Structure de données et algorithmes

Projet 2: Structures de données

Pierre Geurts – Jean-Michel Begon – Romain Mormont

22 mars 2019

L'objectif pédagogique de ce projet est de vous familiariser avec la conception, l'implémentation, et l'analyse de structures de données. L'objectif concret sera de mettre au point un programme permettant de générer un labyrinthe. La solution à ce problème mettra en œuvre une structure de données particulière appelée union-find. Cette structure permet de représenter une partition d'un ensemble (c'est-à-dire un ensemble de sous-ensembles de l'ensemble partitionné disjoints deux à deux et qui recouvrent l'ensemble de départ).

1 Idée générale de la solution

L'objectif de ce projet est de développer une fonction permettant de générer un labyrinthe aléatoirement. Le labyrinthe est découpé en cellules carrées disposées selon une grille elle-même carrée à raison de N cellules par ligne et colonne. Chaque cellule peut être séparée ou non des cellules adjacentes par un mur. L'entrée du labyrinthe sera supposée se faire par le mur gauche de la cellule en haut à gauche et la sortie se fera par le mur droit de la cellule en bas à droite. Tous les autres murs extérieurs seront toujours présents. Un exemple de labyrinthe (pour $N = 5$) est donné à la figure 1. Pour obtenir un labyrinthe visuellement intéressant, on cherche à générer un labyrinthe satisfaisant aux contraintes suivantes :

1. Il ne peut y avoir qu'un seul chemin entre l'entrée et la sortie (si on ne permet pas de revenir sur ses pas)
2. Toutes les cellules doivent être accessibles à partir de l'entrée.
3. Un mur ne peut être absent du labyrinthe que si le reintroduire rendrait une ou plusieurs cellules inaccessibles.

Notez que la première contrainte est en fait une conséquence des deux autres.

Une manière de générer un tel labyrinthe consisterait à procéder de la sorte :

1. On mélange les murs intérieurs du labyrinthe de manière aléatoire.
2. Pour chaque mur intérieur pris dans cet ordre aléatoire, on supprime ce mur du labyrinthe si et seulement si les cellules séparées par ce mur ne sont pas encore accessibles l'une de l'autre (sinon le retirer violera la troisième contrainte).
3. On arrête l'itération précédente dès que toutes les cellules sont accessibles (depuis l'entrée).

La structure union-find permet d'implémenter facilement cette procédure. L'idée est d'utiliser la structure pour maintenir les sous-ensembles de cellules accessibles les unes des autres. Initialement, la structure contiendra un ensemble de singletons correspondant chacun à une cellule (aucune cellule n'étant accessible d'une autre tant qu'aucun mur n'est supprimé). A l'étape 2, on ne supprimera un mur que si les cellules qu'il sépare n'appartiennent pas déjà à un même sous-ensemble dans la structure et en cas de suppression du mur, on fusionnera les deux sous-ensembles contenant les cellules séparées par ce mur. On arrêtera les itérations (étape 3) lorsque la structure ne contiendra plus qu'un seul sous-ensemble contenant tous les nœuds.

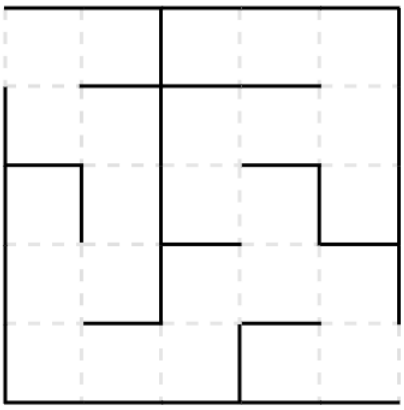

FIGURE 1 – Exemple de labyrinthme

2 Implémentation

Cette section décrit les différentes structures et fonctions qu'on vous demande d'implémenter.

Union-find (fichiers `UnionFindList.c`, `UnionFindTree.c` et `UnionFind.h`)

La structure union-find maintient une partition d'un ensemble d'objets en sous-ensembles disjoints. Pour simplifier l'implémentation, on supposera que les objets sont représentés par les entiers de 0 à $Q-1$, où Q sera supposé connu à la création de la structure. Vous devez implémenter les fonctions suivantes de l'interface :

- ufCreate : qui prend comme argument le nombre d'objets Q et crée une partition constituée des Q singletons contenant les objets 0 à $Q-1$.
- ufUnion : qui réunit les ensembles contenant les deux objets donnés en argument.
- ufFind : qui renvoie un représentant de l'ensemble d'objets qui contient l'objet donné en argument.
- ufComponentsCount : qui renvoie le nombre de sous-ensembles dans la structure.
- ufFree : qui libère l'espace mémoire alloué pour la structure.

On vous demande de fournir (et de comparer ci-dessous) deux implémentations de cette structure : l'implémentation présentée dans les transparents du cours (union-find pondéré par liste-chaînées) et une implémentation à base d'arbres. Plusieurs variantes de cette structure–plus ou moins raffinées–existent. Vous êtes libres d'implémenter celle de votre choix.

Pour vous renseigner sur ces deux dernières implémentations, vous pouvez consulter toutes les sources que vous souhaitez.

