Annexe au chapitre 9

Métaheuristiques

A9.1. Quelques définitions

- En optimisation combinatoire, théorie des graphes et théorie de la complexité, une heuristique est un algorithme qui fournit rapidement (en temps polynomial) une solution réalisable, mais pas nécessairement optimale, pour un problème d'optimisation difficile. Une heuristique, ou méthode approximative, est donc le contraire d'un algorithme exact qui trouve une solution optimale pour un problème donné. L'usage d'une heuristique est pertinente pour calculer une solution approchée d'un problème et ainsi accélérer le processus de résolution exacte.
- Les métaheuristiques forment une famille d'algorithmes d'optimisation visant à résoudre des problèmes d'optimisation difficile (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace. Ces méthodes utilisent cependant un haut niveau d'abstraction, leur permettant d'être adaptées à une large gamme de problèmes différents. Les métaheuristiques les plus connues sont la recherche avec tabous, le recuit simulé, les algorithmes génétiques et les colonies de fourmis.

A9.2. Algorithmes probabilistes

Un algorithme probabiliste est un algorithme dont le déroulement fait appel à des données tirées au hasard. Dans leur ouvrage *Algorithmique, conception et analyse*, G. **Brassard** et P. **Bratley** classent les algorithmes probabilistes en quatre catégories :

Algorithmes numériques

- Utilisés pour approcher la solution à des problèmes numériques (ex. calcul de pi, intégration numérique, etc.).
- La précision augmente avec le temps disponible.
- Certains auteurs classent ces algorithmes dans la catégorie Monte Carlo

Algorithmes de Sherwood

- Utilisés lorsqu'un algorithme déterministe fonctionne plus rapidement en moyenne qu'en pire cas.
- Ces algorithmes peuvent éliminer la différence entre bonnes et mauvaises entrées.
- Exemple : Quicksort

Algorithmes de Las Vegas

- Ces algorithmes peuvent parfois retourner un message disant qu'ils n'ont pas pu trouver la réponse.
- La probabilité d'un échec peut être rendu arbitrairement petite en répétant l'algorithme suffisamment souvent.

Algorithmes de Monte Carlo

- Ces algorithmes retournent toujours une réponse mais celle-ci n'est pas toujours juste.
- La probabilité d'obtenir une réponse correcte augmente avec le temps disponible.

Exercice A9.1

Développez et programmez une méthode basée sur l'estimation de l'aire d'un quart de cercle pour approcher la valeur de π.

Exercice A9.2

Modifiez le programme de l'exercice 9.7 qui teste si un point est à l'intérieur d'un polygone, afin d'estimer l'aire de ce polygone. Pour ce faire, vous générerez 100'000 points au hasard et calculerez le pourcentage de points « tombés » à l'intérieur du polygone.

Exercice A9.3 : Le compte est bon (2)

Nous avons vu au § 7.5 un algorithme récursif cherchant les solutions du jeu. Nous allons ici nous contenter d'une méthode naïve et peu efficace, mais facile à programmer : il s'agira de rechercher **aléatoirement** des solutions et de ne garder que celle qui se rapproche le plus du résultat demandé, ou qui l'atteint.

Données : six nombres dans une liste L et le résultat r à approcher.

```
1.  Choisir deux nombres a et b au hasard dans la liste L.
2.  Choisir une opération arithmétique (+, -, *, /) au hasard. L'opération doit
    être possible (par exemple, on ne peut pas diviser 5 par 9) et utile (il est
    inutile par exemple de multiplier un nombre par 1).
3.  Poser c := opération(a, b). Mémoriser ce calcul intermédiaire dans une chaîne
    de caractères (une string).
4.  Éliminer a et b de la liste L.
5.  Ajouter c à la liste L.
6.  SI c = r ALORS
      afficher tous les calculs intermédiaires.
      STOP
7.  SI il y a plus d'un nombre dans la liste ALORS
      aller à 1
    SINON afficher la liste des calculs intermédiaires et le résultat obtenu
```

On répétera cet algorithme des milliers de fois et on n'affichera que la meilleure solution trouvée. Programmez cet algorithme en Python.

Exercice A9.4

Takeshi **Kitano** est un acteur et un réalisateur japonais né en 1947.

Lors de l'exposition « Mathématiques, un dépaysement soudain » en 2011, il proposa un défi mathématique aux visiteurs : trouver la formule la plus courte permettant de calculer 2011 en écrivant, dans l'ordre, les premiers nombres entiers, séparés par des opérateurs (+, -, *, /, exposants, factorielle).

