IFT1015 – Travail pratique #1 – 2017-10-10

DESSINER DES LABYRINTHES

Marc Feeley

Le TP1 a pour but de vous faire pratiquer les concepts suivants : les boucles, les tableaux, les fonctions, la décomposition fonctionnelle, les graphiques "tortue" et les tests unitaires.

Vous avez à coder et tester un ensemble de fonctions en JavaScript, à l'aide de codeBoot, en vous limitant aux aspects du langage que nous avons vus avant l'examen intra.

---

1 Introduction

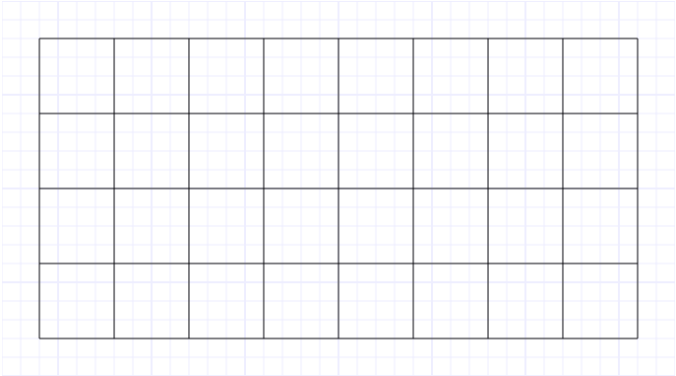
Ce travail pratique consiste à développer un ensemble de définitions de fonctions qui permettent de générer des labyrinthes et les dessiner sur la fenêtre de dessin de codeBoot.

Dans ce travail, il faut s'imaginer que vous êtes le programmeur d'une bibliothèque qui permet de générer des labyrinthes. Vous devez écrire des fonctions qui seront utilisées par d'autres programmeurs pour faire des dessins spécifiques (créer un labyrinthe d'une certaine taille).

L'algorithme pour générer des labyrinthes vous est imposé et est décrit dans la prochaine section. Votre tâche est de coder cet algorithme, et les fonctions auxiliaires nécessaires, en JavaScript à l'aide de codeBoot.

2 Labyrinthes

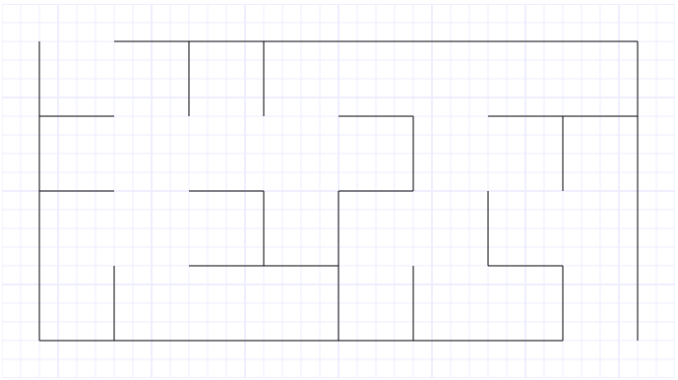
Les labyrinthes que vous devez générer sont basés sur une grille rectangulaire comme celle-ci :



Cette grille est formée de cellules carrées. Dans cet exemple il y a 8 colonnes et 4 rangées de cellules. Pour une cellule donnée il y a 4 murs qu'on appellera les murs Nord, Est, Sud, et Ouest (pour simplifier "N", "E", "S", "O") de la cellule. Un labyrinthe a une certaine largeur en nombre de cellules (NX), une certaine hauteur en nombre de cellules (NY), et un certain "pas" (la largeur et hauteur de chaque cellule carrée, en nombre de pixels).

