Faculté des Sciences appliquées
INFO0902-1 : Structures de données et algorithmes

Modèle de Percolation

Professeur :
GEURTS Pierre

Groupe :
LAMBERMONT Romain
LOUIS Arthur

15 avril 2021

Table des matières

Table des figures

1 Analyse Théorique

1.1 Description `UnionFindTree.c`

Dans le cadre de ce projet nous avons réalisé une structure d'UnionFind utilisant le principe de *Path Compression*, qui permet de réduire la profondeur de l'arbre à chaque appel de la fonction `ufFind`. Ce principe réduit considérablement le temps d'accès à un ensemble et d'union à un autre par la fonction `ufUnion`.

1.2 Complexités `ufUnion` et `ufFind`

Pour l'implémentation par liste liée, nous avons gardé dans la structure de chaque noeud un pointeur vers sa liste. De plus, lors de l'union de deux ensembles, nous ajoutons toujours le plus court ensemble au plus long ensemble grâce à un champ de la structure liste qui garde à jour le nombre d'éléments que possède la liste. Pour cette implémentation, l'opération `ufFind` est $\mathcal{O}(1)$ quoi qu'il arrive et `ufUnion` est dans le pire des cas $\mathcal{O}(n/2)$ et dans le meilleur des cas, elle est $\Omega(1)$.

En ce qui concerne l'implémentation par arbre, nous avons gardé dans la structure de chaque noeud un pointeur vers son parent direct. Nous avons profité des opérations `ufFind` pour procéder à une *Path Compression* et lors des `ufUnion`, nous relions le plus petit arbre au grand, c'est la *Fusion Optimisée des Racines*. Cela permet de réduire le nombre d'opérations nécessaires pour atteindre la racine du plus petit arbre. Cela permet de diminuer la profondeur de l'arbre et donc de réduire le temps de calcul pour le prochain appel à la fonction `ufFind`. Pour cette implémentation, la complexité dans le pire des cas de `ufFind` est $\mathcal{O}(n)$ mais cela est compensé par la path compression qui fait en sort que ce cas ne se présente jamais dans la pratique et dans le meilleur des cas, elle est $\mathcal{O}(1)$. Pour finir, l'implémentation de `ufUnion` est dans le pire des cas $\mathcal{O}(n/2)$ et dans le meilleur des cas $\Omega(1)$.

Voici les sources utilisée pour répondre à cette question :

- `https://stackoverflow.com/questions/53149097/why-is-the-time-complexity-of-performing-n-union-find-union-by-size-operations`
- `https://fr.wikipedia.org/wiki/Union-find`
- `https://algorithms.tutorialhorizon.com/disjoint-set-union-find-algorithm-union-by-rank-and-path-compression/`

1.3 Structure `Percolation.c`

Pour notre structure percolation, nous avons fait le choix de travailler avec trois champs : une size, qui contient la taille de la grille (size $\times$ size), une matrice de booléens (type ajouté par le librairie `stdbool.h`) qui détermine si une case est ouverte ou fermée (`false` = case fermée, `true` = case ouverte) et un `UnionFind` qui permettera de réaliser les opérations `ufUnion` et `ufFind`.

Nous allons maintenant vérifier que les fonctions de l'interface de `Percolation.c` respectent bien les conditions demandées :

- `percCreate` : Cette fonction réalise $n+3$ allocations de mémoire pour la structure et $2n$ appels à une fonction de l'interface d'`UnionFind`
- `percFree` : Cette fonction réalise n appels à la matrice de booléens et trois appels constants à la structure
- `percSize` : Cette fonction réalise un seul appel à la structure
- `percOpenCell` : Cette fonction réalise des appels constants à la structure et des appels à des fonctions de l'interface d'`UnionFind`
- `percIsCellOpen` : Cette fonction réalise un seul appel à la structure
- `percIsCellFull` : Cette fonction réalise un seul appel à la structure et deux appels à des fonctions de l'interface d'`UnionFind`
- `percPercolates` : Cette fonction réalise deux appels à la structure et deux appels à des fonctions de l'interface d'`UnionFind`
- `percPrint` : Vérification non nécessaire

On vérifie bien que les conditions sont respectées.

1.4 Complexité en temps et espace de `thresholdEstimate` par listes

Lorsque nous réalisons T percolations pour une grille de $N \times N$ items et que nous ouvrons m cellule, la complexité en temps dans le pire des cas peut s'écrire :

$$\mathcal{O}(T \times (N^2 + N + m \times (k + N/2)))$$

où k est le nombre de coordonnées alétoires nécessaires pour trouver une cellule fermée.