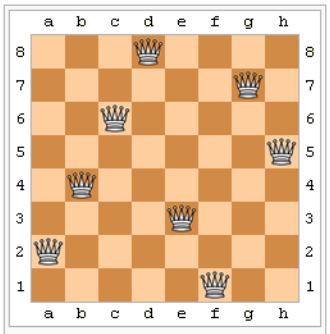
Par exemple, un « Kitano » de 2011 serait $((1+2)!)! + (3!)^4 - 5$

Modifiez l'exercice A9.3 pour trouver le « Kitano » d'une année.

A9.3. Le problème des *n* dames pour illustrer les métaheuristiques

Le problème des *n* dames est une généralisation de du problème des 8 dames, vu au § 9.2 : on considère un échiquier *n* x *n* au lieu d'un échiquier 8 x 8. Bien que ce ne soit pas à proprement parlé un problème d'optimisation, il présente de nombreux avantages :

- il est visuel et facile à comprendre ;
- on peut coder la position des dames très simplement : pour chaque colonne, on note sur quelle ligne se trouve la dame, et on lui soustrait 1. La position ci-contre sera : [1, 3, 5, 7, 2, 0, 6, 4]
- on passe très facilement d'une configuration à une configuration voisine (qui ne satisfait pas forcément les contraintes du problème) : il suffit d'échanger deux colonnes. Une position voisine de celle ci-contre pourrait être [1, 0, 5, 7, 2, 3, 6, 4].



Exemple de solution

On peut le traiter comme un problème d'optimisation si l'on considère qu'il faut minimiser le nombre de conflits (on parlera de conflit quand deux dames se menacent mutuellement). Il s'agira ici de placer *n* dames sur l'échiquier *n* x *n*, sans aucun conflit, en partant d'une solution avec une seule dame par ligne et par colonne (par exemple toutes les dames sur la diagonale) et en échangeant deux colonnes. On ne cherchera pas toutes les solutions possibles : une seule nous suffira. Notons que dans un problème d'optimisation classique, il n'y a en général qu'une seule meilleure solution. Ici il y en a plusieurs.

Exercice A9.5

Il existe un algorithme permettant de trouver une solution quel que soit *n* supérieur à 3 :

```
1. soit r = n mod 12
2. écrire les nombres pairs de 2 à n
3. si r = 3 ou r = 9 mettre le 2 à la fin de la liste
4. écrire ensuite les nombres impairs de 1 à n, mais si r=8 permuter les nombres
   impairs 2 par 2 (i.e. 3, 1, 7, 5, 11, 9, ...)
5. si r = 2, permuter les places de 1 et 3, puis mettre 5 à la fin de la liste
6. si r = 3 ou r = 9, mettre 1 puis 3 en fin de liste
```

Programmez cet algorithme en Python. Il donne une solution au problème des *n* dames. Ainsi, pour *n* = 8 on obtient la position [2, 4, 6, 8, 3, 1, 7, 5] (dans cet algorithme, les lignes sont numérotées à partir de 1). Pour *n* = 15, on aura la solution [4, 6, 8, 10, 12, 14, 2, 5, 7, 9, 11, 13, 15, 1, 3].

Vérifiez la validité de cet algorithme pour *n* = 4 ... 1000.

A9.3.1. Première approche : descente de plus grande pente

```
1. Générer une position de départ.
2. Essayer toutes les permutations de deux colonnes et échanger les deux
   colonnes qui permettent de diminuer le plus le nombre de conflits.
3. Retourner à 2 jusqu'à obtenir 0 conflit (dans l'idéal) ou un blocage.
```

Résultats avec la descente de plus grande pente

Nombre de dames (*n*)	20	40	60	80	100
Temps ou nombre de conflits restants	2 conflits	13 sec.	1 conflit	418 sec.	1308 sec.

Commentaires

- Selon la position initiale et le nombre de dames, il arrive que l'algorithme se bloque dans un minimum local. C'est arrivé ici avec 20 et 60 dames placées au départ sur la diagonale.
- Avec 100 dames, on passe d'une situation avec 4950 conflits à une configuration sans

conflit en 82 échanges de colonnes.
- Les temps sont seulement indicatifs et dépendent évidemment de l'ordinateur utilisé. Toutes les expériences ont été faites sur le même ordinateur et dans les mêmes conditions.

