Master Informatique M1

2018/2019

Algorithme et complexité
Partie 1

Thiebaud MARBACH
Alexis ABLI-BOUYO

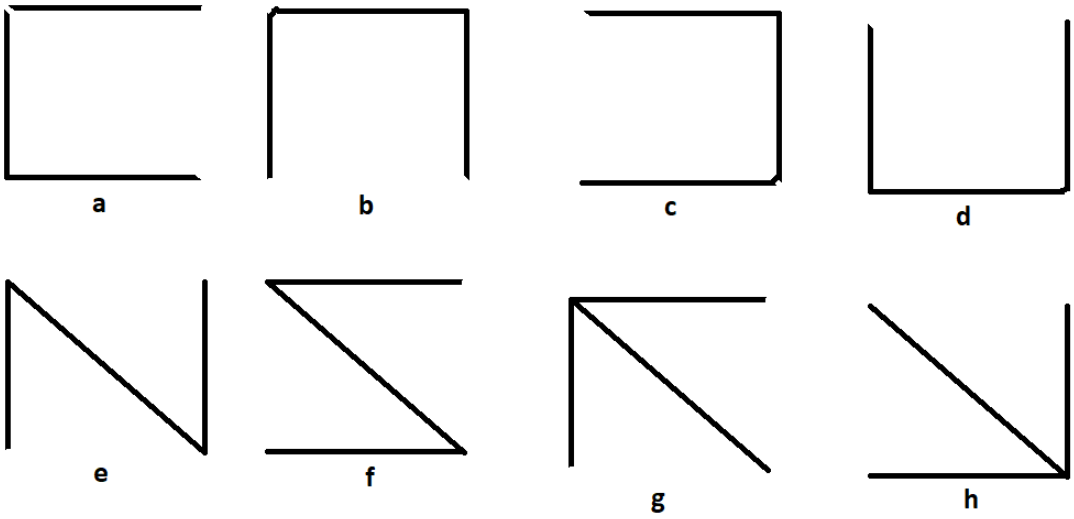
Répartition des tâches:

Bien que la plupart des réflexions de la Partie 1 ont été faites en binômes, le développement a été réparti de la façon suivante:

***Thiebaud**:* Algo de Kruskal et Algo d'Aldous-Broder
Alexis: Algo de Wilson et Gestion des labyrinthes

La partie 2 a été totalement faite en binôme.

Q1) En essayant d'obtenir les solutions de façon empirique, on obtient les 8 résultats suivants:



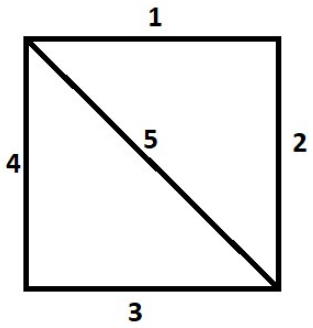
Q2) L'algorithme est implémenté dans la classe Kruskal du projet. La méthode kruskal(Graph g) retourne l'arbre couvrant minimum sous la forme d'un ArrayList d'Edge, tandis que la méthode kruskalHashCode(Graph g), qui ne fonctionne efficacement que sur les petits graphes comme le graphe de l'exemple, retourne un "hashcode" de l'arbre couvrant, ce qui permet de les comparer efficacement.

Q3) La classe main de la classe Kruskal lance un million de fois l'algorithme et compare les résultats de la méthode kruskalHashcode pour distinguer les différents arbres:

```
arbre de code 385 : 116908
arbre de code 66 : 133668
arbre de code 165 : 116463
arbre de code 70 : 133076
arbre de code 231 : 116688
arbre de code 105 : 116894
arbre de code 154 : 133388
arbre de code 30 : 132915
```

Comme on peut le voir, certains arbres semblent apparaître environ 133 000 fois tandis que d'autres n'apparaissent que 117 000 fois.

Q4) Dans notre exemple, les deux premières arêtes choisies seront forcément gardées, et seule la troisième dépendra réellement de l'aléatoire.
Pour les reconnaître plus facilement, nous allons nommer les arêtes comme montré ci dessous:



Voici un tableau décrivant tous les cas de figure possible:

Choix initiaux	**Arbres obtenables**	**Probabilité par choix**
1 et 2	b c	1/2
1 et 3	a f c	1/3
1 et 4	a b g	1/3
1 et 5	f g	1/2
2 et 3	c d h	1/3
2 et 4	b d e	1/3
2 et 5	e h	1/2
3 et 4	a d	1/2
3 et 5	f h	1/2
4 et 5	e g	1/2

Voici la probabilité de sortie pour chaque arbre:
(par exemple pour a: 1/10 * ⅓ + 1/10 * ⅓ + 1/10 * ½ = 0,117 environ)

Arbre	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Probabilité	**0.117**	**0.117**	**0.117**	**0.117**	**0.133**	**0.133**	**0.133**	**0.133**

On constate donc que tous les arbres n'ont pas la même probabilité d'apparaître.