Le labyrinthe est donc créé en éliminant certains murs de la grille. Le mur externe de la grille est percé en haut à gauche (l'entrée du labyrinthe) et en bas à droite (la sortie du labyrinthe). La création d'un

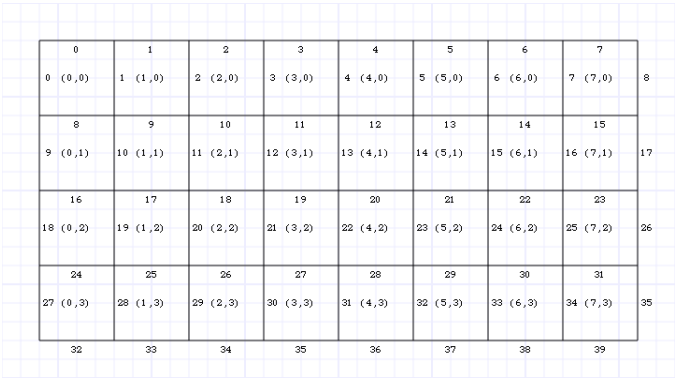
labyrinthe consiste à déterminer quels murs de chaque cellule doivent être éliminés. Pour chaque cellule il faut éliminer au moins un de ses 4 murs, et possiblement tous les 4. Voici un labyrinthe utilisant la grille précédente (i.e. avec NX=8, NY=4, pas=40) :



Le choix des murs à éliminer ne peut pas se faire complètement aléatoirement car cela pourrait générer un dessin qui n'est pas un labyrinthe, c'est-à-dire un dessin qui n'offre pas un chemin se rendant de l'entrée à la sortie.

L'algorithme que vous devez utiliser se sert de nombres aléatoires pour créer le labyrinthe mais garantit qu'il y a toujours exactement un chemin entre l'entrée et la sortie du labyrinthe. En fait, l'algorithme garantit qu'il y a toujours exactement un chemin entre une cellule quelconque et toute autre cellule (ce qu'on appelle formellement un *arbre sous-tendant*).

Avant d'expliquer l'algorithme il faut tout d'abord numéroter chaque mur horizontal et chaque mur vertical et donner une coordonnée (x,y) à chaque cellule comme suit :



Il est à noter que le système de coordonnée (x,y) des cellules place (0,0) en haut à gauche (x grandit vers la droite, et y grandit vers le bas). Remarquez aussi qu'il y a $NX \times (NY + 1)$ murs horizontaux et $(NX + 1) \times NY$ murs verticaux. Donc les murs horizontaux sont numérotés de 0 à $(NX \times (NY + 1)) - 1$ et les murs verticaux de 0 à $((NX + 1) \times NY) - 1$. La relation entre la coordonnée (x,y) d'une cellule et les numéros de ses murs N, E, S, O est la suivante :

$$\begin{aligned}
\text{N} &= x + y \times NX \\
\text{E} &= 1 + x + y \times (NX + 1) \\
\text{S} &= x + (y + 1) \times NX \\
\text{O} &= x + y \times (NX + 1)
\end{aligned}$$

Remarquez finalement qu'on peut se servir du numéro du mur N pour identifier une cellule de façon unique avec un seul numéro. Par exemple dans la grille ci-dessus la cellule en position (5,2) a le numéro 21, c'est-à-dire $5 + 2 \times NX$.

L'algorithme se sert d'ensembles de numéros (dans le sens mathématique d'ensemble, c'est-à-dire une collection de nombres sans duplication). On peut définir **mursH** comme l'ensemble de murs horizontaux (et respectivement **mursV** comme l'ensemble de murs verticaux) qui n'ont pas été retirés par l'algorithme de création de labyrinthe. L'information contenue dans ces ensembles peut être combinée avec les valeurs de NX et NY pour dessiner un labyrinthe en traçant un trait horizontal pour chaque mur horizontal dont le numéro est toujours dans **mursH** et un trait vertical pour chaque mur vertical dont le numéro est toujours dans **mursV**.

L'algorithme considère que la grille est un espace initialement plein sauf pour une cellule choisie aléatoirement qui est vide (c'est la *cavité* initiale). À chaque itération de l'algorithme une nouvelle cellule sera choisie aléatoirement parmi toutes les cellules voisines de la cavité (mais ne faisant pas partie de la cavité) et un des murs (soit horizontal ou vertical) qui la sépare de la cavité sera retiré aléatoirement pour former une plus grande cavité (soit de l'ensemble **mursH** si c'est un mur horizontal ou de l'ensemble **mursV** si c'est un mur vertical). Ce processus est répété jusqu'à ce que toutes les cellules de la grille fassent partie de la cavité.