A9.3.2. Deuxième approche : recherche avec tabous

Fred **Glover** (né en 1937)

La méthode taboue est une métaheuristique d'optimisation présentée par Fred **Glover** en 1986. On trouve souvent l'appellation « recherche avec tabous » en français.

La méthode taboue consiste, à partir d'une position donnée, à explorer le voisinage et à choisir la position dans ce voisinage qui minimise la fonction objectif (comme dans la descente de plus grande pente). Il est essentiel de noter que cette opération peut conduire à dégrader la valeur de la fonction : c'est le cas lorsque tous les points du voisinage ont une valeur plus élevée. Le risque est qu'à l'étape suivante, on retombe dans le minimum local auquel on vient d'échapper. C'est pourquoi il faut que l'heuristique ait de la mémoire : le mécanisme consiste à interdire (d'où le nom de « tabou ») de revenir sur les dernières positions explorées.

Méthode taboue

Rappelons que l'on passe d'une position à une position voisine en échangeant deux colonnes.

```
1. Générer une position de départ.
2. Parmi les positions voisines, choisir la meilleure qui n'est pas dans la
   liste des tabous.
3. Si la liste des tabous est pleine, retirer de cette liste la position la plus
   ancienne.
4. Mettre la position actuelle en queue de la liste des tabous.
5. Retourner à 2, tant qu'on n'a pas décidé de s'arrêter.
```

Les positions déjà explorées sont conservées dans une file (voir § 6.2), souvent appelée liste des tabous, d'une taille donnée. Cette file doit conserver des positions complètes, mais cela ne pose pas de problèmes avec les *n* dames. Pour nos expériences, nous avons utilisé une liste de 10 mouvements tabous.

Résultats avec la méthode taboue

Nombre de dames (*n*)	20	40	60	80	100
Temps ou nombre de conflits restants	1 sec.	15 sec.	114 sec.	422 sec.	1756 sec.

Commentaires

- Le programme ne se bloque plus dans un minimum local.
- La solution finale est toujours la même et dépend de la position initiale.
- Avec 100 dames, on passe d'une situation avec 4950 conflits à une configuration sans conflit en 95 échanges de deux colonnes.

A9.3.3. Troisième approche : recuit simulé

Le recuit simulé (*Simulated Annealing* en anglais) est une métaheuristique inspirée d'un processus utilisé en métallurgie. Ce processus alterne des cycles de refroidissement lent et de réchauffage (recuit) qui tendent à minimiser l'énergie du matériau. Elle est aujourd'hui utilisée en optimisation pour trouver les extrema d'une fonction.

Elle a été mise au point par trois chercheurs de la société IBM, S. **Kirkpatrick**, C.D. **Gelatt** et M.P. **Vecchi** en 1983, et indépendamment par V. **Cerny** en 1985.

Recuit simulé

Remarque sur le point 4

Ici, on cherche à minimiser le score, qui est le nombre

```
1. Choisir une « température » de départ T.
2. Générer une position aléatoire. Appelons-la P1.
3. Copier la position P1 dans une position P2, puis échanger deux colonnes de P1
   choisies aléatoirement.
4. Calculer Δ = score(P2) – score(P1)
5. Si Δ ≤ 0, P1 ← P2 ; le cas échéant, mettre à jour le meilleur score et la
   meilleure position. Aller à 8.
```

de conflits.
Si on cherche à maximiser le score, il faut calculer Δ = score($P1$) – score($P2$)

```
6. Générer un nombre réel aléatoire entre 0 et 1, que nous appellerons r.
7. Si r < exp(−Δ/T), P1 ← P2.
8. Diminuer T.
9. Retourner à 3, tant qu'on n'a pas décidé de s'arrêter.
```

Le réglage des paramètres est ici plus délicat. En particulier, comment faut-il faire baisser la température ? Trop vite, on risque de se bloquer dans un minimum local. Trop lentement, le temps de calcul augmentera, sans garantie de trouver une solution.

La solution retenue a été de faire baisser la température par palier de longueur 10, avec une température initiale $T = 100$, avec $T_{k+1} = 0.6 \cdot T_k$, k représentant un numéro de palier. C'est très inhabituel, car on prend généralement un coefficient proche de 1.

