Recherche de chemins sur une carte

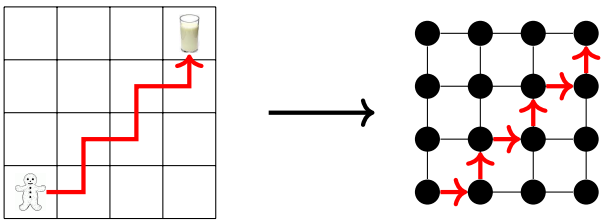
- Les noms et les codes permanents de tous les membres de l'équipe doivent apparaître en entête de chacun des fichiers.
- Pondération : 4% de la note finale (+1% pour le bonus).
- Remise via le lien de remise disponible sur la page Moodle, sous l'onglet «**Mini-projets**».
- Date limite pour remettre votre travail : voir sur la page Moodle.
- En cas de retard, −10% par jour, 0% après trois jours.
- Remettez uniquement les fichiers que vous avez modifiés :

  `Graphe.java` `GrapheParMatrice.java`
  `GrapheParListes.java` `Analyse.txt`
  `Carte.java` (seulement si vous faites le bonus)

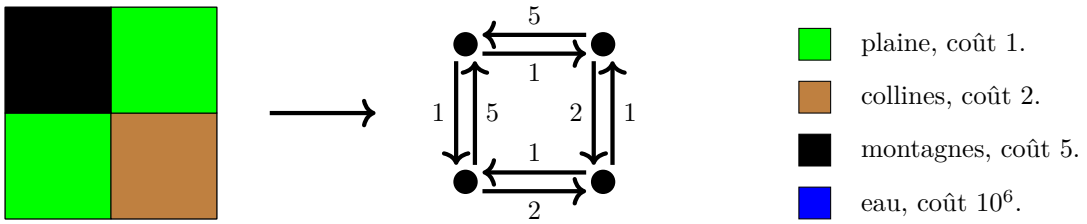
Consignes

1. Vous devez compléter les fichiers Java contenus dans l'archive **MAT210-MP2-Dijkstra.zip** disponible sur la page Moodle du cours, sous l'onglet «**Devoirs**».
2. Tout le code que vous écrivez doit être du **Java natif**. Il est strictement interdit d'ajouter des librairies (pas de nouvelles classes, pas de `import`).
3. Votre programme ne doit **jamais** interagir avec l'utilisateur.
4. Retirez ou commentez tout affichage superflu ajouté à votre code.

Contexte : vous travaillez au développement d'un jeu de type *tile-based*[1], c'est-à-dire un jeu se déroulant sur une carte formée de tuiles régulières formant une *grille*. Des personnages se déplacent sur la carte en passant d'une tuile à l'autre. Un déplacement sur une grille peut facilement être modélisé par un chemin dans un graphe.

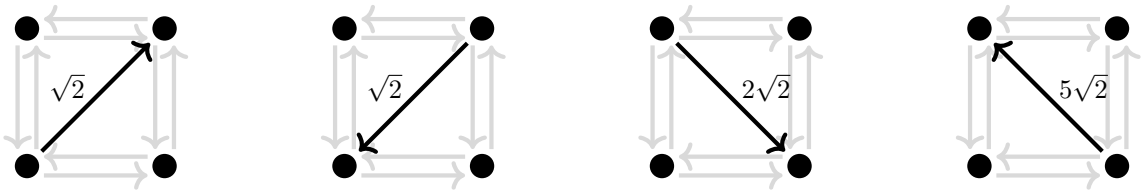


Le coût d'un déplacement peut être différent d'une case à l'autre selon le type de terrain. On utilise donc un graphe orienté et pondéré.



1. `https://en.wikipedia.org/wiki/Tile-based_video_game`

De plus, afin de produite un résultat plus réaliste, les mouvements diagonaux sont permis, mais le coût d'un tel déplacement est multiplié par $\sqrt{2}$. Ainsi au graphe précédent, il faut encoreajouter les arcs suivants :



La recherche d'un chemin optimal est donc équivalent à chercher un chemin de pondération minimum dans un graphe. Veuillez noter que **la conversion d'une carte en graphe a déjà été codée**, vous n'avez pas à le faire. Voir l'**Annexe C** pour un exemple détaillé de graphe obtenu à partir d'une carte.

