

This assignment will be closed on May 02, 2023 (23:59:59).

On April 20, 2023 (11:20:27), we had submissions for the following questions: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 — [Details](#)
🔗 [./ags/INF371/TD01/2022/uploads/:my/](https://ags/INF371/TD01/2022/uploads/:my/).

Recherche de seuil de percolation

Une application d'Union-Find

Benjamin Werner, Julien Signoles, Sylvie Putot, Xavier Rival

-
- [Barème](#)
 - [Introduction](#)
 - [Documentation sur l'environnement de travail](#)
 - [Échauffement](#)
 - [Exercices de base](#)
 - [Seuil de percolation](#)
 - [Union-Find](#)
 - [Optimisations](#)
-

Barème

Ce TD est sur deux séances, donc deux semaines. Il y a 12 exercices au total, après un échauffement vous permettant de prendre en main votre environnement de travail. Idéalement, il faudrait être aux alentours de la question 6 au début de la deuxième séance. Voici le barème de ce double TD pour avoir les notes A, B ou C:

- A : exercices 1-12 (tout le TD)
- B : exercices 1-9
- C : exercices 1-7

Introduction

Ce TD commence par une petite introduction à l'environnement de travail et à la prise en main de Java à travers quelques programmes simples en Java.

Ensuite, l'objectif du TD, qui se déroulera sur deux séances, est de déterminer s'il y a ou non percolation à travers une grille du plan pavée de cases noires et blanches, c'est-à-dire s'il existe ou non un chemin de cases noires allant du haut au bas d'une telle grille. Ensuite, il s'agira d'estimer la proportion de cases noires nécessaires à ce qu'il y ait percolation. Cette proportion est appelée le seuil de percolation.

Pour cela, vous aurez à réaliser un programme, découpé en petites fonctions élémentaires, que vous optimiserez ensuite étape par étape en étant guidé(e) par le sujet. N'oubliez pas de tester les fonctionnalités que vous écrivez chaque fois que possible, et au fur et à mesure !

Vous manipulerez ici les fonctions et structures de contrôle classiques de Java, et vous y ferez un usage intensif des tableaux (entiers et booléens) et des instructions de contrôle fondamentales. Vous programmerez notamment une structure de données appelée [Union-Find](https://fr.wikipedia.org/wiki/Union-find) (<https://fr.wikipedia.org/wiki/Union-find>), permettant de représenter efficacement des classes d'équivalence à l'aide de leurs représentants canoniques. Elle permet d'optimiser la résolution de certains problèmes dans des domaines variés, par exemple en théorie des graphes ou en preuve automatique.

À partir de l'exercice 3, vous devrez soumettre vos solutions *via* le formulaire présent à la fin de chaque exercice. Il est possible de soumettre plusieurs fichiers. Il est également possible de resoumettre une solution si vous y avez apporté des changements, et ce n'est pas un problème si le fichier contient plus de fonctions que demandé, par exemple parce que vous avez déjà fait la question suivante. Le site vous donnera un retour automatique sur vos soumissions, sous forme d'un feu

tricolore en haut à gauche de l'écran. Cliquez dessus pour en savoir plus. Le bandeau vert `Last submission: date` signifie seulement que vous avez déposé un fichier, il ne dit rien sur les tests. Il est important de comprendre qu'un retour positif ne garantit pas que votre code soit bon. Les tests ne détectent que certaines erreurs simples. En cas de problème sur une question, il faut donc aussi envisager la possibilité que cela soit causé par le code d'une question antérieure, même si les tests n'avaient rien détecté à cette question.

Documentation sur l'environnement de travail

La version de Java nécessaire pour les TDs est Java 1.8. Pour écrire vos programmes, nous vous conseillons d'utiliser l'environnement de développement Eclipse (http://www.enseignement.polytechnique.fr/informatique/INF361/TD_X18-2019/INF311.clone/pratique/eclipse/intro_eclipse/demarrer_eclipse.html), qui fonctionne sur la plupart des systèmes d'exploitation (notamment Linux, OSX et Windows). Des instructions pour installer Java 1.8 et Eclipse sont fournies sur Moodle (<https://moodle.polytechnique.fr/course/view.php?id=12380#section-15>).

L'utilisation d'Eclipse n'est que conseillée : vous pouvez utiliser celui de votre choix, notamment si vous êtes déjà à l'aise avec un en particulier.