Résultats avec le recuit simulé

*Il s'agit d'une moyenne sur 10 essais, puisque le hasard joue ici un rôle important. Les temps des essais où l'on n'a pas trouvé de solution n'ont pas été pris en compte.

Nombre de dames (n)	20	40	60	80	100
Temps moyen en cas de succès*	0.3 sec.	1.5 sec	4 sec.	9 sec.	23 sec.
Nombre de succès sur 10 essais	7	10	10	10	10

Commentaires

- Étant donné l'usage du hasard, on ne sait pas quelle solution finale sera trouvée. Ce ne sera pas toujours la même, contrairement à la recherche avec tabous.
- Curieusement, le recuit simulé marche le moins bien quand il y a peu de dames (7 succès seulement avec 20 dames).
- Il est par contre redoutable en temps de calcul, puisqu'il ne lui faut que 23 secondes pour trouver une solution avec 100 dames, alors que la méthode taboue en mettait 1756.

A9.3.4. Quatrième approche : algorithme génétique

John H. **Holland** (1929-2015)

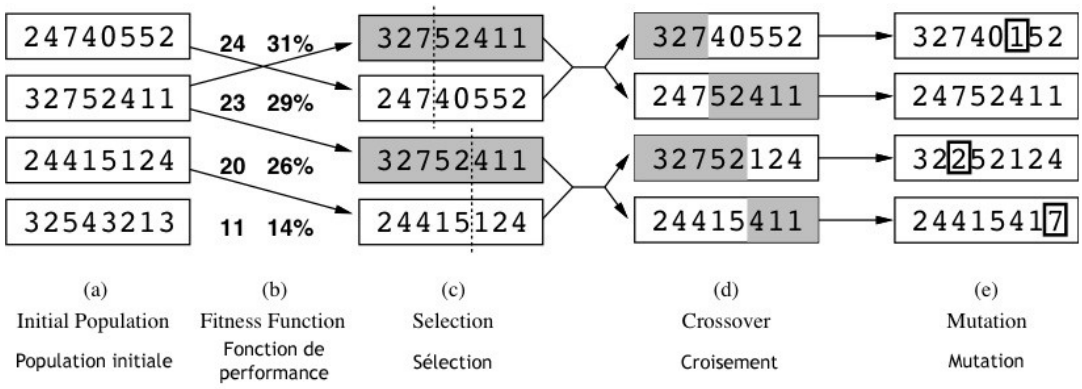
Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnistes (un sous-ensemble des métaheuristiques). Leur but est d'obtenir une solution approchée, en un temps correct, à un problème d'optimisation, lorsqu'il n'existe pas ou qu'on ne connaît pas de méthode exacte pour le résoudre en un temps raisonnable. Les algorithmes génétiques utilisent la notion de sélection naturelle développée au 19ème siècle par le célèbre scientifique **Darwin** et l'appliquent à une population de solutions potentielles au problème donné. On se rapproche par bonds successifs d'une solution.

L'utilisation d'algorithmes génétiques, dans la résolution de problèmes, est à l'origine le fruit des recherches de John Henry **Holland** et de ses collègues et élèves de l'Université du Michigan qui ont, dès 1960, travaillé sur ce sujet. Le premier aboutissement de ces recherches a été la publication en 1975 de *Adaptation in Natural and Artificial System*.

Algorithme génétique

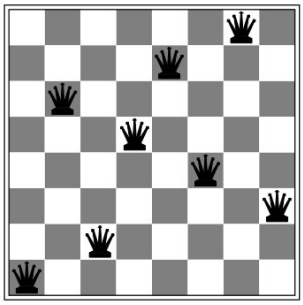
```
1. Générer une population de 2n positions aléatoires.
2. Classer et numéroter ces 2n positions selon leur score, du meilleur au moins
   bon.
3. Croiser les grilles 2k−1 et 2k, pour k allant de 1 à n.
4. Effectuer une mutation pour chaque position : avec une certaine probabilité,
   effacer une case et placer un nouveau chiffre de 0 à 7.
5. Classer et numéroter les 2n grilles obtenues selon leur score, du meilleur au
   moins bon.
6. Dupliquer la grille 1 (la meilleure) et placer ce doublon en position 2n,
   après avoir éliminé la grille 2n (la moins bonne).
7. Le cas échéant, mettre à jour le meilleur score et la meilleure position.
8. Retourner à 3, tant qu'on n'a pas décidé de s'arrêter.
```

Schéma de l'algorithme génétique



Quelques commentaires sur ce schéma

- Chaque position est représentée pas une liste de *n* nombres indiquant la ligne occupée pour la colonne correspondante.
  Avec 8 dames, la position ci-dessous est représentée par la liste [0, 5, 1, 4, 6, 3, 7, 2].