Ce mini-projet vise 3 objectifs :

Objectif #1 : Implémenter les deux représentations de graphes : «**Listes**» et «**Matrice**».

Objectif #2 : Implémenter l'algorithme de Dijkstra.

Objectif #3 : Tester plusieurs variantes d'implémentations de l'algorithme de Dijkstra et comparer leurs performances sur différentes tailles de cartes.

Code fourni, description des classes :

- **Classe Carte**
  Une carte est une grille rectangulaire formée de cases carrées disposées en lignes et en colonnes. Une carte est toujours définie à partir d'une image au format `png`. Chaque pixel correspond à une case et la couleur du pixel détermine le type de terrain sur cette case. Il y a quatre types de terrains :

Terrain	Couleur	RGB	Coût de déplacement
Plaine	Vert	(0,255,0)	1
Collines	Brun	(165,42,42)	2
Montagnes	Noir	(0,0,0)	5
Eau	Bleu	(0,0,255)	10^6

  Bien que vous n'avez pas à utiliser cette classe directement, il est important de comprendre que les cases d'une carte sont identifiées de deux manières :

$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$
$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$
$(2,0)$	$(2,1)$	$(2,2)$	$(2,3)$

$\longleftrightarrow$

0	1	2	3
4	5	6	7
8	9	10	11

  — Par un couple de coordonnées (i, j). Il s'agit de la case située en ligne i, colonne j.
  — Par un entier k. Il s'agit de la k-ième case en comptant ligne par ligne depuis le coin en haut à gauche.

  Lorsqu'une carte est convertie en graphe, les sommets du graphe correspondent à l'identification des cases par des entiers. Si pour des fins de débogage vous voulez effectuer cette conversion, utilisez les fonctions `coordonneeToInt` et `intToCoordonnee` de la classe `Carte`.

- **Classe Graphe**
  Classe abstraite représentant un graphe orienté pondéré. Les sommets d'un graphe sont **toujours** les entiers de 0 à $n-1$ où n est le nombre de sommets. Les arcs d'un graphe sont représentés par la **sous-classe Graphe.Arc**. Un **Arc** contient trois informations :
  — `depart`, le sommet initial de l'arc,
  — `arrivee`, le sommet terminal de l'arc,
  — `ponderation`, un `double` indiquant le coût de l'arc.

- **Classe GrapheParListes**
  Hérite de la classe **Graphe** et implémente un graphe représenté par listes d'adjacence.

- **Classe GrapheParMatrice**
  Hérite de la classe **Graphe** et implémente un graphe représenté par une matrice d'adjacence.

- **Classe TesterGraphes**
  Fournit une fonction `main` qui teste une implémentation de graphes. Des graphes sont générés aléatoirement puis les fonctionnalités de la classe **Graphe** sont testées. Les fonctions testées sont : `getNbSommets, getPonderationArcAbsent, ajouterArc, getArcs, getPonderation`.

- **Classe TracerChemin**
  Fournit une fonction `main` qui trace un chemin de coût minimum sur une carte. Le résultat sauvegardé dans une image au format `png`. Le but est de visualiser le chemin calculé.

- **Classe Chronometre**
  Implémente un chronomètre utilisé pour mesurer les temps d'exécution.

(/0) **Exercice 0 : La classe Graphe**

Avant de commencer, il est primordial de **lire en détail** et de **comprendre** tout le code de la classe abstraite `Graphe`.

(/15) **Exercice 1 : Graphes représentés par une matrice d'adjacence**

Dans le fichier **GrapheParMatrice.java** complétez l'implémentation de la classe `GrapheParMatrice`. Vous devez écrire le code de quatre fonctions :
— le constructeur,
— la fonction `AjouteArc`,
— la fonction `getPonderation`,
— la fonction `getArcs`.