Échauffement

L'exercice qui suit permet de vérifier que l'on maîtrise les trois phases de la mise en oeuvre d'un programme. Pour les réaliser, il n'est pas nécessaire de comprendre le langage Java. Pour l'instant, il suffit de suivre fidèlement les instructions données ci-dessous.

1. Commencer proprement.

Une fois Eclipse lancée, créer un nouveau projet Java que vous appellerez `warmup` en cliquant sur `File > new > Java project`. Si `Java project` n'est pas proposé, choisissez d'abord `Project`, et l'option `Java Project` devrait apparaître dans la fenêtre suivante. Sélectionnez bien comme environnement d'exécution JRE `JavaSE-1.8`, cela réduira les risques de problèmes lors de la soumission des fichiers.

Ce projet apparaît maintenant dans le `Package explorer` à gauche de votre fenêtre Eclipse (si Eclipse est revenu à l'écran d'accueil, vous pouvez cliquer sur la flèche `workbench` en haut à droite). Il contient la bibliothèque standard de Java (`JRE System Library` mais la dénomination peut varier en fonction de votre installation) et un répertoire `src` vide. C'est ce dernier qui contiendra les fichiers sources Java de votre projet.

Nous vous conseillons pour la suite de créer un projet par sujet de TD (mais pas par exercice).

2. Programme vide.

Créer une classe Java que vous appellerez `Empty`, par exemple en cliquant avec le bouton droit de la souris sur le répertoire `src`, puis sur `New > Class`. Dans la boîte de dialogue ouverte, outre indiquer le nom de la classe, supprimer `warmup` de la ligne `Package` et cocher la case `public static void main(String[] args)` pour créer automatiquement la fonction principale de votre programme qui sera appelée au début de son exécution.

Eclipse a automatiquement généré le fichier `Empty.java` dans le répertoire `src`, tout en ajoutant une indication (`default package`) (la notion de package sera expliquée ultérieurement). Ce fichier contient le programme suivant :

```
public class Empty {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Le sens exact des différentes lignes du programme et, en particulier, des mots-clés `public`, `class` et `static` sera introduit dans les semaines à venir, lorsque les concepts de la programmation orientée objets vous seront enseignés. En attendant de lever leurs mystères, considérez-les comme des conventions syntaxiques obligatoires.

Tout particulièrement, la fonction principale de votre programme doit obligatoirement avoir le prototype `public static void main(String[] args)` pour être reconnue comme telle par le compilateur Java.

Le tableau de chaînes de caractères `args` en argument de cette fonction correspond aux arguments communiqués au programme par l'utilisateur via la ligne de commandes.

3. **Compilation et exécution du fichier `Empty.java` .**

Les phases de compilation et d’exécution, théoriquement distinctes, se font de manière combinée dans Eclipse en cliquant sur le bouton `Run` . Les résultats de l’exécution apparaissent en bas de la fenêtre, dans l’onglet `Console` . Néanmoins, comme ce programme ne fait rien (le corps de la fonction `main` est vide, hormis une ligne de commentaire), aucun résultat n’apparaît ici.

Pour s’assurer que votre programme a bien été compilé et exécuté, modifier le pour introduire une erreur, par exemple en supprimant le mot-clé `static` . Si vous essayez à nouveau de compiler et d’exécuter ce programme, vous devez obtenir l’erreur suivante dans la console :

```
Erreur : la méthode principale n'est pas static dans la classe
Empty, définissez la méthode principale comme suit : public static
void main(String[] args)
```

Corriger cette erreur en remettant le mot-clé précédemment enlevé et vérifier que l’erreur a disparu.

4. **Programme `Hello` .**

On se propose maintenant de modifier ce programme pour qu’il affiche le message “Hello”. Pour cela, nous allons faire une nouvelle classe `Hello` dans le même projet. Vous pouvez la créer comme précédemment ou copier la classe vide.

Après avoir ouvert le fichier `Hello.java` en double-cliquant dessus, insérer l’instruction suivante entre les accolades de sa fonction `main` :

