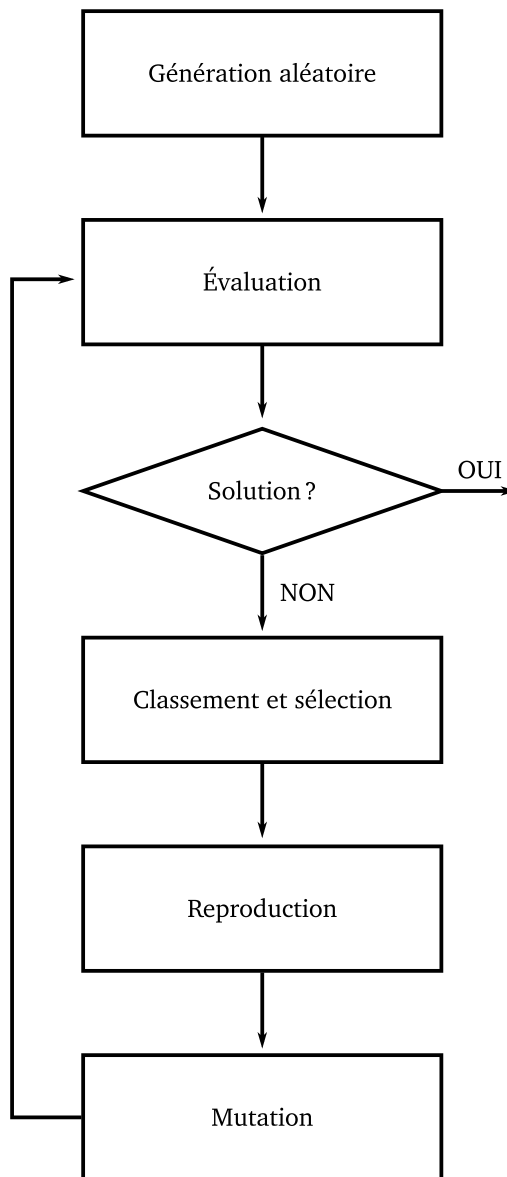
Cette production de nouvelles solutions se base sur le principe suivant : « utiliser les description des solutions sélectionnées pour en produire de nouvelles ». Ce principe est illustré dans la figure ci-dessous.



Afin d'éviter un blocage de l'évolution (dans le cas où les enfants sont moins bons que leurs parents) des mutations sont réalisées sur les nouvelles solutions proposées comme cela est présenté sur la figure ci-dessous.



Ce processus itératif est répété jusqu'à l'obtention d'une solution acceptable résumé dans la figure suivante.



1.1 Écriture de l'algorithme génétique

L'objectif de cette première partie est de réaliser les fonctions implantant le processus d'algorithme génétique décrit précédemment.

Question 1.1. Génération de solutions aléatoires

(a) Déterminer le nombre n de bits nécessaires pour représenter une solution composée de k variables pouvant chacune prendre m valeurs.

(b) Définir une fonction `créer_solution` prenant en paramètre une longueur n de bits et retourne une chaîne `C` comportant n bits tirés de manière aléatoire.

(c) Définir une fonction `génération_aléatoire` prenant en paramètre un nombre n de solutions potentielles et une longueur m et génère aléatoirement une liste `L` de taille n comportant des chaînes de taille m.

Question 1.2. Évaluation

(a) Définir une fonction `évaluation` qui prend en paramètre une liste de solutions potentielles `L` et une fonction d'évaluation `f` (cette fonction prend elle-même un unique paramètre `C` qui représente une solution potentielle et renvoie l'évaluation de cette solution), et qui renvoie une liste d'évaluation de taille égale au nombre de solutions évaluées. Par exemple avec `f` qui effectue simplement la somme des bits :

```
>>> f('101001001')
4
>>> évaluation(['101001001','101001111','100100100'], f)
[4, 6, 3]
```

(b) Définir une fonction `classer_solutions` qui prend en paramètre une liste `L_sol` de solutions et une liste `L_eval` d'évaluations et qui retourne la liste des descriptions classées de manière croissante en fonction de leurs évaluations.

Question 1.3. Sélection

(a) Définir une fonction `sélection_simple` qui prend en paramètre une liste `L_sol` de solutions décrites sous forme de chaînes de caractère et retourne une solution choisie de manière aléatoire parmi la moitié des solutions ayant la plus petite évaluation. On considère que ces différentes chaînes sont préalablement triées par évaluation croissante.