Remarque : le constructeur doit initialiser la matrice `this.m`. Les trois autres fonctions sont héritées de la classe `Graphe` ; votre implémentation doit être conforme la description faite dans la classe `Graphe`.

Aide : Pour la fonction `getArcs` il faut retourne un itérateur. Vous devez commencer par construire un **ArrayList** contenant tous les arcs demandés et ensuite retourner un itérateur sur ce dernier avec la fonction `.iterator()`.

Contrainte : vous ne devez pas ajouter de données membres à la classe. La matrice `m` suffit.

Testez votre implémentation à l'aide du programme `TesterGraphes`, en faisant :

```
$ java TesterGraphes --graphe Matrice
```

(/15) **Exercice 2 : Graphes représentés pas des listes d'adjacences.**

Dans le fichier **GrapheParListes.java** complétez l'implémentation de la classe `GrapheParListes`. Vous devez écrire le code de quatre fonctions :

- le constructeur,
- la fonction `AjouteArc`,
- la fonction `getPonderation`,
- la fonction `trouverArc`.
- la fonction `getArcs`.

Remarque : le constructeur doit initialiser le tableau `this.listes`. Il s'agit d'un `ArrayList` dont chacune des cases est de type `ArrayList<Arc>`. Le tableau `this.listes` doit avoir une case pour chaque sommet ; la i^{eme} case contient la liste des arcs dont le sommet initial est le sommet i. À l'initialisation, toutes les cases du tableau `this.listes` doivent être un `ArrayList<Arc>` initialisé et vide.

Aide : Pour la fonction `getArcs`, ne perdez pas de temps à recopier quoi que ce soit. Vous avez déjà un tableau contenant tous les arcs nécessaires, retournez simplement un itérateur sur ce tableau.

Contrainte : vous ne devez pas ajouter de données membres à la classe. Le tableau `listes` suffit.

Testez votre implémentation à l'aide du programme `TesterGraphes`, en faisant :

```
$ java TesterGraphes --graphe Listes
```

(/10) **Exercice 3 : Construction du chemin minimum**

Dans le fichier **Graphe.java** implémentez la fonction `tableauPredecesseurVersChemin` qui calcule le chemin de coût minimum à partir du tableau `predecesseur` calculé par l'algorithme de Dijkstra. Ce chemin doit débuter au sommet utilisé comme point de départ de l'algorithme de Dijkstra et terminer au sommet `destination`.

Par exemple, pour le tableau tableau `predecesseur` suivant :

0	1	2	3	4	5	6	7
−1	0	1	1	2	2	5	4

L'appel `tableauPredecesseurVersChemin(predecesseur, 7)` retourne le tableau :

0	1	2	3	4
0	1	2	4	7

Ce qui correspond au chemin $0 - 1 - 2 - 4 - 7$.

(/33) **Exercice 4 : Dijkstra par ArrayList**

Dans le fichier **Graphe.java** complétez la fonction `DijkstraParArrayList`. Vous perdrez des points si vous ne respectez pas les consignes suivantes :

- Vous devez implémenter l'algorithme de Dijkstra tel que présenté en classe.
- L'ensemble S doit être représenté par un `ArrayList<Integer>`.
  - Pour ajouter le sommet x à S faites `S.add(x)`
  - Pour tester si le sommet x est dans l'ensemble S, faites `S.contains(x)`.
- La recherche du sommet d'étiquette minimale parmi les sommets qui ne sont pas dans S doit obligatoirement être effectuée dans une fonction distincte. Ne codez pas tout d'un bloc.
- Contrairement à l'algorithme présenté dans les notes de cours, la fonction `DijkstraParArrayList` doit construire un tableau `predecesseur` dont l'utilisation sera présentée en détail lors des cours [2].