- Chaque position est évaluée grâce à la fonction de performance, ici ce sera le nombre de conflits. (b)
- Une phase de reproduction détermine quelles positions seront sélectionnées pour la reproduction. Certaines positions peuvent être reproduites plusieurs fois, d'autres disparaîtront. (c)
- Pour chaque paire se combinant, on détermine aléatoirement le point de croisement. (c)
- Les enfants sont créés en croisant chaque paire. (d)
- Finalement, chaque position subit éventuellement une mutation aléatoire. (e)

Commentaire

- Cette approche n'a (pour l'instant) pas donné de résultats intéressants. Elle ne fonctionne qu'avec de petits damiers. Il semble difficile de choisir les différents paramètres.

A9.4. Trouver tous les mots d'une grille de Ruzzle

Ruzzle est un jeu de lettres aux règles simples : une grille de quatre fois quatre lettres, et deux minutes pour trouver le plus de mots possibles, avec pour seul impératif que les cases se touchent. On ne peut pas utiliser deux fois la même case, ce qui fait que la longueur maximale d'un mot sera de 16.

Pour pimenter le jeu, *Ruzzle* reprend également les cases bonus « lettre compte double », « lettre compte triple », « mot compte double » et « mot compte triple » du Scrabble. Pour gagner, il faudra obtenir le plus gros total de points !

Dans la version que nous allons voir, on ne tiendra pas compte de ces bonus, ni de la valeur des lettres ; on se contentera de trouver tous les mots possibles, du plus long au plus court.

Voici le programme complet que nos allons décortiquer :

```
from trie_fr import Trie

# --------- case de la grille de Ruzzle -------------

class Case:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

    def addNeighbor(self, neighbor):
        # ajoute un voisin à la liste
        self.neighbors.append(neighbor)

    def getNeighbors(self):
        # renvoie la liste des voisins d'une case
        return self.neighbors

    def getValue(self):
        # renvoie la lettre d'une case
        return self.value

# -------------------------------------------------

def creer_trie(dico):
    print("Lecture du dictionnaire")
    fichier = open(dico, 'r')
    liste_mots = fichier.readlines()
    fichier.close()
    print("Création du Trie")
    trie = Trie()
    trie.makeTrie(liste_mots)
    print("Trie terminé")
    return trie

def creer_connexions(grille):
    # crée les listes d'adjacences du graphe
    for m in range(16):
        case = grille[m]
        i, j = m//4, m%4
        for n in range(16):
            x, y = n//4, n%4
            if abs(i-x)<=1 and abs(j-y)<=1 and m != n:
                case.addNeighbor(grille[n])

def chercher(noeud, case, vus=[]):
    # cherche les mots français dans le Trie
    trouves = []
    if noeud == None:
        return trouves
```

```
    if noeud.isWordEnd():
        trouves.append(noeud.getLetter())
    vus.append(case)
    for voisin in case.getNeighbors():
        if voisin not in vus:
            results = chercher(noeud[voisin.getValue()], voisin, vus)
              trouves.extend([noeud.getLetter()+ending for ending in results])
    vus.remove(case)
    return trouves

# -------------- programme principal --------------------

trie = creer_trie('ruzzle_dictionnaire.txt')
while True:
    donnees = input("Entrez la grille ligne par ligne : ")
    while len(donnees)!=16 and len(donnees)!=0:
        donnees = input("Entrez la grille ligne par ligne : ")
    if donnees=="": # taper 'return' pour sortir de la boucle infinie
        break
    donnees = donnees.lower()
    grille = [Case(lettre) for lettre in donnees]
    creer_connexions(grille)
    resultats = []
    for case in grille: # première lettre du mot à chercher
        noeud = trie    # on se place au début du Trie
        mots_trouves = chercher(noeud[case.getValue()], case)
        if len(mots_trouves) > 0:
            resultats.extend(mots_trouves)
    resultats = list(set(resultats))  # élimine les doublons
     output = sorted(resultats, key=lambda mot: [len(mot)], reverse=True)
    print(len(output),"mots trouvés")
    print(output)
    print()
```