Le choix de la prochaine cellule à ajouter à la cavité peut se faire simplement en conservant en tout temps deux autres ensembles de numéros : **cave** et **front**. Ce sont des ensemble qui contiennent des numéros de cellules. L'ensemble **cave** est l'ensemble des cellules qui ont été mis dans la cavité par l'algorithme. L'ensemble **front** est l'ensemble des cellules qui sont voisines des cellules dans la cavité (mais pas dans la cavité). On peut maintenir à jour ces ensembles au fur et à mesure qu'on sélectionne des nouvelles cellules à ajouter à la cavité. En effet si on a ajouté à la cavité la cellule aux coordonnées (x,y) contenue dans l'ensemble **front**, il faut ajouter les cellules voisines à (x,y) horizontalement et verticalement (mais pas dans l'ensemble **cave**) à l'ensemble **front** et retirer la cellule (x,y) de l'ensemble **front** et ajouter la cellule (x,y) à l'ensemble **cave**. Remarquez qu'on doit retirer ou ajouter de ces ensembles le *numéro* des cellules.

Le codage de cet algorithme sera grandement simplifié par l'écriture de fonctions de manipulation d'ensembles. Ces fonctions sont décrites dans la prochaine section. À vous de les utiliser judicieusement pour coder l'algorithme de création de labyrinthe. Nous utiliserons des tableaux de nombres pour représenter les ensembles. Par exemple le tableau `[9,2,5]` représente l'ensemble qui contient les trois éléments 2, 5 et 9.

3 Spécification

Vous devez concevoir et coder les fonctions spécifiées ci-dessous en respectant le nom spécifié **exactement** pour qu'on puisse les tester plus facilement. Si vous devez définir des fonctions auxiliaires utilisez des noms appropriés de votre choix. Chaque fonction, sauf la fonction `laby`, doit avoir une fonction de tests unitaires qui est exécutée par votre code. Votre code doit être dans un fichier nommé "`labyrinthe.js`".

3.1 Fonction `iota(`n`)`

Cette fonction prend un entier non-négatif n en paramètre et retourne un tableau de longueur n contenant en ordre les valeurs entières de 0 à n-1 inclusivement. Par exemple :

`iota(5) = [0,1,2,3,4]`

3.2 Fonction `contient(`*tab*`, `*x*`)`

Cette fonction prend en paramètre un tableau de nombres (*tab*) et un nombre x et retourne un booléen indiquant si x est contenu dans le tableau. Par exemple :

```
contient([9,2,5], 2) = true
contient([9,2,5], 4) = false
```

3.3 Fonction `ajouter(`*tab*`, `*x*`)`

Cette fonction prend en paramètre un tableau de nombres (*tab*) et un nombre x et retourne un nouveau tableau avec le même contenu que *tab* sauf que x est ajouté à la fin s'il n'est pas déjà contenu dans *tab*. Par exemple :

```
ajouter([9,2,5], 2) = [9,2,5]
ajouter([9,2,5], 4) = [9,2,5,4]
```

3.4 Fonction `retirer(`*tab*`, `*x*`)`

Cette fonction prend en paramètre un tableau de nombres (*tab*) et un nombre x et retourne un nouveau tableau avec le même contenu que *tab* sauf que x est retiré du tableau. Par exemple :

```
retirer([9,2,5], 2) = [9,5]
retirer([9,2,5], 4) = [9,2,5]
```

3.5 Fonction `voisins(`*x*`, `*y*`, `*nx*`, `*ny*`)`

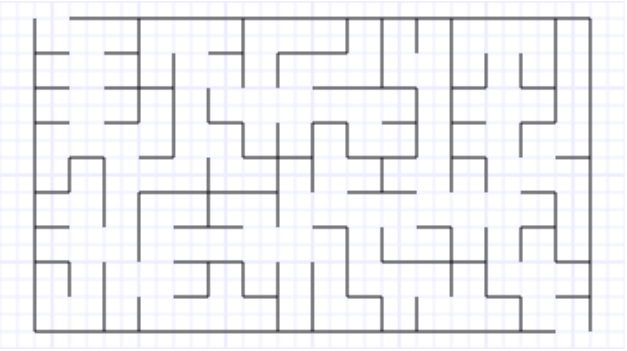
Cette fonction prend la coordonnée (x,y) d'une cellule et la taille d'une grille (largeur=nx et hauteur=ny) et retourne un tableau contenant le numéro des cellules voisines. Par exemple :

```
voisins(7, 2, 8, 4) = [15,22,31]
```