Testez votre code ! Par exemple :

2. L'utilisation de ce tableau est également décrite dans la page Wikipedia de l'algorithme de Dijkstra.

```
java mat210.TracerChemin --carte ./cartes/40x25.png --graphe Listes --algo DijkstraParArrayList --depart 3 5 --arrivee 24 39
########################################
# Carte           : ./cartes/40x25.png
# Nombre de cases : 1000
# Représentation  : Listes
# Algorithme      : DijkstraParArrayList
# Départ          : (3, 5)
# Arrivée         : (24, 39)
# Image en sortie : out.png
########################################
# Initialisation du graphe... terminée en 5.88 millisecondes.
# Calcul du chemin minimum de (3, 5) à (24, 39)... terminé en 309.73 millisecondes.
# Cout total du chemin : 47.69848480983498
# Écriture du résultat dans le fichier ``out.png``... terminée en 3.95 millisecondes.
########################################
```

Pour des fins de débogage, voici quelques exemples de coût minimum que vous devriez obtenir :

Carte	Départ	Arrivée	Coût
3x2.png	(0, 0)	(1, 2)	3.828
4x4.png	(0, 3)	(3, 0)	12.727
5x5.png	(4, 0)	(0, 4)	6.828
40x25.png	(10, 30)	(24, 0)	36.627
56x36.png	(0, 0)	(35, 55)	77.941
80x52.png	(0, 0)	(51, 79)	140.225
80x52.png	(0, 79)	(51, 0)	108.911
128x80.png	(79, 0)	(0, 127)	196.722
256x256.png	(0, 0)	(255, 255)	598.724

Remarque : selon le cas, le temps de calcul peut varier d'une fraction de seconde à plusieurs heures.

(/6) **Exercice 5 : Dijkstra par TreeSet**

Dans le fichier **Graphe.java** implémentez la fonction `DijkstraParTreeSet`. Cette fonction est identique à celle codée à l'**Exercice 4** à la différence que l'ensemble S doit être du type `TreeSet<Integer>`.

```
java mat210.TracerChemin --carte ./cartes/40x25.png --graphe Listes --algo DijkstraParTreeSet
```

Vous devriez obtenir les mêmes chemins et les mêmes coûts qu'avez la version précédente !

Aide : cet exercice est extrêmement simple, il n'y a pratiquement rien à faire. Commencez par faire un copier/coller du code de `DijkstraParArrayList` puis apportez-y les modifications nécessaires. Si vous modifiez plus de trois lignes, c'est probablement que vous êtes sur une fausse piste !

(/6) **Exercice 6 : Dijkstra par BooleanArray**

Dans le fichier **Graphe.java** implémentez la fonction `DijkstraParBooleanArray`. Cette fonction est identique à celles codées aux **Exercices 4 et 5** à la différence que l'ensemble S doit être du type `boolean[]`. Étant donné un sommet x, $S[x]$ est `vrai` si x est dans S, `faux` sinon.

```
java mat210.TracerChemin --carte ./cartes/40x25.png --graphe Listes --algo DijkstraParBooleanArray
```

(/15) **Exercice 7 : faire le bon choix**

La réponse à cette dernière question doit être rédigée dans un fichier texte nommé `Analyse.txt`.

Dans le contexte du développement d'un jeu *tile-based*, on vous demande de fournir vos conclusions quant à la représentation du graphe (Listes ou Matrices) et l'algorithme de recherche de chemins à utiliser.

Commencer par donner,**en notation grand-O**, la complexité asymptotique des trois variantes de l'algorithme de Dijkstra que vous avez codées. Pour effectuer cette analyse, tenez compte des points suivants :

— Voici un extrait de la documentation de la classe `ArrayList` :
« *The* `size`, `isEmpty`, `get`, `set`, `iterator`, *and* `listIterator` *operations run in constant time. The* `add` *operation runs in amortized constant time [...]. All of the other operations run in linear time (roughly speaking).*»
Vous pouvez donc considérer que $S.add()$ s'effectue en temps constant, $\mathcal{O}(1)$ mais que `S.contains()` est en $\mathcal{O}(n)$.

— Voici un extrait de la documentation de la classe `TreeSet` :
« *This implementation provides guaranteed log(n) time cost for the basic operations (*`add`, `remove` *and* `contains`*).*»

— Consulter ou modifier une case d'un tableau de booléens se fait en temps constant $\mathcal{O}(1)$.

