Séance 6: Compression et labyrinthes

Université Paris Cité

1 Lempel-Ziv

L'algorithme de Lempel-Ziv (dû à Ziv et Lempel en 1977) est un algorithme de compression universel qui a été largement utilisé, par exemple dans gzip. Ici nous allons étudier une variante appelée LZ'78.

Afin de compresser un texte, l'algorithme se base sur les répétitions de (morceaux de) mots. Il construit un « dictionnaire » de suites de lettres déjà vues, ce qui permet de donner simplement l'indice dans le dictionnaire lorsqu'on voit à nouveau la même suite.

Plus précisément, supposons que l'on veuille compresser le texte « Lou, la lacune lacustre », duquel on ôtera les capitales, les accents, la ponctuation et les espaces pour obtenir la chaîne "loulalacunelacustre". On parcourt la chaîne en complétant le dictionnaire au fur et à mesure. Le dictionnaire sera un tableau de chaînes de caractères. La première lettre lue est "l", c'est donc la première chaîne du dictionnaire. Ensuite, à partir de la position actuelle dans le texte, on ajoute dans le dictionnaire la plus courte suite de lettres qui ne s'y trouve pas encore. Ici, après "l" on lit "o", qui n'est pas dans le dictionnaire, donc on l'y ajoute. Ensuite, on lit "u" qu'on ajoute également dans le dictionnaire. Le comportement intéressant vient ensuite : on a déjà "l" dans le dictionnaire, mais pas encore "la", donc on ajoute "la". Le dictionnaire contient maintenant `"l"`, `"o"`, `"u"`, `"la"`. On continue ainsi : la chaîne suivante est "lac", puis "un", puis "e", "lacu", etc. On obtient donc le dictionnaire

```
{"l", "o", "u", "la", "lac", "un", "e", "lacu", "s", "t", "r", "e"}
```

On ne va évidemment pas donner le dictionnaire tel quel, sinon on ne compresserait pas du tout. On remarque que pour obtenir la chaîne "la", par exemple, on a ajouté une lettre à "l" : on va coder cela avec l'indice de "l" dans le tableau (0) et avec la lettre ajoutée ('a'). De même, pour "lacu", c'est la chaîne "lac" (position 4) à laquelle on a ajouté la lettre 'u'. Chaque chaîne du dictionnaire sera ainsi codée par l'indice de la chaîne qu'elle complète et par la lettre qu'on ajoute. Pour les chaînes de taille 1 comme "l", "o", etc., on prendre la convention qu'elles viennent compléter une chaîne virtuelle à la position -1. Dans l'exemple, cela donnerait :

$$(-1, l); (-1, o); (-1, u); (0, a); (3, c); (2, n); (-1, e); (4, u); (-1, s); (-1, t); (-1, r); (-1, e).$$

Nous allons stocker les indices successifs `{-1, -1, -1, 0, 3, 2, -1, 4, -1, -1, -1, -1}` dans un tableau d'entiers, et les lettres correspondantes `{'l', 'o', 'u', 'a', 'c', 'n', 'e', 'u', 's', 't', 'r', 'e'}` dans un tableau de caractères. Ces deux tableaux constituent la compression du texte.

Exercice 1 (Compression  , ⋆⋆⋆)

Écrire une fonction `LZencode` *qui prend en arguments une chaîne de caractères s, un tableau d'entiers t et un tableau de caractères u (supposés suffisamment grands), qui remplit les tableaux t et u jusqu'à la case d'indice $i-1$ grâce à l'algorithme de Lempel-Ziv ci-dessus et renvoie i.*

Contrat:
Par exemple, l'appel `LZencode("loulalacunelacustre", t, u)` *renvoie* 12*, et après cet appel, les* 12 *premières cases des tableaux* t *et* u *doivent contenir respectivement :*

```
{-1, -1, -1, 0, 3, 2, -1, 4, -1, -1, -1, -1, ...}
{'l', 'o', 'u', 'a', 'c', 'n', 'e', 'u', 's', 't', 'r', 'e',
    ...}
```

Évaluer la taille de la compression (= la taille des tableaux t *et* u*) par rapport à la taille du texte de départ, pour les textes donnés sur moodle.*

□

Exercice 2 (Décompression 🔍, ⋆⋆)
Programmer `LZdecode`*, la fonction qui attend un tableau d'entiers et un tableau de caractères, et renvoie la chaîne de caractères qu'ils représentent.*