Q5) L'algorithme est implémenté dans la classe AldousBroder du projet. La méthode aldousBroder(Graph g) retourne l'arbre couvrant minimum sous la forme d'un ArrayList d'Edge, tandis que la méthode aldousBroderHashCode(Graph g), qui ne fonctionne efficacement que sur les petits graphes comme le graphe de l'exemple, retourne un "hashcode" de l'arbre couvrant, ce qui permet de les comparer efficacement.

La classe main de la classe AldousBroder lance un million de fois l'algorithme et compare les résultats de la méthode aldousBroderHashCode pour distinguer les différents arbres:

```
arbre de code 385 : 125167
arbre de code 66 : 124787
arbre de code 165 : 124922
arbre de code 70 : 125010
arbre de code 231 : 125258
arbre de code 105 : 124881
arbre de code 154 : 125012
arbre de code 30 : 124963
```

On voit bien que le nombre de chaque arbre tend vers 125000, ce qui semble indiquer que les arbres ont la même probabilité d'apparaître.

Q6) L'algorithme est implémenté dans la classe Wilson du projet. La méthode wilson(Graph g) retourne l'arbre couvrant minimum sous la forme d'un ArrayList d'Edge, tandis que la méthode wilsonHashCode(Graph g), qui ne fonctionne efficacement que sur les petits graphes comme le graphe de l'exemple, retourne un "hashcode" de l'arbre couvrant, ce qui permet de les comparer efficacement.
La classe main de la classe Wilson lance un million de fois l'algorithme et compare les résultats de la méthode wilsonHashCode pour distinguer les différents arbres:

```
arbre de code 385 : 125222
arbre de code 66 : 125122
arbre de code 165 : 124821
arbre de code 70 : 125120
arbre de code 231 : 124569
arbre de code 105 : 124903
arbre de code 154 : 125146
arbre de code 30 : 125097
```

On voit également que le nombre de chaque arbre tend vers 125000, ce qui semble indiquer que les arbres ont la même probabilité d'apparaître.

Q7) La méthode main de la classe Labyrinthe permet d'afficher un arbre couvrant 20x20 généré avec l'un des trois algorithmes précédents, selon le choix indiqué. L'entrée du labyrinthe est le sommet 0 et la sortie est le sommet 399.

Q8) Avec la méthode main de la classe Labyrinthe, on peut obtenir la moyenne du nombre de culs-de-sac et la moyenne de la distance entrée-sortie sur un échantillon de 1000 labyrinthes générés par l'algorithme choisi.

Résultats pour Kruskal (très long à l'exécution):

```
Etape 997 sur 1000
Etape 998 sur 1000
Etape 999 sur 1000
Etape 1000 sur 1000
Moyenne de culs de sac: 119
Moyenne de distance de l'entree a la sortie: 56
```

Résultats pour Aldous-Broder (quelques secondes d'exécution):

```
1: Labyrinthe 20x20 Kruskal
2: Labyrinthe 20x20 AldousBroder
3: Labyrinthe 20x20 Wilson
4: Stats de 1000 labyrinthes 20x20 Kruskal
5: Stats de 1000 labyrinthes 20x20 AldousBroder
6: Stats de 1000 labyrinthes 20x20 Wilson
5
Moyenne de culs de sac: 114
Moyenne de distance de l'entree a la sortie: 59
```

Résultats pour Wilson (quelques secondes d'exécution):

```
1: Labyrinthe 20x20 Kruskal
2: Labyrinthe 20x20 AldousBroder
3: Labyrinthe 20x20 Wilson
4: Stats de 1000 labyrinthes 20x20 Kruskal
5: Stats de 1000 labyrinthes 20x20 AldousBroder
6: Stats de 1000 labyrinthes 20x20 Wilson
6
Moyenne de culs de sac: 115
Moyenne de distance de l'entree a la sortie: 59
```

D’après nos résultat, il semblerait que la méthode de création de labyrinthe importe peu si l’objectif est d’améliorer “l’efficacité” du labyrinthe.

.