Est-ce que votre analyse est confirmée par la pratique ? Tester vos implémentations avec différentes tailles de cartes et rédigez vos conclusions.

Vous en voulez encore ? Section bonus à l'annexe E.

Annexe A – Compilation et exécution

En ligne de commande

> Les commandes sont fournies pour un terminal *à la Unix* (Linux, FreeBSD, MacOS, etc.), sous Windows, tout fonctionne pas mal de la même manière à la différence que le prompt est habituellement > plutôt que $. **Remarque, sous Windows** vous devrez modifier la variable `PATH`. Commencez par identifier le dossier où est situé le programme `javac`, par exemple `C:\Program Files\Java\jdk1.8.0_152\bin`, puis entrez la commande :
> `set PATH=%PATH%;"C:\Program Files\Java\jdk1.8.0_152\bin"`

—

— Extraire les fichiers de l'archive **MAT210-MP2-Dijkstra.zip**, et se positionner dans le dossier `MAT210-MP2-Dijkstra`.

— On compile avec la commande :
```
$ javac mat210/*.java
```

— Pour afficher l'aide du programme de test :
```
$ java mat210.TesterGraphes --help
```

— Pour tester l'implémentation par matrice d'adjacence :
```
$ java mat210.TesterGraphes --graphe Matrice
```

— Pour tester l'implémentation par listes d'adjacence :
```
$ java mat210.TesterGraphes --graphe Listes
```

Compilation et exécution avec Eclipse

— Dans **Eclipse**, créer un nouveau projet Java en utilisant toutes les options par défaut. Si Eclipse, vous propose de créer un **module-info.java**, pour pouvez lui dire de ne pas le faire.
— Dans le **Package Explorer**, faire un clic droit sur le projet que vous venez de créer, choisir **New** > **Package** et créer un package nommé `mat210` (en minuscules).
— Extraire les fichiers java de l'archive **MAT210-MP2-Dijkstra.zip** et les ajouter au package `mat210` (par exemple en faisant un *drag-and-drop*).
— Double cliquer sur le fichier à exécuter (**TesterGraphes.java** ou **TracerChemin.java**) et cliquer sur le bouton **run**.

> **Rappel** : référez-vous à l'annexe B de l'énoncé du Mini-projet 1 pour les problèmes suivants :
> — Encodage UTF-8 des fichiers sources.
> — La gestion des arguments de la ligne de commande.

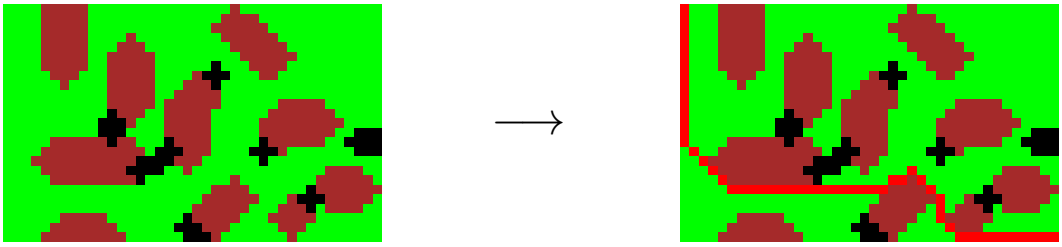
Annexe B – Modes d'affichage

Lorsque vous exécutez le programme `TracerChemin`, vous pouvez choisir l'un des deux modes d'affichages suivants :

— Le mode **pixel** (par défaut).