```
System.out.println("Hello");
```

On doit prendre tout de suite l’habitude de ne mettre qu’**une instruction par ligne**, d’**éviter les lignes trop longues**, et d’**indenter correctement** ses programmes. Eclipse vous aide sur ce dernier point.

Compiler et exécuter ce programme, puis vérifier que le message de bienvenu apparaît dans la console.

5. **Fichier vs programme.**

Le nom du programme, ici `Hello` , qui figure dans la première ligne `class Hello` n’est pas nécessairement identique au nom du fichier. On peut ainsi remplacer `class Hello` par `class HelloProgram` .

Néanmoins, pour qu’Eclipse sache quel programme exécuter dès lors qu’il ne porte plus le même nom que le fichier dans lequel il est contenu, il faut ouvrir la boîte de dialogue de configuration de l’exécution *via* `Run > Run Configurations` (aussi accessible en cliquant sur le petit triangle pointant vers le bas juste à côté du bouton `Run`). Dans cette boîte de dialogue, il faut remplacer `Hello` par `HelloProgram` comme classe principale.

Il faut en outre supprimer le mot-clé `public` à gauche de la classe. Ce point sera expliqué lors d’un prochain cours.

Vérifier la compilation et l’exécution de votre programme `HelloProgram` .

6. **Cas d’un programme qui ne termine pas.**

Si on insère :

```
while (true)
```

avant

```
System.out.println("Hello");
```

on obtient un programme qui affiche `Hello` sans fin. Pour l’interrompre, il faut cliquer sur le carré rouge à droite de la console.

Exercices de base

Les exercices qui suivent pointent un certain nombre de sources d’erreurs (ou d’incompréhension) fréquentes. Ils ont donc pour but que vous les évitiez par la suite.

Pour ces exercices, continuez à utiliser le projet `warmup` en y ajoutant de nouvelles classes.

Exercice 1 : les types primitifs

1. Calculer et afficher $20!$, en utilisant une variable `i` de type `int` puis une variable `n` de type `long` . Pouvez-vous expliquer la différence de comportement ?
2. Quel est le résultat des instructions suivantes placées dans une fonction `main` :

