université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Projet IPF
LDD2 IM

Point d'étape

Objectif

Le but de cette feuille est de :

1. de détailler les attendus du rapport
2. de préciser les dernières fonctionnalités attendues
3. donner l'intuition des algorithmes de recherche

1 Rapport

Le contenu du rapport doit être le suivant :

— répartition du travail au sein du binôme (si vous êtes en binôme). On s'attend à ce que chacun des membres ait participé à l'écriture du code, aux tests et à l'écriture du rapport. Les découpages du style « unetelle a écrit tout le code et untel a écrit tout le rapport » seront sanctionnés du point de vue de la note (1 demi-page max). Attention, les statistiques de git pourront être utilisée pour vérifier cette répartition (i.e. si tout le monde a contribué au code, on s'attend à ce que tout le monde ait fait des commits).

— présentation générale de la structure de données de grille (1 page) (types choisis, fonctions implémentées, ...).

— Présentation d'**une** difficulté rencontrée dans l'implémentation (2 pages). Vous êtes libres de ce choix (recherche de chemin, génération du labyrinthe, problème lors de la conception, ...). Vous devez expliqué ce qui vous a posé problème et comment vous avez abordé et solutionné ce dernier.

— description des tests faits (1 page). Décrire les tests que vous avez faits. Ces derniers doivent être reproductibles, donc les fichiers de tests doivent être dans votre dépôt git.

— description du type d'affichage amélioré choisi (cf. section suivante, 1 à 2 pages avec illustrations).

On s'attend que ce rapport soit déposé dans le dépôt git dans un sous-répertoire `doc` avec le nom `rapport.pdf` ou `rapport.md`. Le format attendu est soit du PDF (par exemple, écrire votre rapport avec un traitement de texte et sauvegarder au format PDF), soit au format Markdown [1]. Vous pouvez constater que le site Gitlab de l'université sait afficher joliement les fichiers markdown.

2 Fonctionalité d'affichage

On précise les fonctionalités d'affichage demandées et on introduit une nouvelle option qui n'était pas dans le sujet initial.

2.1 Affichage basique

Pour les options `print` et `random`, l'affichage du labyrinthe doit être **exactement** au format demandé, sur la sortie standard (utilisation de `Printf.printf` par exemple). Ainsi, les commandes suivantes doivent fonctionner, sans problème.

1. `https://en.wikipedia.org/wiki/Markdown`

```
# on suppose l'existance d'un fichier test.laby
$ ./maze.exe print test.laby > test2.laby
$ diff test.laby test2.laby
# pas d'affichage attendu, i.e. fichiers identiques.

$ ./maze.exe random 10 10 42 > random.laby
$ ./maze.exe print random.laby > random2.laby
$ diff random.laby random2.laby
# pas d'affichage attendu, i.e. fichiers identiques.
```

Si vous souhaitez afficher des information de débuggage, vous devez utiliser la sortie d'erreur, par exemple avec `Printf.eprintf`. On rappelle qu'on peut rediriger indépendemment la sortie standard et la sortie d'erreur avec `>` et `2>` respectivement.

Pour l'affichage du chemin trouvé (option `solve`). Le chemin soit être affiché par un caractère de votre choix autre que `-`, `+`, `|`, `S`, `E` et évidemment l'espace. Par exemple si on choisit « `.` », alors un labyrinthe résolu peut être affiché comme :

```
$ ./maze.exe solve test.laby
+-+-+-+-+-+-+
|S  |       |
+.+ +-+ +-+ +
|.|     |   |
+.+-+-+-+-+-+
|...|       |
+-+.+ +-+-+ +
|...|   |   |
+.+-+ +-+ + +
|.    |   | |
+.+-+-+ +-+ +
|..E  |   | |
+-+-+-+-+-+-+
```

On pourra effectuer le test suivant pour s'assurer que l'affichage est correct

```
$ ./maze.exe solve test.laby | tr '.' ' ' > test2.laby
$ diff test.laby test2.laby
```

La commande `tr` du shell permet de convertir un ensemble de caractères en un autre ensembles de caractères. Ici on convertit juste le point en espace. Attention, cette fonction ne vérifie absolument pas que votre chemin est correct, elle permet juste de contrôler que vous n'avez pas introduit d'erreur d'affichage en introduisant le chemin.