Contrat:
Par exemple, l'appel à `LZdecode({-1, -1, -1, 0, 3, 2, -1, 4, -1, -1, -1, -1}, {'l', 'o', 'u', 'a', 'c', 'n', 'e', 'u', 's', 't', 'r', 'e'})` *doit renvoyer* `"loulalacunelacustre"`*.*

□

2 Labyrinthes

Dans cette partie, nous allons travailler avec des tableaux de tableaux d'entiers qui représentent un labyrinthe. À l'aide des boucle `while` et `for` nous allons implémenter un simple algorithme permettant de calculer le nombre minimal de cases à parcourir pour aller du coin haut gauche au coin bas droit dans le labyrinthe.

Rédiger les réponses aux questions et les tests pour chaque réponse dans le fichier *Labyrinthe.java* qui vous est fourni.

Dans le fichier *labyrinthe1.csv* vous est donnée la description d'un labyrinthe. On vous donne également dans *Labyrinthe.java* une fonction `chargeLabyrinthe` qui prend en argument un nom de fichier contenant la description d'un labyrinthe et renvoie le tableau de tableaux d'entiers correspondant au labyrinthe. Ainsi l'instruction `chargeLabyrinthe('labyrinthe1.csv')` renverra un tableau de tableaux d'entiers.

Dans ce tableau, chaque sous-tableau représente une ligne du labyrinthe, le premier tableau représente la première ligne, le deuxième tableau la deuxième ligne, etc. Pour le moment ce tableau ne contient que des 0 et des 1, mais nous verrons que nous y stockerons d'autres valeurs entières positives. L'idée est qu'un 0 représente un mur dans le labyrinthe et une valeur strictement positive une case libre.

Le but est de savoir s'il existe un chemin dans le labyrinthe depuis la position en haut à gauche jusqu'à la position en bas à droite. Pour se déplacer dans le labyrinthe, on peut seulement aller vers le haut, vers le bas, à gauche et à droite sans passer par des cases où se trouvent des murs.

Pour ce faire l'algorithme commence par écrire 2 dans la case en haut à gauche, ensuite pour chaque case voisine écrite avec un 1, il met un 3, ensuite pour chaque case écrite contenant un 3, il met un 4 dans chaque voisine contenant un 1, etc L'algorithme s'arrête soit s'il a changé l'entier dans la case en bas à droite soit s'il n'a plus de cases à changer. Voilà un exemple des différentes étapes.

1	1	1	1
0	1	0	1
0	1	1	1

2	1	1	1
0	1	0	1
0	1	1	1

2	3	1	1
0	1	0	1
0	1	1	1

2	3	4	1
0	4	0	1
0	1	1	1

2	3	4	5
0	4	0	1
0	5	1	1

2	3	4	5
0	4	0	6
0	5	6	1

2	3	4	5
0	4	0	6
0	5	6	7

Dans cet exemple, à la fin de l'algorithme la case en bas à droite contient un 7 donc on peut en déduire qu'il faut traverser 6 cases pour traverser le labyrinthe depuis la case en haut à gauche.

Exercice 3 (À vous de jouer !, ⋆-⋆⋆)

Dans ce qui suit, le labyrinthe désignera un tableau de tableaux d'entiers. **Testez vos fonctions régulièrement avec le labyrinthe qui vous est fourni.**

1. *Écrire une fonction* `afficheLab` *qui prend en argument un labyrinthe et qui l'affiche dans le terminal ligne par ligne en mettant un caractère* `'X'` *s'il y a un mur et un caractère* `' '` *(espace) s'il n'y a pas de mur.*
2. *Nous souhaitons travailler sur une copie du labyrinthe. Écrire une fonction* `copieLab` *qui prend en argument un labyrinthe et renvoie une copie de ce labyrinthe.*
3. *Écrire une fonction* `caseHaut` *qui prend en arguments un labyrinthe et un numéro de ligne et un numéro de colonnes correspondant à une case (on suppose que ces numéros commençant à 0 sont dans les limites du labyrinthe) et qui renvoie l'entier contenu dans la case au-dessus si celle-ci existe et -1 si elle n'existe pas (dans le cas par exemple où la case donnée en arguments se trouve sur la première ligne).*
4. *De la même façon, écrire des fonctions* `caseBas`, `caseGauche` *et* `caseDroite` *qui prennent en arguments un labyrinthe et un numéro de ligne et un numéro de colonnes correspondant à une case et qui renvoie l'entier contenu dans la case au-dessous, respectivement à gauche respectivement à droite si celles-ci existent et si elle n'existent pas, ces fonctions renvoient -1.*
5. *En utilisant les fonctions des questions précédentes, écrire une fonction* `voisinsLibres` *qui prend en arguments un labyrinthe et un numéro de ligne et un numéro de colonnes correspondant à une case et qui renvoie un tableau de tableaux de deux entiers où chaque sous-tableau contient deux entiers (numéro de ligne et numéro de colonne) correspondant à une case voisine de la case donnée en arguments et dans laquelle se trouve un 1.*