```
double d = 0;
d = 1 + 1/2 + 1/3;
System.out.println(d);
```

Que faire pour obtenir un résultat plus conforme à l’intuition ?

3. Écrire un programme qui calcule et affiche à l’écran les quantités :

$$2 + 2$$
$$e + \pi$$

Les constantes mathématiques qui interviennent ici sont des champs statiques de la classe `java.lang.Math` dont voici la [documentation](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html) (<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>). Par exemple, pour utiliser la constante π , il suffira d’écrire `Math.PI` dans votre programme.

Exercice 2 : déboguer un programme

On programme rarement sans faire d’erreur. Il est donc important de savoir détecter les erreurs dans un programme et les corriger rapidement. Pour cela, le programmeur dispose de plusieurs outils qui lui facilitent considérablement la tâche :

- les erreurs indiquées à la volée par Eclipse *via* des croix rouges dans la marge lorsque vous éditez votre programme ;
- les messages d’erreur du compilateur à regarder dans l’ordre d’émission : ils sont relativement explicites en Java ; en particulier, le numéro de la ligne où est détectée la première erreur est souvent pertinent ;
- les messages d’erreur à l’exécution ;
- l’affichage de valeurs intermédiaires quand le programme ne produit pas le bon résultat ;
- l’utilisation d’un débogueur comme celui intégré à Eclipse même si nous n’approfondirons pas ce point en INF371.

Il est très important de vérifier l’absence de croix rouges et de compiler et tester son programme très régulièrement, par exemple après l’écriture de chaque fonction.

Pour vous entraîner à déboguer rapidement, on vous fournit plusieurs versions erronées d’un même programme censé calculer la valeur de $n!$ où n est un entier entré au clavier par l’utilisateur.

Pour cela, téléchargez l’archive (zip) [Bug.zip \(resources/bugs/Bug.zip\)](#). Elle contient huit fichiers Java nommés de `Bug1.java` à `Bug8.java` . Pour les ajouter à votre projet `warmup` , vous pouvez cliquer avec le bouton droit de la souris sur `src` , puis sélectionner `Import` . Dans la boîte de dialogue, il faut choisir `General` puis `Archive File` , cliquer sur `Next` , puis sur `Browse` sur la première ligne et sélectionner le fichier `Bug.zip` téléchargé. Enfin, cliquer sur `Finish` .

L’objectif de cet exercice est de trouver l’erreur dans chacun de ces huit programmes en utilisant les différents messages d’erreur retournés par le compilateur ou à l’exécution.

Pour le futur, vous avez [ici](http://www.enseignement.polytechnique.fr/informatique/INF321/Deboguage.html) (<http://www.enseignement.polytechnique.fr/informatique/INF321/Deboguage.html>) un récapitulatif des erreurs fréquentes, également accessible directement depuis la page du cours.

Seuil de percolation

Nous allons calculer une valeur approchée du seuil de percolation d’un chemin de haut en bas dans une matrice contenant des cases noires et blanches. Pour ce faire, nous allons noircir aléatoirement des cases et, après chaque étape, nous allons détecter s’il existe un chemin fait de cases noires allant d’une case quelconque du haut (première ligne) à une case quelconque du bas (dernière ligne) de la matrice. Dans l’affirmative, la percolation sera atteinte. Le seuil de percolation est la proportion de cases noires qu’il est probabilistiquement nécessaire de noircir pour atteindre la percolation. Ce seuil est le même pour toute matrice pas trop petite.

Pour représenter une matrice carrée `M` de taille `N`, nous allons utiliser un tableau unidimensionnel `T` de taille `NxN` : la case à la ligne `i` et la colonne `j` dans `M` (avec `i` et `j` compris entre `0` et `N-1`) correspondra à l'élément d'indice `i.N + j` dans `T`. Par exemple, voici ci-dessous l'encodage d'une matrice de taille 3 par un tableau unidimensionnel de taille 9 (chaque case du tableau contient ici l'indice (ligne, colonne) de la case de la matrice qu'elle encode).

| | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| (0,0) | (0,1) | (0,2) | (1,0) | (1,1) | (1,2) | (2,0) | (2,1) | (2,2) |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|

Exercice 3 : définition de la classe

Dans un nouveau projet, définir une classe `Percolation` contenant les **constantes** `size`, dans un premier temps égal à 10, et correspondant à la taille d'une matrice et `length` de valeur `size * size`. Cette classe contiendra aussi un tableau statique de booléens `grid` de taille `length` correspondant à la matrice : une case contenant `true` (resp. `false`) sera noire (resp. blanche).

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 18, 2023 (16:12:21)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Percolation.java

Exercice 4 : fonction `init`

Définir une fonction `void init()` sans argument initialisant la matrice avec que des cases blanches. Cette fonction servira aussi à réinitialiser la matrice entre deux expériences de percolation.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 18, 2023 (16:16:00)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Percolation.java

Exercice 5 : fonction `print`

Définir une fonction `void print()` sans argument affichant la matrice sur la sortie standard. Les cases blanches seront représentées par le caractère `' - '` (tiret) et les cases noires par `' * '` (étoile). Faites bien attention à ce que les lignes correspondent bien à la première dimension de la matrice et les colonnes à sa deuxième.

Par exemple, le tableau

| | | | | | | | | |
|------|-------|-------|------|------|-------|-------|------|-------|
| true | false | false | true | true | false | false | true | false |
|------|-------|-------|------|------|-------|-------|------|-------|

(encodant une matrice 3x3) sera affiché comme suit :

