Université de Liège

Programmation avancée

INFO2050

Projet 2 : Structures de données

Noémie Lecocq
Andrew Sassoye

2017-2018

1 Analyse théorique

a)

Pour implémenter la structure Union-Find à l'aide d'un arbre binaire, nous avons décidé de créer un vecteur dont chaque élément est initialement la racine d'un arbre binaire. Chaque arbre du vecteur représente donc un singleton. Lorsque nous procédons a une union, nous utilisons l'implémentation des arbres binaires vue au cours théorique.

FIGURE 1 – Après UfCreate

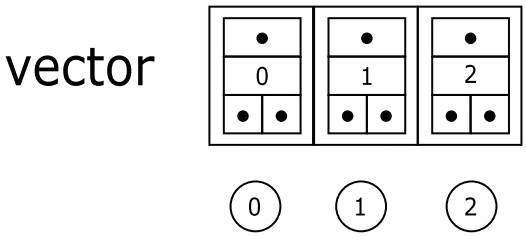
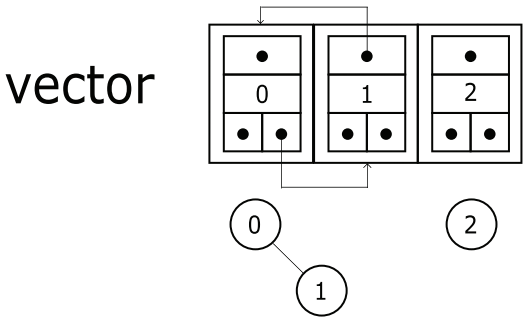


FIGURE 2 – Après UfUnion



b)

La complexité de *ufFind* dans l'implémentation par listes est dans le meilleur cas $\Theta(Q)$ et dans le pire cas $\Theta(Q^2)$.

La complexité de *ufUnion* dans l'implémentation par listes est identique à la complexité de *ufFind* puisqu'elle utilise cette fonction. Nous avons donc $\Theta(Q)$ dans le meilleur cas et $\Theta(Q^2)$ dans le pire cas.

La complexité de *ufFind* dans l'implémentation par arbre est dans le meilleur cas $\Theta(1)$ et dans le pire cas $\Theta(Q)$.

La complexité de *ufUnion* dans l'implémentation par arbre est dans le meilleur cas $\Theta(Q)$ et dans le pire cas $\Theta(Q^2)$ tel que vu au cours théorique.

c) La structure de labyrinthe

La structure maze est composée de 4 éléments.

— size : contient la taille du labyrinthe. Pour une taille N, le labyrinthe sera un carré N x N.

— unionFind : contient l'ensemble disjoint qui a été utilisé pour créer le labyrinthe.

— neighbours : est un vecteur contenant les paires de cellules étant voisines et un booléen égal à *true* si elles sont séparées par un mur, à *false* sinon.

— convert : est une matrice permettant de retrouver le numéro d'une case du labyrinthe à partir de ses coordonnées.

En effet, nous avons choisi d'utiliser la plupart du temps des entiers pour identifier les cases afin de pouvoir utiliser la structure *UnionFind* préalablement définie (et qui contenait des entiers). Ainsi nous représentons chaque case par un chiffre de 0 à $N * N - 1$

d) Pseudocode

```
mzCreate(size)
 1  Maze maze
 2  maze.size = size
 3  maze.neighbours = all pairs of cells that are neighbours with a wall between each of them
 4  maze.unionFind = ufCreate(size)
 5  while ufComponentsCount(maze.unionFind) > 1
 6        (coord1, coord2) = a random pair of neighbours
 7        if There is a wall
 8              mzSetWall(maze, coord1, coord2, false) //Remove the wall
 9              ufUnion(maze.unionFind, coord1, coord2)
10  return maze
```

```
mzIsValid(maze)
1  if ufComponentsCount(maze.unionFind) > 1
2        return false
3  else
4        return true
```

e) La structure de labyrinthe

La structure maze est composée de 4 éléments.

— size : contient la taille du labyrinthe. Pour une taille N, le labyrinthe sera un carré N x N.

— unionFind : contient l'ensemble disjoint qui a été utilisé pour créer le labyrinthe.

— neighbours : est un vecteur contenant les paires de cellules étant voisines et un booléen égal à *true* si elles sont séparées par un mur, à *false* sinon.

— convert : est une matrice permettant de retrouver le numéro d'une case du labyrinthe à partir de ses coordonnées.

En effet, nous avons choisi d'utiliser la plupart du temps des entiers pour identifier les cases afin de pouvoir utiliser la structure *UnionFind* préalablement définie (et qui contenait des entiers). Ainsi nous représentons chaque case par un chiffre de 0 à $N * N - 1$

f) Pseudocode

```
mzCreate(size)
 1  Maze maze
 2  maze.size = size
 3  maze.neighbours = all pairs of cells that are neighbours with a wall between each of them
 4  maze.unionFind = ufCreate(size)
 5  while ufComponentsCount(maze.unionFind) > 1
 6        (coord1, coord2) = a random pair of neighbours
 7        if There is a wall
 8              mzSetWall(maze, coord1, coord2, false) //Remove the wall
 9              ufUnion(maze.unionFind, coord1, coord2)
10  return maze
```

```
mzIsValid(maze)
1  if ufComponentsCount(maze.unionFind) > 1
2        return false
3  else
4        return true
```

g) Complexité en temps avec UnionFindList

La complexité de mzCreate est dans le meilleur cas est de $\Theta(Q^2)$ puisqu'il faut au minimum $Q-1$ unions pour rassembler tous les sous-ensembles et que la complexité au meilleur cas de *ufUnion* est de $\Theta(Q)$.

La complexité au pire cas est difficile à définir puisque l'on choisi un mur à retirer de manière aléatoire parmi tous les "emplacements possibles" de murs. Il faut donc tomber si un emplacement contenant un mur pour avancer dans la fonction.

mzIsValid est constant puisqu'il s'agit simplement d'aller lire la taille de l'*UnionFind*.

2 Analyse empirique

a)

b)

Nous gagnons pas mal de performances en utilisant l'implémentation par arbre. En effet, la courbe a une tangente plus grande pour l'implémentation par liste qui s'explique par la meilleure complexité de la fonction *ufFind* pour l'implémentation par arbre.

FIGURE 3 – Implémentation par liste

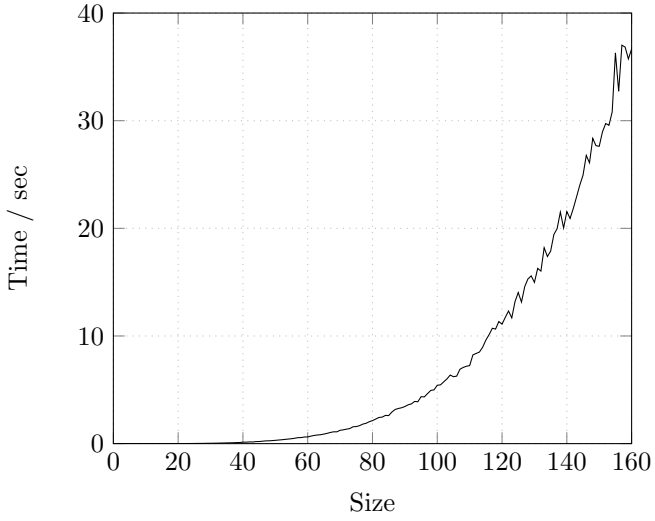


FIGURE 4 – Implémentation par Arbre