INFO2050-1 - Programmation Avancée

Projet 2 : Structures de Données

Aliaksei Mazurchyk
Antoine Sadzot

26 novembre 2017

Université de Liège

1 Analyse Théorique

1.1 Variante d'union-find à base d'arbres choisie

La version de l'union-find en arbre implémentée est une version améliorée. Comme dans la version de base chaque élément pointe vers son parent. Lors d'une union de sous-ensembles, les racines des deux arbres sont fusionnés, un parent se retrouve donc avec deux enfants.

1.1.0.1 Rang La première amélioration consiste à associer un rang a chaque élément. Cela permet d'optimiser la fusion. L'union-find de base pouvait être fort déséquilibré, ce qui rendais l'appel Find() assez long. L'ajout d'un rang permet d'équilibrer l'arbre lors de l'union.

Au début tous les éléments ont un rang 0. Lors d'une fusion de deux arbres de même hauteur, le rang de la nouvelle racine est incrémenté de 1. Si les rangs sont différents, l'arbre ayant le rang le plus petit est ajouté à la racine de l'arbre le plus grand. Cela permet d'équilibrer la structure au fur et à mesure de la construction et ainsi de diminuer la hauteur maximale.

1.1.0.2 Compression de chemin Cette seconde amélioration consiste à utiliser la fonction Find() pour aplatir l'arbre. Cela ce fait lors de l'appel récursif, la racine de l'arbre remonte dans la call-stack et est définie en tant que parent de chaque neud parcouru. De cette manière, tous les éléments parcouru par un Find(), pointeront directement vers la racine et nécessiteront donc un temps constant pour récupérer cette dernière.

1.2 Complexités en temps de ufUnion et ufFind

1.2.1 Implémentation à base de listes chainées

ufFind : Dans tous les cas, la complexité est $\Theta(1)$

ufUnion : Dans le meilleur cas, le plus petit ensemble à lier à la suite du second ensemble ne contient qu'un élément. La complexité dans le meilleur cas est donc $\Theta(1)$. Dans le pire cas, l'ensemble à lier à la suite du second ensemble est de taille N/2, N étant le nombre d'éléments du labyrinthe. La complexité dans le pire cas est donc $\Theta(N/2)$.

1.2.2 Implémentation à base d'arbres

ufFind : Dans le meilleur des cas, la complexité est $\Theta(1)$. Dans le pire est est $\mathcal{O}(h)$, avec h comme auteur de l'arbre.

ufUnion : Dans le meilleur des cas, la complexité est $\Theta(1)$. Dans le pire est est $\mathcal{O}(n + log(n))$.

Complexité combinée : Dans le meilleur des cas, la complexité est $\Theta(1)$. Dans le pire elle est $\Omega(\alpha(n))$, ce qui équivaut à un temps constant car $\alpha(n)$ (fonction d'Ackermann inversée) a une croissance extrêmement lente. En pratique : $\alpha(2^{65536}) = 5$ ce qui garanti une complexité $\Theta(5)$. Cette complexité vaut pour l'*Union* et le *Find*, car les deux améliorations sont complémentaire et l'un tire parti des améliorations de l'autre. Ceci fut démontré par Fredman et Saks en 1989.

1.2.2.1 Sources

- `en.wikipedia.org/wiki/Disjoint-set_data_structure`
- `fr.slideshare.net/NikitaShpilevoy/algorithms-union-find`
- `fr.slideshare.net/WeiLi73/time-complexity-of-union-find-55858534`
- `github.com/emintham/Papers/blob/master/Fredman%2CSaks-%20The%20cell%20probe%20complexity%20of%20dynamic%20data%20structures.pdf`

1.3 Implémentation de la structure du labyrinthe

Les données du labyrinthe sont stockées dans la structure Maze.

```
struct maze_t {
    UnionFind *union_Find;
     struct Walls *myWalls;
     size_t size;
     size_t number_inner_walls;
};
```

La structure "maze_t" contient la structure "UnionFind", qui sera décrite pour chaque implémentation par la suite. Elle contient également la structure "Walls" qui est un vecteur enregistrant tous les murs internes du labyrinthe. La variable "size" contient la taille du labyrinthe. La variable "number_inner_walls" contient le nombre de mur internes du labyrinthe. C'est également la taille du vecteur de type "Walls".

```
typedef struct Walls {
    Coord Cell1;
    Coord Cell2;
    bool wall_between;
};
```

La structure "Walls" contient les coordonnées de deux cellules adjacentes, partageant un mur. La variable booléenne "wall" vaut "true" si le mur est présent. Elle vaut "false" si le mur est absent.