```
* - -
** -
- * -
```

Pour tester vos deux fonctions, ajouter une fonction `main` initialisant la matrice avec que des cases blanches sauf la case à la ligne 2 et la colonne 4, puis l'affichant.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 18, 2023 (16:20:39)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Percolation.java

Exercice 6 : percolation

1. Définir une fonction `int randomShadow()` noircissant au hasard une des cases blanches de la matrice (en supposant qu'il en existe une) et retournant l'indice correspondant. Pour ce faire, vous pourrez utiliser la fonction `Math.random` qui retourne un nombre flottant entre `0.0` (inclus) et `1.0` (exclus).

Remarque algorithmique : On peut supposer que la tableau n'est pas trop rempli, contient au moins autour de 40% de cases blanches et que l'on va donc trouver une case blanche après quelques tirages au sort. C'est la solution la plus simple. Une autre solution qui garantit un temps de calcul faible même dans le pire cas est la suivante: commencer par déterminer un ordre aléatoires des cases, puis les noircir dans cet ordre. Si vous voulez faire cela, vous pouvez utiliser l'algorithme de mélange de Fisher-Yates (https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle) pour trouver une permutation aléatoire. L'expérience montre que cela n'apporte pas de gain pratique important.

2. Définir une fonction `boolean isNaivePercolation(int n)` retournant le booléen `true` si et seulement s'il existe un chemin noir dans la matrice, entre une de ses cases de la ligne `0` (le haut de la matrice) et une de ses cases de la ligne `size-1` (le bas de la matrice), et passant par `n`.

Pour cela, il est possible de procéder en deux étapes similaires:

- la détection d'un (demi-)chemin entre la case `n` et le haut de la matrice; et
- la détection d'un (demi-)chemin entre cette même case et le bas de la matrice.

Il faut néanmoins prendre garde à **ne pas visiter deux fois la même case** sur un même (demi-)chemin et à **factoriser le code** pour la détection des deux (demi-)chemins. Pour cela, vous pourrez définir puis utiliser une fonction récursive auxiliaire `boolean detectPath(boolean[] seen, int n, boolean up)` détectant un (demi-)chemin de la case d'indice `n` au haut (resp. bas) de la matrice si `up` est vrai (resp. faux), en considérant le tableau `seen` des cases déjà visitées pour ce (demi-)chemin.

3. Ajouter une fonction `boolean isPercolation(int n)` se contentant d'appeler `isNaivePercolation`. Elle sera utilisée pour tester facilement les différentes versions de `isPercolation` sans avoir à modifier tout votre programme.
4. Définir une fonction `double percolation()` noircissant des cases de la matrice au hasard, tant que sa percolation n'a pas eu lieu (le nom de la fonction est juste `percolation`, et `double` est le type de retour). Une fois la percolation atteinte, elle retournera la proportion de cases noircies.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (09:15:56)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Percolation.java

Exercice 7 : méthode de Monte-Carlo

1. Définir une fonction `double monteCarlo(int n)` calculant une estimation (entre `0` et `1`) du seuil de percolation en effectuant `n` simulations de percolation. Effectuer un tel calcul approché par simulations successives est appelé la *méthode de Monte-Carlo* (d'où le nom de la fonction). Elle est utilisée pour résoudre efficacement plein de problèmes pratiques pour lesquels un calcul exact est impossible ou trop coûteux.
2. Modifier la fonction `main` de façon à afficher le seuil de percolation d'une matrice blanche après `n` simulations, `n` étant fourni par l'utilisateur *via* l'onglet `Arguments` de la boîte de dialogue accessible à partir de `Run > Run Configurations`. Vous pouvez utiliser `Integer.parseInt` (<https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#parseInt-java.lang.String->) pour lire un entier dans une chaîne de caractères. À l'aide de la fonction `System.currentTimeMillis()`, afficher également le temps d'exécution nécessaire pour atteindre le seuil de percolation.
3. Essayer votre programme avec différentes tailles de matrice et différents nombre de simulations.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (09:27:33)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Percolation.java