   Exemple : *Sur le premier labyrinthe dessiné ci-dessus, pour la case (0,0), la fonction* `voisinsLibres` *renverra le tableau* `{{0,1}}` *et pour la case (1,1), la fonction* `voisinsLibres` *renverra le tableau* `{{0,1}, {2,1}}`.
6. *Écrire maintenant une fonction* `changeVoisins` *qui prend en arguments un labyrinthe, deux entiers correspondant à une case (numéro de ligne et numéro de colonne) et un entier* `i` *et qui change le contenu de toutes les cases voisines de la case donnée et qui contiennent un 1 en y mettant à la place la valeur* `i+1`. *Cette fonction ne renvoie rien mais elle change le labyrinthe.*
7. *Écrire maintenant une fonction* `etapeParcours` *qui prend en arguments un labyrinthe et un entier* `i` *et parcourt toutes les cases du labyrinthe et pour chaque case qui contient l'entier* `i`, *change ses voisins dans lesquels figurent un 1 en le remplaçant par* `i+1`. *Cette fonction ne renvoie rien mais elle change le labyrinthe.*
8. *Écrire une fonction* `finParcours` *qui servira à tester si le parcours de notre labyrinthe peut s'arrêter. Cette fonction prend en arguments un labyrinthe et renvoie* `True` *si dans la case en bas à droite il y a une valeur différente de 1 ou si toutes les cases contenant une valeur différente de 1 n'ont pas de voisins avec une valeur différente de 1. Dans les autres cas, cette fonction renverra* `False`.
9. *Nous avons maintenant tous les ingrédients pour parcourir notre labyrinthe selon l'algorithme décrit au début de ce TP.*

   Écrire une fonction `parcours` *qui prend en argument un labyrinthe. Cette fonction commence par écrire un 2 dans la case en haut à gauche du labyrinthe. Ensuite tant que le labyrinthe ne vérifie pas les conditions de fin de parcours données dans la question précédente, elle appelle la fonction* `etapeParcours` *en lui donnant en arguments le labyrinthe et les entiers 2,3,4, etc.*

   Quand cette fonction s'arrête, si le chiffre dans la case en bas à droite est strictement plus grand que 1 alors il correspond au nombre de cases minimal moins un à visiter pour traverser le labyrinthe.

Pensez à tester votre fonction. Combien de pas faut-il pour traverser le labyrinthe contenu dans labyrinthe1.csv *?*

10. **Bonus :** *On peut aussi se servir de la fonction* `parcours` *pour générer des labyrinthes "corrects" de façon aléatoire. Pour cela, vous générez un tableau de tableaux d'entiers remplis de 0 et vous choisissez un entier* `a`*. Ce tableau de tableaux représentera notre labyrinthe. Vous mettez un 1 dans la case en haut à gauche et ensuite vous tirez* `a` *cases au hasard (rappelons qu'une case est donnée par un numéro de lignes et un numéro de colonnes) et si dans ces cases, il y a un 0 vous le remplacez par un 1. Vous testez ensuite si votre labyrinthe peut être traversable.*

    Écrire une fonction `genereLab` *qui prend en arguments trois entiers* `n`, `m` *et* `a` *et qui génère un labyrinthe à* `n` *lignes et* `m` *colonnes en tirant à chaque tour avant de tester si le labyrinthe est traversable en* `a` *cases. Cette fonction renvoie un labyrinthe.*

    Vous pouvez tester votre fonction avec la fonction `afficheLab`*.*

□