2.2 Affichage amélioré

Les deux commandes `print` et `solve` doivent pouvoir prendre une option supplémentaire (dont le nom est laissé libre mais qui commence par `--` et intervient directement après la commande). Par exemple

```
$ ./maze.exe solve --pretty test.laby
$ ./maze.exe print --print test.laby
$ ./maze.exe solve --html test.laby
```

Vous pouvez choisir d'implémenter **l'une** des deux fonctionalités suivantes et la décrire dans le rapport. Quel que soit votre choix, on s'attend à une réalisation modulaire, sans copier-collers massifs du code existant.

ASCII Art, Unicode et Ansi Vous affichez dans la console le labyrinthe en utilisant

— des séquences Unicode graphique

`https://en.wikipedia.org/wiki/Box-drawing_character`

— des séquences d'échappement ANSI du terminal

`https://en.wikipedia.org/wiki/ANSI_escape_code`

On peut, en OCaml, créer une chaîne de caractère contenant de l'UTF-8. Par exemple, le caractère « ├ » peut être simplement copié-collé depuis la page Wikipedia dans une chaîne de caractère OCaml

```
let angle = "├"
let () = Prinf.printf "%s" angle
```

Il faut juste faire attention que la fonction `String.length` d'OCaml renvoie le nombre d'octets et donc que la chaîne `angle` fait 3 octets, car l'encodage en UTF-8 de ce symbole prend 3 octets, qui sont `0xE2 0x94 0x9C`. On obtient d'ailleurs le même résultat en affichant :

```
let () = Prinf.printf "\xe2\x94\x9c"
```

La notation `\xNN` permettant de donner le code de l'octet en hexadécimal. Votre affichage peut donc utiliser des caractères de dessins d'angles pour les murs, et des emojis pour la case de départ et d'arrivée. Pour ce qui est de `solve`, la fonction d'affichage du chemin pourra utiliser les séquences d'échappement ANSI du terminal pour afficher une case en couleur plutôt qu'un «.». La séquence de caractères `ESC[` mentionnée Wikipedia peut être obtenue en OCaml par

```
let () = Prinf.printf "\x1b["
```

HTML Vous affichez dans la console du code HTML représentant le labyrinthe. Le code doit être un fichier HTML complet avec des éléments de style CSS. Il doit être tel que si on fait :

```
$ ./maze.exe print --html test.laby > test.html
$ firefox test.html
```

Alors le labyrinthe s'affiche joliement. De même pour `solve`, le chemin s'affiche en couleur. Il est possible en OCaml d'écrire des chaînes de caractères complexes avec la syntaxe `{| ... |}`. Cette dernière évite à avoir à échapper les guillemets, les antislashs et autres caractères spéciaux.

```
let header = {|
  <!DOCTYPE html5>
   <html>
    <title>Mon labyrinthe</title>
|}
```

3 Résolution du labyrinthe

. On peut resoudre facilement un labyrinthe, c'est à dire trouver la sortie (si c'est possible) en appliquant un algorithme récursif (qui rentre dans la catégorie plus générales des algorithmes avec retour sur trace ou algorithmes avec *backtracking*). L'algorithme procède comme suit :

— Initialiser la position courante à `S`

— Partie récursive :

  — si la case courante est E on a trouvé la sortie

  — si la case courante a déjà été visitée, abandon

  — sinon, marquer la case courante comme visitée

  — pour chaque case c accessible depuis la case courante :

    — ajouter c au chemin courant

    — chercher un chemin récursivement à partir de c

    — si trouvé, on a fini

    — si échec, passer à la case accessible suivante

— si on a trouvé un chemin pour aucune case échec

Il vous faut donc implémenter les fonctionalités basiques de cette algorithme (cases voisines, test de E et S,...) réfléchir à la façon de se souvenir des cases visitées ....

4 Génération d'un labyrinthe

On peut procéder de la même façon qu'avec l'algorithme de recherche de chemin, en le faisant varier un peu.

- Initialiser la grille de façon à ce que toutes les cases aient 4 murs (grille pleine). Choisir une case de départ (n'importe laquelle).
- Partie récursive :
  - si la case courante a déjà été visitée ne rien faire
  - sinon, marquer la case courante comme visitée
  - pour chacune des 4 case c (haut, bas, gauche, droite, en ignorant les murs), prises dans un ordre aléatoire
    - retirer le mur entre la case courante et c
    - continuer à partir de c
    - passer à la case accessible suivante

On peut ensuite choisir aléatoirement `S` et `E`, le labyrinthe ainsi créé ayant la propriété que toute case est accessible.