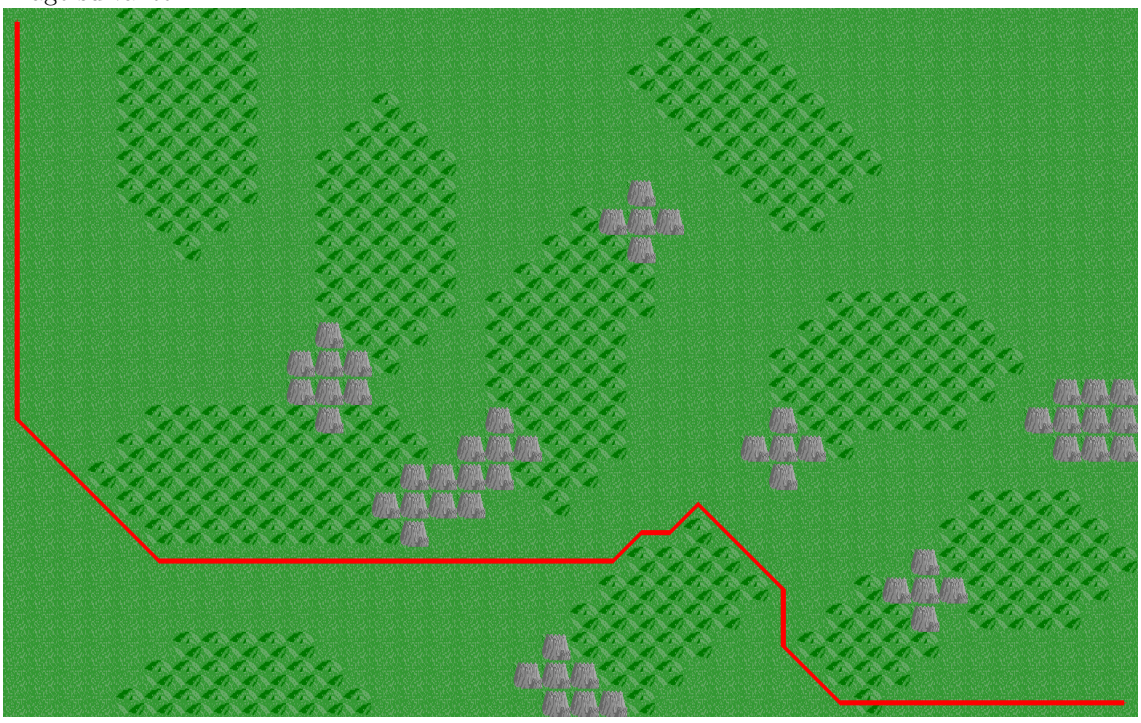
La carte est dessinée de sorte que chaque case est représentée par un pixel. On obtient ainsi une image similaire à celle utilisée pour définir la carte.

```
java mat210.TracerChemin --carte 40x25_bis.png --algo DijkstraParBooleanArray --graphe Listes --affichage pixel
```



— Le mode **graphique** (par défaut).

Chaque type de terrain est représenté par une petite image appelée *tuile*. L'image produite est significativement plus grande que l'image initiale. Si on reprend la commande de l'exemple précédent, mais qu'on y remplace l'argument "`--affichage pixel`" par "`--affichage graphique`" on obtient l'image suivante :