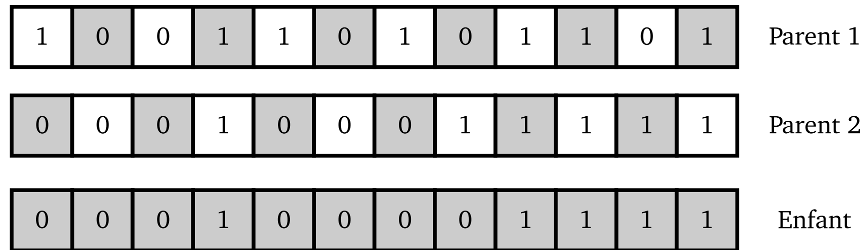
(b) Écrire une fonction `tirage` qui prend en entrée une liste d'objets `L` et une liste de probabilités `p` (de même taille) telle que la somme des probabilités dans `p` vaut 1, et qui renvoie un élément aléatoire de `L` de telle sorte que `L[i]` est choisi avec probabilité `p[i]`. *Indication : si on tire un réel aléatoire r entre* 0 *et* 1*, la probabilité que r appartienne à un intervalle $[a, b]$ est $(b-a)$.*

(c) Définir une fonction `sélection_aléatoire` qui prend en paramètre une liste `L_sol` de solutions et une liste `L_eval` décrivant les évaluations de chacune de ces solutions, et renvoie une solution tirée de manière aléatoire, de telle sorte que la probabilité pour une solution d'être choisie soit inversement proportionnelle à son évaluation :

$$\mathbb{P}[\text{solution } s_i \text{ choisie}] = \frac{1/f(s_i)}{\sum_{i=1}^{n} 1/f(x_i)}.$$

(d) Définir une fonction `sélection` qui prend en paramètre une liste `L_sol` de solutions et leurs évaluations `L_eval`, et renvoie une liste de taille moitié par rapport à `L_sol`, construite en tirant aléatoirement des solutions avec la distribution de la question précédente. *Attention, la liste renvoyée ne doit pas contenir deux fois la même solution.*

Question 1.4. Reproduction On souhaite effectuer le croisement de deux solutions en prenant un bit sur deux de chaque parent en commençant par le parent 2, comme dans la figure ci-dessous.



- **(a)** Définir une fonction `reproduction_croisement_iter` qui prend en entrée deux chaînes binaires `C1` et `C2` et renvoie leur croisement comme décrit ci-dessus. **Cette fonction doit être écrite de manière itérative.**
- **(b)** Définir une fonction `reproduction_croisement_rec` ayant les mêmes spécifications que la précédente, mais **écrite de manière récursive.**
- **(c)** Définir une fonction `reproduction_aléatoire` qui prend en entrée deux chaînes binaires `C1` et `C2` et qui renvoie une chaîne obtenue en prenant pour chaque bit, soit le bit de `C1` soit le bit de `C2`, aléatoirement.
- **(d)** Étant données deux chaînes `C1` et `C2`, calculer le nombre minimal et le nombre maximal de chaînes différentes qui peuvent être produites par `reproduction_aléatoire`.
- **(e)** Définir une fonction `reproduction` qui prend en entrée une liste `L` et renvoie une liste qui contient tous les éléments de `L`, plus autant d'éléments contruits par reproduction aléatoire : pour créer un nouvel élément, on choisit aléatoirement deux éléments de `L` et on effectue leur reproduction aléatoire. *La liste renvoyée a donc taille double par rapport à* `L`.

Question 1.5. Mutations

- **(a)** Définir une fonction `choix_multiple` qui prend en entrée une liste `L` et une proportion `m` (entre 0 et 1) et qui sélectionne aléatoirement une proportion t des éléments de `L` (sans doublon). *S'il n'est pas possible de sélectionner une proportion exactement t, on sélectionnera le nombre minimal d'éléments de* `L` *tel que la proportion sélectionnée soit supérieure à t.*
- **(b)** Définir une fonction `mutation_aléatoire` ayant comme entrées une chaîne binaire `C` ainsi qu'un taux de mutation `t` (entre 0 et 1) et qui produit en sortie une chaîne binaire dont une proportion t des bits a muté. *Remarque : chaque mutation consiste à inverser un bit.*
- **(c)** Définir une fonction `mutation` ayant comme entrées une liste `L` et un taux de mutation `t` et renvoie la liste constituée des éléments de `L` mutés avec le taux `t`.

Question 1.6. Boucle principale Écrire une fonction `algo_génétique` qui effectue la boucle générale de l'algorithme génétique, telle que décrite en introduction. Les spécifications de cette fonction sont les suivantes :

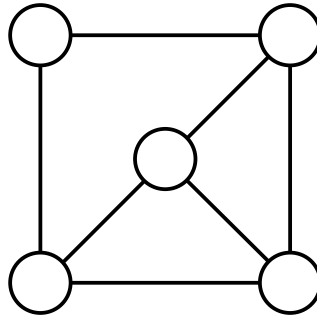
- — elle prend en entrée une taille de population n, une taille de solution m, une fonction d'évaluation f, un taux de mutation t, et un seuil s ;
- — elle applique la boucle tant qu'il n'y a pas de solution dont l'évaluation est inférieure ou égale au seuil (dès qu'une telle solution est trouvée, elle est renvoyée) ;
- — les étapes de sélection, reproduction et mutation sont effectuées avec de l'aléatoire.