On voit sur la première ligne que le programme importe la classe `Trie` du § 8.9.1. La procédure `creer_trie` a déjà été utilisée au § 8.9.2

On a créé une classe `case`. Une case contient une lettre et la liste des lettres qui sont sur des cases voisines (pas voisin, on entend qui jouxte horizontalement, verticalement ou en diagonale).

```
class Case:
    def __init__(self, value):
        self.value = value
        self.neighbors = []

    def addNeighbor(self, neighbor):
        # ajoute un voisin à la liste
        self.neighbors.append(neighbor)

    def getNeighbors(self):
        # renvoie la liste des voisins d'une case
        return self.neighbors

    def getValue(self):
        # renvoie la lettre d'une case
        return self.value
```

La liste des lettres voisines d'une case est créée par la procédure `creer_connexions(grille)`. La grille est implémentée par une liste de 16 éléments.

Pour chacune des 16 lettres de la grille, on calcule sa ligne `i` et sa colonne `j` en fonction de la position `m` dans `grille`. On parcourt ensuite les 15 autres cases de la grille et on calcule leur ligne `x` et leur colonne `y`. Pour être voisin de cette case, il faut que `|i-x|` soit inférieur ou égal à 1 et de même pour `|j-y|`.

```
def creer_connexions(grille):
    # crée les listes d'adjacences du graphe
    for m in range(16):
        case = grille[m]
        i, j = m//4, m%4
        for n in range(16):
```

```
            x, y = n//4, n%4
            if abs(i-x)<=1 and abs(j-y)<=1 and m!=n:
                case.addNeighbor(grille[n])
```

Voici les listes que l'on obtient pour la grille

```
u a e e
s l r u
i i e a
o t s y
```

```
u : a s l
a : u e s l r
e : a e l r u
e : e r u
s : u a l i i
l : u a e s r i i e
r : a e e l u i e a
u : e e r e a
i : s l i o t
i : s l r i e o t s
e : l r u i a t s y
a : r u e s y
o : i i t
t : i i e o s
s : i e a t y
y : e a s
```

La fonction la plus délicate à analyser est évidemment `chercher`, car elle est récursive.

On a vu comment chercher un mot dans le trie (§ 8.9.2). Ici, le problème est un peu différent. En effet, il ne serait pas très efficace de chercher toutes les suites de lettres possibles[1] et vérifier ensuite lesquels sont des mots français dans le trie. Par expérience, il y a généralement entre 250 et 400 mots valides dans une grille. On va faire l'inverse : on va parcourir le trie en utilisant les voisins.

On part du sommet du trie avec la première lettre de la grille (celle en haut à gauche).

On choisit son premier voisin dans la grille et on regarde si la suite de ces deux lettres existe dans le trie.

Si la suite n'existe pas, on abandonne cette branche du trie, on marque ce voisin comme vu et on essaie un autre voisin.

Si la suite existe, on regarde si c'est un mot complet. Si oui, on le mémorise. Si non, on prend un voisin de la deuxième lettre et on analyse la suite des trois lettres. Et ainsi de suite…

```
def chercher(noeud, case, vus=[]):
    # cherche les mots français dans le Trie
    trouves = []
    if noeud == None:
        return trouves
    if noeud.isWordEnd():
        trouves.append(noeud.getLetter())
    vus.append(case)
    for voisin in case.getNeighbors():
        if voisin not in vus:
            results = chercher(noeud[voisin.getValue()], voisin, vus)
            trouves.extend([noeud.getLetter()+ending for ending in results])
    vus.remove(case)
    return trouves
```

On parcourt selon ce principe le trie 16 fois, car il y a 16 cases possibles pour la première lettre. On élimine finalement les doublons dans la liste des mots trouvés et on affiche tous les mots du plus long au plus court.

```
    for case in grille: # première lettre du mot à chercher
        noeud = trie    # on se place au début du Trie
        mots_trouves = chercher(noeud[case.getValue()], case)
        if len(mots_trouves) > 0:
            resultats.extend(mots_trouves)
    resultats = list(set(resultats))  # élimine les doublons
```

1 il y en a 12'029'624 selon [7]

```
    output = sorted(resultats, key=lambda mot: [len(mot)], reverse=True)
print(len(output),"mots trouvés")
print(output)
```

Exercice A9.6

Modifiez le programme ci-dessus pour trouver la plus belle grille de *Ruzzle* grâce à un algorithme probabiliste. Par « plus belle », on entend celle qui contient le plus de mots français.