1.3.1 Implémentation par listes chainées

La structure union_find_t est décrite de cette manière :

```
struct union_find_t {
    struct Element* elements;
    struct Sentinel* sentinels;
    size_t numberComponents;
};
```

Elle contient un vecteur "elements" contenant tous les élements du labyrinthe structurés en listes chainées. Elle contient également un vecteur "sentinels" contenant toutes les sentinelles pointant sur les premiers et derniers éléments de chaque liste chainée. La variable "numberComponents" contient le nombre d'ensembles distincts.

```
typedef struct Element {
    size_t numero;
    struct Sentinel* head;
    struct Element* next;
}Element;
```

Le "numero" correspond à l'indice de l'élément dans le labyrinthe. Le pointeur "head" pointe vers la sentinelle de l'ensemble auquel appartient l'élément. Le pointeur "next" pointe vers l'élement suivant dans la liste chainée.

```
typedef struct Sentinel {
    struct Element* first;
    struct Element* last;
    size_t numberElements;
}Sentinel;
```

Le pointeur "first" pointe vers le premier élément de la liste chainée liée à la sentinelle. Le pointeur "last" pointe vers le dernier élément de cette liste. La variable "numberElements" contient le nombre d'éléments présents dans cette liste.

1.3.2 Implémentation par arbres

La structure union_find_t est déclarée ainsi :

```
struct union_find_t {
    size_t* items;
    size_t* parrents;
    size_t* rank;
    size_t n_items;
    size_t n_trees;
};
```

Cette structure est composée de 3 tableaux :

- **items[n] :** Indices dans la grille du labyrinthe.
- **parents[n] :** Parent direct de l'item située à l'indice n.
- **rank[n] :** Rang de l'item de l'item située à l'indice n

Deux autres variables sont définies :

- **n_items :** Nombre total d'éléments.
- **n_trees :** Nombre d'ensembles.

Cette structure simplifie grandement le parcourt des ensembles disjoints, les pointeurs étant masqués derrière des tableaux.

1.4 Pseudocode des fonctions mzCreate et MzIsValid

$mzIsValid(maze)$

```
1  if ufComponentsCount(maze->union_Find) > 1
2      return false;
3  else
4      return true;
```

Par soucis de simplicité, certaines séquences d'instructions évidentes ont été simplifiées en appels de fonctions. Comme par exemple l'initialisation des murs.

$mzCreate(size)$

```
 1  innerWalls = NumberInnerWalls(size)
 2  myMaze->size = size
 3  myMaze->union_find = ufCreate(size * size)
 4  myMaze->number_inner_walls = innerWalls
 5  InitializeWalls(myMaze->myWalls, size * size)
 6  parcours = SetUpParcoursOrder(myMaze->number_inner_walls)
 7  wallsToTest = 0
 8  while !mzIsValid(myMaze)&&wallsToTest < innerWalls
 9  visits = parcours[wallsToTest]
10      indexCell1 = IndexFromCoord(myMaze->myWalls[visits].Cell1)
11      indexCell2 = IndexFromCoord(myMaze->myWalls[visits].Cell2)
12      status = ufUnion(myMaze->union_Find, indexCell1, indexCell2)
13      if status == UF_MERGED
14          close = mzIsWallClosed(myMaze, myMaze->myWalls[visits].Cell1,
    myMaze->myWalls[visits].Cell2)
15          mzSetWall(myMaze, myMaze->myWalls[visits].Cell1,
    myMaze->myWalls[visits].Cell2, close)
16      wallsToTest++
17  return myMaze
```

1.5 Analyse de la complexité en temps de mzCreate et mzIsValid

1.5.1 Implémentation à base de listes chainées

mzIsValid : La fonction ufComponentsCount s'écrit comme telle :

$ufComponentsCount(union_Find)$

1 **return** $union_Find -> numberComponents$

Dans tous les cas, sa complexité est $\Theta(1)$.

mzCreate : Seules les parties de la fonction présentant une complexité égale ou supérieure à N^2 sont présentées. Les autres sont ignorées car négligeables. **Dans tous les cas.** La ligne 3 est $\Theta(N^2)$ car chaque cellule est visitée une à une. A la la ligne 5, N^2 itérations sont nécessaires car toutes les cellules sont visitées. Donc la complexité est $\Theta(N^2)$.

A la ligne 6, chaque mur intérieur est visité, ce qui fait $2N^2$ - 2N opérations, donc $\Theta(N^2)$. La boucle à la ligne 8 va varier en fonction du meilleur et du pire cas.

Dans le meilleur cas, chaque itération permet une connexion de cellules. Cela fait donc N^2-1 itérations nécessaires. Au sein de la boucle, toutes les instructions on une complexité linéaire. C'est également le cas pour la ligne 11 dans le meilleur des cas. Car à chaque connexion d'ensembles, seul un élément est connecté à la fois. Donc, **dans le meilleur cas**, la complexité en temps est de l'ordre de $\Omega(N^2)$.