Union-Find

La solution précédente n'est pas satisfaisante car le test de percolation est trop inefficace. Nous allons utiliser une structure de données appelée *Union-Find* pour optimiser notre recherche. Elle permet de calculer efficacement une relation d'équivalence *via* le représentant canonique de chacune des classes d'équivalence. Ici, deux indices de la matrice sont équivalents s'ils sont sur un même chemin noir, de sorte que détecter la percolation revient à détecter le fait qu'un élément du haut est équivalent à un élément du bas de la matrice.

Exercice 8 : définir Union-Find

Nous allons d'abord programmer une première version de la structure d'Union-Find, simple mais peu efficace (quick-find).

1. Dans un nouveau fichier `UnionFind.java`, créer une classe `UnionFind` contenant un tableau `equiv` et une fonction `void init(int len)` initialisant ce tableau avec une taille `len` et associant `i` à chaque indice `i` (initialement `i` est son propre représentant dans sa classe d'équivalence singleton).
2. Ajouter à cette nouvelle classe une fonction `int naiveFind(int x)` retournant le représentant canonique associé à `x`.
3. Ajouter à cette nouvelle classe une fonction `int naiveUnion(int x, int y)` réalisant l'union des classes d'équivalence de `x` et `y` : les membres de la classe d'équivalence de `x` auront pour nouveau représentant canonique celui de `y`. Ce nouveau représentant sera *in fine* retourné par la fonction. La taille d'un tableau `t` est disponible dans la variable `t.length`.
4. Ajouter les fonctions `int find(int x)` et `int union(int x, int y)` se contentant d'appeler respectivement `naiveFind` et `naiveUnion`. Elles seront utilisées pour tester facilement les différentes versions de `find` et `union` sans avoir à modifier tout votre programme.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (09:40:28)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: UnionFind.java

Exercice 9 : utiliser Union-Find

Nous allons à présent utiliser notre structure d’Union-Find pour optimiser notre programme détectant une percolation.

1. Modifier la fonction `init` de la classe `Percolation` pour qu’elle initialise aussi la structure d’UnionFind (via un appel à `UnionFind.init`).
2. Définir une fonction `void propagateUnion(int x)` unissant la classe d’équivalence de la case `x` (supposée noire) à tous ses voisins noirs : à la fin de l’exécution de cette fonction, tous les voisins noirs et `x` doivent appartenir à la même classe d’équivalence.
3. Modifier la fonction `randomShadow` pour appeler la fonction `propagateUnion` à l’endroit approprié.
4. Écrire la fonction `boolean isFastPercolation(int n)` avec un comportement similaire à `isNaivePercolation` mais plus efficace grâce à l’utilisation de la structure d’Union-Find.
5. Modifier la fonction `boolean isPercolation(int n)` pour qu’elle utilise désormais `isFastPercolation`.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (09:50:58)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

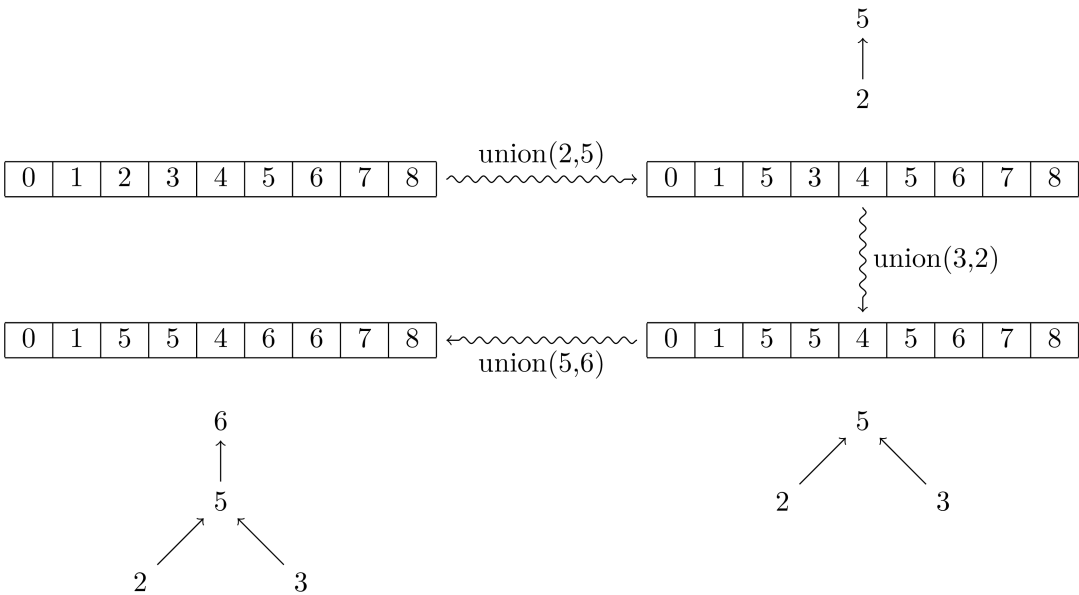
Expected files: Percolation.java, UnionFind.java

Optimisations

Exercice 10 : Union-Find paresseux

La solution précédente n’est pas efficace car l’opération d’union est trop coûteuse : sa complexité est linéaire en la taille de la matrice. Nous allons effectuer une première optimisation au prix d’une dégradation de la complexité de la fonction `find` (de temps constant à logarithmique). L’efficacité pratique sera déjà globalement bien meilleure.

L’optimisation consiste, lors d’une union de deux classes d’équivalence `C1` et `C2`, à ne pas modifier tous les membres d’une classe d’équivalence (disons `C1`), mais juste son représentant canonique. Ce dernier est alors associé à celui de `C2`, de sorte que des unions successives définissent une forêt dont chaque arbre ayant pour racine `r` représente une classe d’équivalence ayant pour représentant canonique `r`. Le tableau `UnionFind.equiv` contient ainsi non plus nécessairement les représentants canoniques des différentes classes d’équivalence mais plutôt les différents pères de ces arbres. Ce principe est illustré par le schéma suivant. Ce dernier montre le résultat de trois unions successives sur un tableau de taille 9. En complément du contenu du tableau `UnionFind.equiv`, les arbres non réduits à une racine sont également représentés.