Vous utiliserez le dictionnaire du chapitre 9 qui est disponible sur le site web compagnon : https://www.apprendre-en-ligne.net/info/algo/

Voici les fréquences des lettres de ce dictionnaire (on a analysé tous les mots de 2 à 16 lettres sans tiret et sans apostrophe) :

E	S	A	I	R	N	T	O	L	U	C	M	D
14.89%	10.21%	9.71%	9.40%	8.67%	7.34%	6.82%	5.82%	4.01%	3.60%	3.40%	2.54%	2.36%
P	G	B	F	H	Z	V	Q	Y	X	J	K	W
2.35%	1.60%	1.40%	1.36%	1.16%	1.07%	0.96%	0.50%	0.34%	0.25%	0.18%	0.05%	0.01%

Exercice A9.7

Modifiez le programme de l'exercice A9.6 pour rechercher la meilleure grille de Ruzzle grâce à une descente de plus grande pente.

Exercice A9.8

Modifiez le programme de l'exercice A9.6 pour rechercher la meilleure grille de *Ruzzle* grâce à une recherche avec tabous.

Exercice A9.9

Modifiez le programme de l'exercice A9.6 pour rechercher la meilleure grille de *Ruzzle* grâce à un recuit simulé.

Exercice A9.10

Modifiez le programme de l'exercice A9.6 pour rechercher la meilleure grille de *Ruzzle* grâce à un algorithme génétique.

A9.5. Affectation sous contraintes

Dans un lycée suisse, en deuxième année, 150 élèves doivent choisir une option complémentaire (une OC). Ils doivent indiquer deux choix en précisant leur préférence. Une OC ne peut s'ouvrir que s'il y a au moins 8 élèves.

Comment faire pour satisfaire le maximum d'élèves tout en ouvrant le plus d'OC possible ?

On dira qu'un élève est *déçu* si on lui a attribué son second choix et *mécontent* si aucun de ses deux choix n'a été retenu, car les deux OC qu'il a choisies n'ont pas pu être ouvertes, par manque

d'effectif.

De plus, on constate que statistiquement 4 des 13 OC ont nettement moins de succès que les autres (disons qu'en moyenne 3 fois moins d'élèves choisissent ces OC).

On va simuler deux méthodes de résolution.

Première méthode

On attribue à tous les élèves leur premier choix. Les OC ayant moins de 8 élèves sont fermées et on attribue aux élèves concernés leur deuxième choix. Les élèves mécontents seront convoqués par la direction pour faire un troisième choix parmi les OC ouvertes.

Seconde méthode

Dans un premier temps, on répartit les élèves aux OC au hasard, selon un des leurs deux choix, sans tenir compte de leur préférence.

Dans un deuxième temps, on utilise le recuit simulé. On obtiendra une solution voisine en mettant un élève aléatoire dans son autre choix d'OC. Le score devra tenir compte de l'ouverture ou non de l'OC et des choix des élèves.

Recuit simulé

1. Choisir une « température » de départ T.
2. Générer une répartition aléatoire. Appelons-la $R1$.
3. Copier la répartition $R1$ dans une répartition $R2$, puis déplacer un élève aléatoire dans son autre choix d'OC.
4. Calculer Δ = score($R2$) − score($R1$)
5. Si $\Delta > 0$, $R1 \leftarrow R2$; le cas échéant, mettre à jour le meilleur score et la meilleure répartition. Aller à 8.
6. Générer un nombre réel aléatoire entre 0 et 1, que nous appellerons r.
7. Si $r < \exp(\Delta/T)$, $R1 \leftarrow R2$.
8. Diminuer T.
9. Retourner à 3, tant qu'on n'a pas décidé de s'arrêter.

Exercice A9.11

a. Programmez la première méthode. Faites une dizaine de simulations et notez le nombre d'élèves déçus, d'élèves mécontents et d'OC fermées.

b. Idem pour la seconde méthode.

c. Comparez les résultats obtenus pour les deux méthodes.

Sources

[1] Wikipédia, « Méta-heuristique », <http://fr.wikipedia.org/wiki/Métaheuristique>

[2] Wikipédia, « Recuit simulé », <http://fr.wikipedia.org/wiki/Recuit_simulé>

[3] Wikipédia, « Algorithme génétique », <http://fr.wikipedia.org/wiki/Algorithme_génétique>