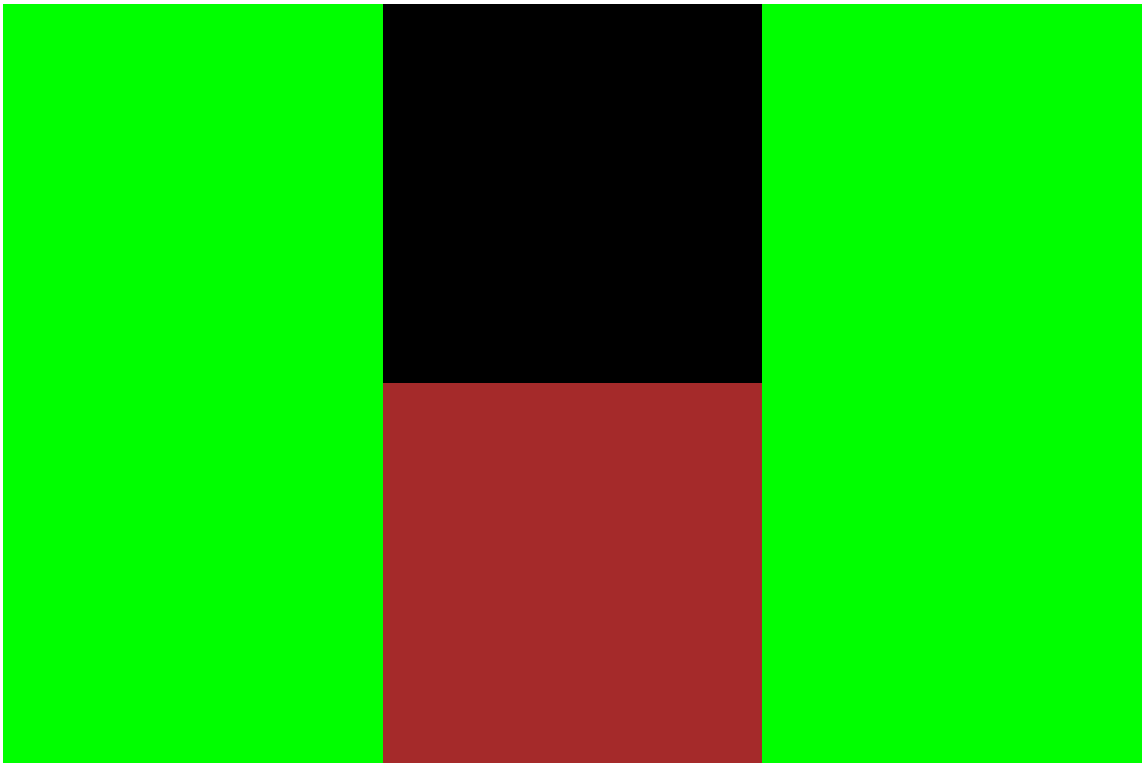


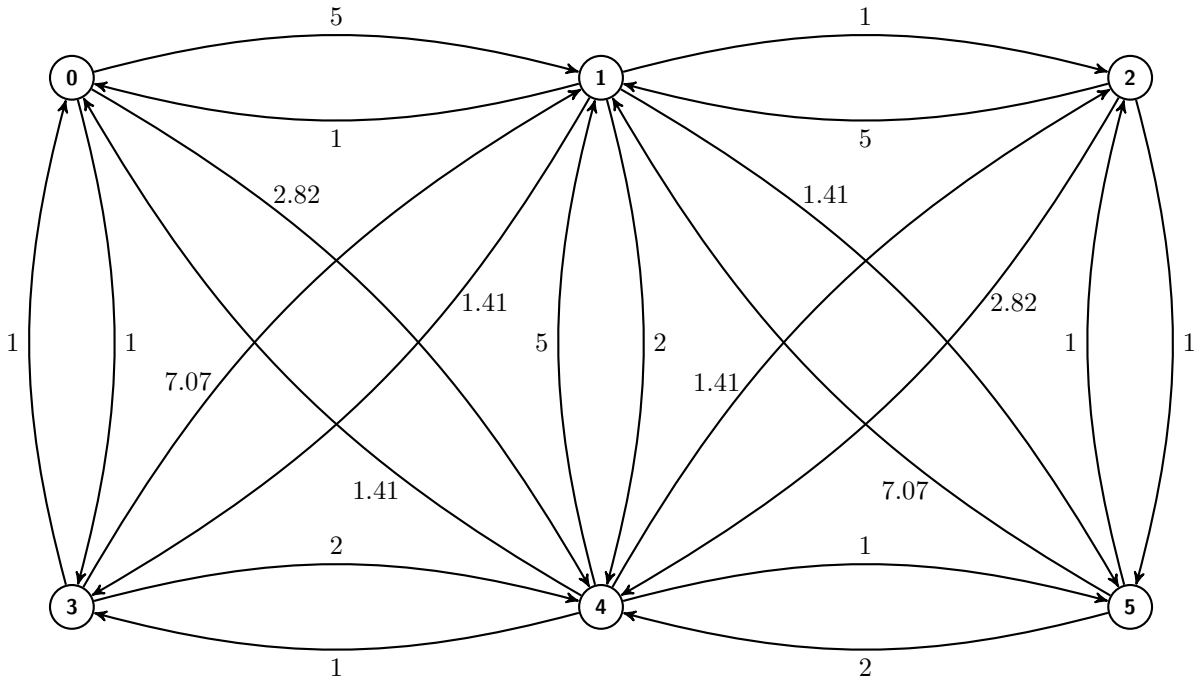
Si vous obtenez l'erreur :