Dans le pire cas, tous les éléments du vecteur myWalls sont visités. Cela fait donc $2N^2$ - 2N itérations. Il n'y aura cependant toujours que N^2-1 opérations de connexion de cellules. Donc seule la ligne 11 aura une complexité variable. L'analyse de la complexité du ufUnion va se faire pour les cas où N est une puissance de 2, par facilité. Le pire cas correspond à celui où les ensembles connectés sont toujours de taille égale. Au début, tous les ensembles seront de taille 1, puis de taille 2, 4, 8,... jusqu'à ce qu'il ne reste plus qu'un ensemble. En prenant le problème à l'envers, on part d'un ensemble de N^2 éléments à N^2 ensemble d'un éléments :

$$T(1) = 0 T(m^2) = \frac{N^2}{2} + T(\frac{m^2}{2}) \text{ pour } m = N \tag{1}$$

Par Plug-and-Chug, on obtient la solution suivante :

$$log_2(N^2) * \frac{N^2}{2} \tag{2}$$

Ce qui est la complexité totale de la ligne 11. Le pire cas de la boucle qui démarre en ligne 8 influencera donc la complexité totale. **Dans le pire cas**, la complexité est de l'ordre de $\Omega(log_2(N^2) * N^2)$ On observe que la complexité en temps du pire et du meilleur cas sont proches, à un multiple de N^2 près.

2 Analyse empirique

2.1 Tests de performance

FIGURE 1 – List

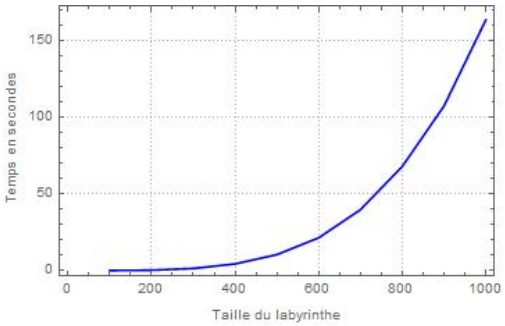


FIGURE 2 – Tree

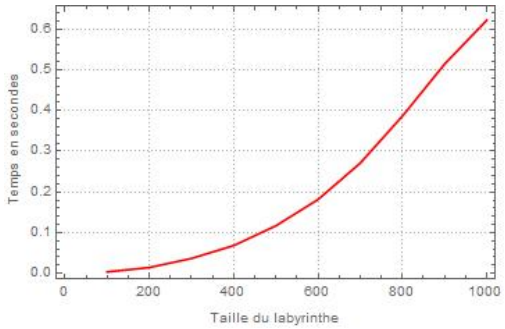


FIGURE 3 – List et Tree

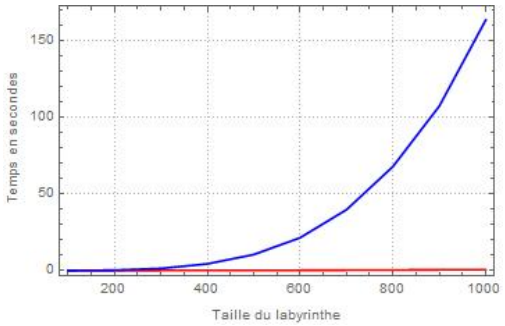
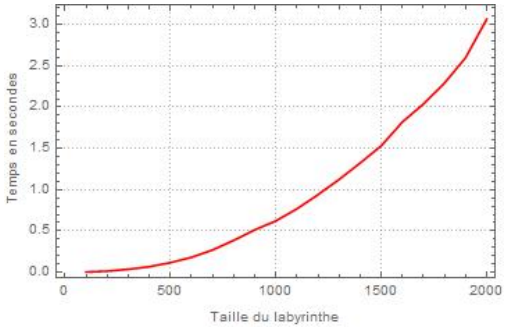


FIGURE 4 – Tree grande taille



2.2 Analyse des résultats

La figure 1 montre que le temps de calcul croit très rapidement avec le carré de de la taille du labyrinthe. Ceci semble concorder avec les analyses théoriques combinées de l'*unionFindList* et de *mzCreate*.

Sur la figure 2 on peut remarquer que l'implémentation de *unionFindTree* croit très lentement, presque linéairement à partir de la taille 800. Des test plus nombreux furent réalisés ce qui donna la figure 4. On y voit que la courbe croit très lentement. Le temps dédié aux opération Union et Find devant être constant en pratique (démontré en 1989 par Fredman et Saks), on peut supposer que la légère croissance vient des opérations de *mzCreate*.

La comparaison entre l'implémentation *List* et *Tree* illustrée par la figure 3 montre que la structure en arbre est bien plus performante. Pour un tableau de taille 1000, 10^6 éléments sont gérés par l'Union-Find. La génération du labyrinthe en utilisant la structure en liste nécessite 160 secondes, alors que celle en arbre ne prend que 0.6 secondes. Ceci est principalement dû aux améliorations implémentées à la structure en arbre (rang et compression de chemin).