La complexité en espace peut s'écrire :

$$N^2 + T + \log T$$

Le $\log T$ est du au `QuickSort` du tableau des résultats.

1.5 Complexité en temps et espace de `thresholdEstimate` par arbres

Lorsque nous réalisons T percolations pour une grille de $N \times N$ items et que nous ouvrons m cellule, la complexité en temps dans le pire des cas peut s'écrire :

$$\mathcal{O}(T \times (N^2 + N + m \times (k + \frac{3 \times N}{2})))$$

où k est le nombre de coordonnées alétoires nécessaires pour trouver une cellule fermée.
La complexité en espace est la même qu'au cas précédent.

2 Analyse Empirique

2.1 Centilles et seuil critique de percolation

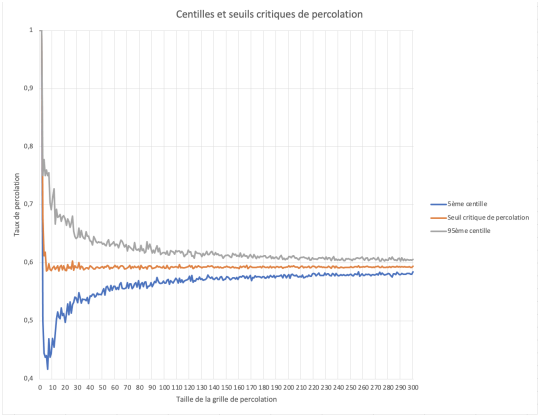

FIGURE 1 – Graphique des centilles et seuils critiques de percolation en fonction de la taille de la grille

En observant ces courbes, on remarque qu'au départ, avec une taille de grille 1×1, les valeurs du tableau de résultats, représenatant dans l'ordre les 5èmes centilles, le seuil critique de percolation et les 95èmes centilles, que les trois valeurs démarrent logiquement toutes à la valeur 1, en effet une grille 1×1 percole forcément.
En augmentant la taille, dans notre cas, jusque 300×300, on remarque très logiquement que la courbe des 5èmes et 95èmes centilles se retrouvent respectivement au-dessus et en-dessous de la courbe de seuil critique de percolation tout en convergeant vers celle-ci. En observant la courbe de seuil, on se rend compte que cette valeur tend vers 59%, valeur plutôt logique.

2.2 Temps d'exécution en fonction de l'implémentation

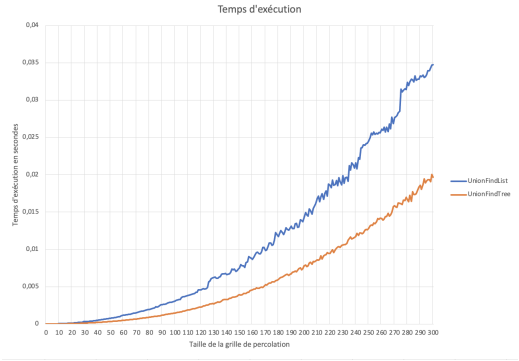

FIGURE 2 – Graphique des temps d'exécution en fonction de la méthode d'implémentation et de la taille de la grille

En observant le graphique obtenu ci-dessus, on remarque que pour des petites grilles (jusque 40×40), les temps de calculs sont fort similaires et ce pour une bonne, raison. Les amélioration

du *Path Compression* et de la *Fusion Optimisée des Racines* n'ont pas encore eu le temps de se démarquer par rapport à la fonction `UnionFindList.c`. Par après, les deux courbes se séparent car le `ufFind` de l'implémentation sous forme de liste devient alors plus lent et moins efficace.

2.3 Discussion des courbes

Si on observe ces courbes et en les comparant à l'analyse théorique réalisée plus haut, tout fait sens. En effet, après discussion de notre implémentation de la structute `UnionFind` dans la fonction `UnionFindTree.c`, on sait que par le fait de ne pas avoir réalisé une implémentation naïve de celle-ci, nous avons amélioré ses performances (avec le *Path Compression* et la *Fusion Optimisée des Racines*), la rendant ainsi plus efficace que `UnionFindList.c`.

Pour ce qui est du graphique des centilles et du seuil de percolation, tout est également logique. Il est normal que les centilles se retouvent respectivement au-dessus et en-dessous de la courbe seuil. Et cette même courbe seuil fait également sens car beaucoup de modèles montrent un seuil avoisinant les $59\% \pm 0.1$ (source : `http://percolation.free.fr/theseweb006.html`).