`ERREUR incapable de lire les tuiles pour l'affichage graphique.`

C'est probablement parce que le programme ne regarde pas au bon endroit pour trouver les images des tuiles. Dans le fichier `Carte.java`, modifiez la variable `cheminVersImages`.

Annexe C – Exemple détaille du graphe d'une carte





Annexe D – Fabriquer vos propres cartes

Vous pouvez créer de nouvelles cartes. Il suffit de créer une image au format `png` donc chaque pixel correspond **exactement** l'une des quatre couleurs supportées par la classe `Carte` (voir le tableau en page 2).

Annexe E – Bonus : l'algorithme A*

L'algorithme A* est une variante de l'algorithme de Dijkstra optimisée pour le cas où on est en mesure de calculer efficacement une *heuristique* qui **sous-estime** le coût d'un chemin minimum entre deux sommets.

Heuristique de sous-estimation

Dans le fichier `Carte.java`, complétez la fonction `sousEstimation` de la sous-classe `Heuristique`.

Votre fonction doit impérativement satisfaire les deux critères suivants :
- Complexité en $\mathcal{O}(1)$.
- La valeur retournée doit être **inférieure ou égale** au coût d'un chemin minimum.

Plus la valeur retournée est grande, plus l'algorithme A* sera efficace. Par contre, si la valeur retournée est supérieure au coût d'un chemin minimum, alors plus rien ne fonctionne et vous produirez possiblement un résultat faux !

En particulier, une heuristique qui retourne toujours zéro fait en sorte que l'algorithme $A^\star$ effectue exactement le même travail que l'algorithme de Dijkstra.

La note accordée pour ce bonus dépendra, entre autre, de la qualité de votre heuristique. **Commentez en détail** le calcul effectué. Vos commentaires doivent expliquer pourquoi le calcul effectué ne peut jamais produire une surestimation.

L'algorithme A*

On apporte les modifications suivantes à l'algorithme de Dijkstra.

- **Données supplémentaires**. On se dote d'un tableau supplémentaire qui à chaque sommet associe une estimation du coût d'un chemin minimum du sommet de départ jusqu'au sommet d'arrivée et qui passe par ce sommet.

  Rappel : dans l'algorithme de Dijkstra, $L[u]$ est le coût du chemin minimum actuellement connu du point de départ jusqu'à u.

  L'estimation du coût d'un chemin minimum du sommet de départ jusqu'au sommet d'arrivée et qui passe par u est donc $L[u]$ plus la valeur de l'heuristique entre u et le sommet d'arrivée.

- **Choix du sommet à visiter**. Dans l'algorithme de Dijkstra, la première étape de la boucle principale consiste à sélectionner le **sommet non visité dont l'étiquette est minimale**. Dans l'algorithme A*, on sélectionne plutôt le **sommet non visité dont l'estimation est minimale**.

Testez l'algorithme A* avec différentes tailles de cartes. Dans le fichier `Analyse.txt`, répondez aux questions suivantes :

Q1. Au niveau de l'analyse grand-O, est-ce que l'algorithme A* est plus efficace que l'algorithme de Dijkstra ?

Q2. En pratique, comment est-ce que les performances de l'algorithme A* se comparent à celle de l'algorithme de Dijkstra ?

Q3. Dans le contexte d'un jeu de type *tiled-based* où des centaines de personnages se déplacent sur une carte de grande taille, est-ce que l'utilisation de l'algorihtme A* est une bonne idée ? Si oui, expliquez pourquoi. Sinon, expliquez pourquoi et suggérez une meilleure approche.