1.2 Exemple d'utilisation

Le problème de coloration de graphe consiste à trouver le nombre minimum de couleurs (nombre chromatique) permettant de colorer chaque nœud d'un graphe G de manière à ce que deux nœuds adjacents ne soient pas colorés par la même couleur. Un graphe G est dit k-colorable s'il est possible de le colorer avec k couleurs, en respectant la condition précédente. Dans toute la suite, on représentera les couleurs par des entiers, et une k-coloration consiste à attribuer des *couleurs* entre 0 et $k-1$ à chaque sommet.

Question 1.7.

(a) Quel est le nombre minimum de couleurs pour colorer le graphe suivant ? *Attention à ne pas oublier de montrer que c'est le nombre minimal possible.*



(b) Donner une borne supérieure sur le nombre de couleurs nécessaires pour colorer un graphe à n sommets. Cette borne est-elle atteinte ?

Question 1.8. Représentation des graphes

(a) Compléter la classe `Noeud` permettant de décrire un nœud de graphe, avec les spécifications suivantes :

- `_id` est un entier identifiant le nœud, `_couleur` représente sa couleur, et `_voisins` est la liste des identifiants de ses voisins ;
- les trois méthodes à compléter ne renvoient rien, et modifient le `Noeud` sur lequel elles sont appliquées ;
- la méthode `ajouter_voisin` prend en paramètre l'identifiant d'un `Noeud` et l'ajoute à la liste des voisins ;
- la méthode `supprimer_voisin` prend également l'identifiant d'un `Noeud` mais le supprime (s'il existe) de la liste des voisins ;
- `changer_couleur` prend en paramètre une couleur et modifie la couleur du sommet.

Il suffira d'écrire sur la copie le code des trois méthodes à compléter.

```
class Noeud:
    def __init__(self, id, c):
        self._id = id
        self._couleur = c
        self._voisins = []

    def ajouter_voisin(self, voisin):
        ...

    def supprimer_voisin(self, voisin):
        ...

    def changer_couleur(self, c):
        ...
```

(b) Définir une classe python `Graphe` permettant de représenter un graphe avec les fonctionnalités suivantes :

- un constructeur prenant comme paramètre d'entrée une liste d'objets de type `Noeud` ;
- une méthode `noeud` qui prend en paramètre un identifiant et renvoie le nœud correspondant ;
- une méthode `ajout_arête` qui prend en paramètre deux identifiants de nœuds et ajoute une arête entre eux ;
- une méthode `compter_conflits` qui compte le nombre de *conflits*, c'est-à-dire le nombre de couples de nœuds adjacents ayant la même couleur.

Question 1.9. Représentation des colorations On représente une coloration d'un graphe G par une chaîne binaire construite de la manière suivante. On fixe un ordre quelconque sur les sommets de G. On représente chaque couleur (un entier) par son écriture en base 2 sur le même nombre k de bits, et on concatène les représentations binaires des couleurs des sommets de G. Par exemple, pour un graphe à 3 sommets ayant les couleurs 0, 2 et 1 respectivement, la coloration sera représentée par la chaîne `'001001'`.

(a) Exprimer la taille de la représentation d'une coloration de n sommets avec k couleurs.

On suppose qu'on dispose d'une fonction `binaire` qui prend en paramètre un entier n et une longueur k et qui renvoie la chaîne binaire représentant n sur k bits, et une fonction `entier` qui effectue la transformation inverse.

(b) Définir une fonction `découper_chaîne` prenant en paramètre une chaîne binaire `C` ainsi qu'une taille de découpe t et qui renvoie une liste de chaînes de caractères `[C1, C2, ...]` de longueur t dont la concaténation est `C`. La fonction affichera un message d'erreur si la taille de coupe t est incompatible avec la longueur de la chaîne `C`.

(c) Définir une fonction `traduire_solution` prenant en paramètre un chaîne binaire `C`, la taille de découpe t et retournant une liste d'entiers `L` correspondant à la conversion des sous-chaînes binaires.

(d) Définir une fonction d'évaluation `eval_graphe` qui prend en entrée un graphe G et une chaîne binaire `C` et renvoie le nombre de conflits dans le graphe G si on colore les sommets à l'aide de `C`.

Question 1.10. Algorithme génétique appliqué à un `Graphe`

(a) Écrire une fonction `coloration` qui prend en entrée un graphe G et un nombre de couleurs k, et qui cherche une k-coloration de G à l'aide de l'algorithme génétique. La fonction prendra en paramètres optionnels une taille de population (par défaut 5) et un taux de mutation t (par défaut 0.2). Quelle valeur de seuil doit-on imposer ? *On supposera que k est une puissance de 2.*

Dans les deux questions suivantes, on ne demande pas d'écrire du code explicitement, mais simplement de donner les idées pour résoudre les deux problèmes.

(b) Proposer une solution pour traiter le cas où k n'est pas une puissance de 2.

(c) Comment peut-on utiliser l'algorithme génétique (éventuellement légèrement modifié) pour trouver la coloration du graphe qui utilise le moins de couleurs ?