Coder cette optimisation dans deux fonctions `fastFind` et `fastUnion`. Comparer ensuite l'efficacité de cette implémentation avec celle des autres versions en modifiant `find` et `union`.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (09:59:10)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: UnionFind.java

Exercice 11 : complexité logarithmique

1. Lors d'une union `union(x, y)`, une autre optimisation est possible. En effet, pour l'instant, nous avons arbitrairement choisi de modifier la classe d'équivalence de `x` pour qu'elle ait pour représentant canonique celui de la classe d'équivalence de `y`. Pour limiter les hauteurs des arbres, on peut choisir de conserver pour représentant canonique celui dont la hauteur de l'arbre correspondant à sa classe d'équivalence est la plus grande : en intégrant un arbre strictement plus petit à sa racine, sa hauteur ne changera pas. Cette optimisation permet de borner la hauteur de chaque arbre par le logarithme du nombre de ses éléments (voir poly). La complexité en temps de chaque étape est donc alors également logarithmique.

Coder cette modification dans une nouvelle fonction `logUnion` en ajoutant un tableau d'entiers `height` de même longueur que `equiv` à la classe `UnionFind`. L'élément d'indice `i` de ce tableau contiendra la hauteur de l'arbre ayant `i` pour racine. Ce tableau sera initialisé dans `UnionFind.init`.

2. Une autre optimisation, appelé *compression de chemin*, consiste à rapprocher de la racine les éléments rencontrés lors d'une recherche. De cette façon, les arbres seront moins hauts. On peut simplifier cette optimisation en se contentant de ne faire pointer chaque élément rencontré lors d'une recherche que vers son grand-père dans l'arbre (plutôt que vers son père comme avant, ou vers la racine pour la compression de chemin non simplifiée).

Coder la compression de chemin simplifiée dans une fonction `logFind` et comparer l'efficacité de cette implémentation avec celle des autres versions en modifiant `find` et `union`.

Quand on compresse les chemins, cela peut modifier la hauteur des arbres (c'est le but !). Calculer la hauteur exacte des arbres, pour l'utiliser dans la version optimisée de `union`, serait très coûteux. La stratégie habituelle est d'ignorer le problème, conserver la même fonction de calcul des hauteurs que sans compression (on parlera alors plutôt de *rang* que de hauteur, mais par simplicité on gardera le nom `height` dans le code), et il a été prouvé que la quantité ainsi calculée apporte les mêmes avantages que la vraie hauteur.

Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (10:00:00)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: UnionFind.java

Exercice 12 : optimisation du nombre de recherches

Maintenant que la fonction d'union est logarithmique, il devient avantageux de réduire le nombre de recherches, quitte à augmenter le nombre d'unions. Pour ce faire, nous allons introduire deux éléments distingués à la fin des tableaux d'Union-Find (dont les tailles seront donc égales à celle de la matrice + 2) pour représenter l'intégralité des bords haut et bas. Ainsi, il devient possible d'effectuer le test de percolation en comparant simplement les classes d'équivalence de ces deux éléments distingués avec un unique test, sans même tenir compte de la dernière case noircie.

On remarquera que la définition des classes d’équivalence est légèrement modifiée pour la première et la dernière ligne: une classe d’équivalence ne regroupe plus uniquement les cases sur un même chemin noir, mais contient toute la première (resp. dernière) ligne et l’élément distingué correspondant dès lors qu’une case de la première (resp. dernière) ligne est sur ce chemin.

Néanmoins, il devient nécessaire de modifier la fonction `propagateUnion` pour propager les classes d’équivalence à ces éléments spéciaux dès qu’une case du haut ou du bas est noircie.

Coder cette dernière optimisation dans une fonction `boolean isLogPercolation()` en modifiant également la fonction `propagateUnion` de manière appropriée.


Soumettez vos fichiers dans le formulaire ci-dessous.

Last submission: April 19, 2023 (10:11:24)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions 

Expected files: Percolation.java, UnionFind.java