Labyrinthe (fichiers `Maze.c` et `Maze.h`)

On vous demande de définir une structure de données pour stocker un labyrinthe implémentant l'interface suivante :

- mzCreate : prenant comme argument une taille de labyrinthe N et créant un labyrinthe complètement fermé de $N \times N$ cellules. Une labyrinthe complètement fermé est un labyrinthe dont chaque cellule est entouré de quatre murs.
- mzMakeRandom : créant un labyrinthe valide aléatoire en suivant la procédure décrite précédemment. Notez que cette procédure nécessite de démarrer d'un labyrinthe complètement fermé.
- mzIsWallClosed : prenant comme argument deux cellules adjacentes et renvoyant `true` si un mur sépare ces deux cellules dans le labyrinthe, `false` sinon.

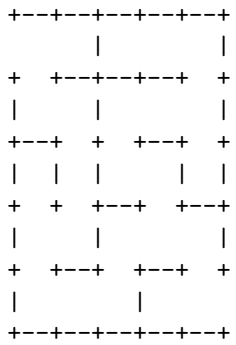

FIGURE 2 – Labyrinthe sous forme ascii

MZSETWALL : prenant comme argument deux cellules adjacentes et un booléen. Si ce booléen vaut `true`, un mur est ajouté entre les deux cellules dans le labyrinthe, si celui-ci n'existait pas. S'il vaut `false`, le mur entre ces deux cellules, s'il existe, est enlevé.

MZSIZE : qui renvoie la taille N du labyrinthe.

MZISVALID : prenant comme argument un labyrinthe et renvoyant `true` si le labyrinthe est valide (c'est-à-dire satisfait aux contraintes plus haut), `false` sinon.

MZPRINT : prenant comme argument un labyrinthe et un pointeur de fichier et imprimant dans le fichier une chaîne de caractères représentant le labyrinthe selon la convention décrite ci-dessous.

MZFREE : qui libère l'espace mémoire alloué pour le labyrinthe.

Une cellule du labyrinthe sera représentée par une paire de coordonnées (i, j) (voir la structure `Coord` définie dans `Maze.h`), avec $i \in \{0, \ldots, N-1\}$ représentant la ligne de la cellule (de haut en bas) et $j \in \{0, \ldots, N-1\}$ représentant la colonne de la cellule (de gauche à droite).

La fonction MZISVALID peut être implémentée sur base du union-find en vous inspirerant de la procédure de génération du labyrinthe décrite plus haut.

Pour l'affichage du labyrinthe, on vous demande d'utiliser *strictement* le codage en chaîne de caractères suivant :

— On mettra un `+` à chaque coin d'une cellule
— Un mur horizontal est représenté par double tiret : `--`
— Un mur vertical est représenté par un `|`
— Un mur absent est représenté par un espace
— Une cellule est représenté par deux espaces
— Chaque ligne, sauf la dernière, se termine par un end-of-line (\n)
— Les ouvertures d'entrées et de sorties doivent être représentés par un espace

Un exemple de labyrinthe valide est donné à la figure 2.

3 Rapport et analyse théorique

On vous demande de répondre aux questions suivantes dans le rapport :

1. **Analyse théorique** :
   (a) Décrivez la variante d'union-find à base d'arbres que vous avez implémentée.
   (b) Donnez, sans les justifier, les complexités en temps au pire et au meilleur cas des deux implémentations des fonctions UFUNION et UFFIND en fonction du nombre d'objets Q présents dans la structure. Citez vos sources.
   (c) Expliquez et justifiez vos choix d'implémentation de la structure de labyrinthe. Décrivez et justifiez toutes les méta-données associées à cette structure.

(d) Donnez le pseudocode des fonctions mzMakeRandom et mzIsValid.

(e) Analysez les complexités en temps au pire et au meilleur cas de ces deux fonctions en fonction de N, la taille du labyrinthe, pour l'implémentation à base de liste-chaînées.

Bonus Analysez les complexités en temps au pire et au meilleur cas de ces deux fonctions en fonction de N, la taille du labyrinthe, pour l'implémentation à base d'arbres.

2. **Analyse empirique** :

(a) Tracez un graphe de l'évolution des temps de calcul de la fonction mzMakeRandom en fonction de la taille du labyrinthe N pour les deux implémentations du union-find. Afin de stabiliser les résultats, faites des moyennes des temps de calcul sur plusieurs exécutions.

(b) Discutez de ces courbes en les mettant en parallèle avec l'analyse théorique.

4 Deadline et soumission

Le projet est à réaliser **par groupe de deux étudiant(e)s** pour le **21 avril 2019, 23h59** au plus tard. Il doit être soumis via la plateforme de soumission (`http://submit.montefiore.ulg.ac.be/`).

Le projet doit être rendu sous la forme d'une archive `tar.gz` contenant :

1. Votre rapport (5 pages maximum) au format PDF. Soyez bref mais précis et respectez bien la numération des (sous-)questions.
2. Les fichiers `UnionFindList.c` `UnionFindTree.c` et `Maze.c`

Vos fichiers seront évalués avec les flags habituels (`--std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes -DNDEBUG`) sur les machines ms8xx. Ceci implique que :

— Les noms des fichiers doivent être respectés.
— Le projet doit être réalisé dans le standard C99.
— La présence de *warnings* impactera négativement la cote finale.
— Un projet qui ne compile pas sur ces machines recevra une cote nulle (pour la partie code du projet).

Notez que le choix d'implémentation de l'union-find se fait à la compilation.

Un projet non rendu à temps recevra une cote globale nulle. En cas de plagiat avéré, l'étudiant se verra affecter une cote nulle à l'ensemble des projets.

Les critères de correction sont précisés sur la page web des projets.

Bon travail !