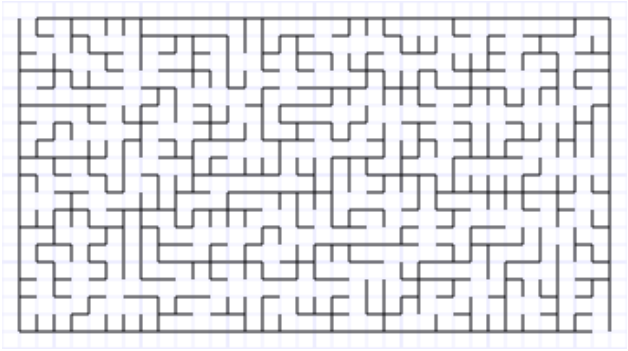
3.6 Procédure `laby(`*nx*`, `*ny*`, `*pas*`)`

Cette procédure crée un labyrinthe aléatoire (largeur=nx et hauteur=ny) et dessine ce labyrinthe au centre de la fenêtre de dessin en utilisant une grille avec des cellules de *pas* pixels de largeur et hauteur. Par exemple, voici deux appels à `laby` et les dessins obtenus dans chaque cas :

`laby(16, 9, 20);`



`laby(34, 18, 10);`



4 Évaluation

- Ce travail compte pour 15 points dans la note finale du cours. Vous devez faire le travail par groupes de 2 personnes. Indiquez vos noms clairement dans les commentaires au début de votre code.
- Vous devez remettre votre fichier "`labyrinthe.js`" uniquement. La remise doit se faire au plus tard à 23h55 dimanche le 29 octobre sur le site Studium du cours. Il y a une pénalité de 33% pour chaque jour de retard.
- Voici les critères d'évaluation du travail : l'exactitude (respect de la spécification), l'élégance et la lisibilité du code, la présence de commentaires explicatifs, le choix des identificateurs, la décomposition fonctionnelle, le choix de tests unitaires pertinents et l'utilisation d'un français sans fautes. La performance de votre code (nombre de pas pour faire le traitement demandé) doit être raisonnable.

  Voici le barême précis, sur 100

-50% le respect de la spécification (perte de jusqu'à 50% si la spécification n'est pas respectée, par exemple demander des informations avec `prompt`, génération d'un labyrinthe incorrect (tel que plus d'un chemin entre l'entrée et la sortie), affichage incorrect du labyrinthe, etc)

30% l'élégance du code (logique du programme, pas de code redondant, utilisation de paramètres, la décomposition des fonctions en sous-fonctions (pas de fonctions trop longues), utilisation appropriée des opérateurs JavaScript, éviter les variables globales, etc)

30% la lisibilité et la présence de commentaires explicatifs (chaque fonction devrait avoir un bref commentaire pour dire ce qu'elle fait, il devrait y avoir des lignes blanches pour que le code ne soit pas trop dense, les identificateurs doivent être bien choisis pour être compréhensibles et respecter le standard CamelCase)

30% le choix de tests unitaires pertinents (incluant des tests avec des tableaux vides, et les cas spéciaux)

10% performance du code (nombre de pas pour `laby(10,9,20)` doit être < 2 million... le programme du professeur prend un peu moins que 500,000 pas)

20% points bonus pour l'implantation de la procédure `labySol(`*nx*`,` *ny*`,` *pas*`)` qui dessine un labyrinthe exactement comme la procédure `laby` mais qui en plus dessine dans le labyrinthe, en rouge, le trajet parcouru par l'algorithme de Pledge pour sortir du labyrinthe à partir de l'entrée en haut à gauche. Voir le lien suivant pour la description de cet algorithme :
`https://interstices.info/jcms/c_46065/l-algorithme-de-pledge`