ITI 1521. Introduction à l'informatique II
Hiver 2019

Devoir 3
(Version du 1er juin 2019)

Échéance: 24 mars 2019, 23 h 30

Objectifs d'apprentissage

- **Concevoir** une application utilisant la programmation événementielle
- **Découvrir** le patron de conception Modèle-Vue-Contrôleur (MVC)
- **Expérimenter** avec l'évolution d'une application logiciel

Introduction

Pour ce devoir, nous allons concevoir l'interface utilisateur graphique pour le jeu **Lights Out** et ainsi réutiliser une partie du code du devoir 2. Vous pouvez utiliser votre propre implémentation ou, si vous le souhaitez, utilisez la solution proposée (ou encore un mélange des deux).

- Vidéo YouTube — Présentation du troisième devoir: IUG Lights Out

1 Solutionner le jeu [30 points]

La solution développée pour le devoir 2 devait trouver une solution à partir d'un tableau où toutes les cellules étaient éteintes («*off*»). Il nous faut donc d'abord généraliser ce travail pour partir d'une solution partielle arbitraire.

1.1 GameModel

La classe **GameModel** représente l'état actuel du tableau de jeur. Ses caractéristiques sont les suivantes :

- **GameModel(int width, int height)** est un constructeur. Initialise l'objet afin de représenter un tableau où toutes les cellules sont éteintes. Les paramètres **width** et **height** sont les dimensions du tableau de jeu.
- **int getHeight()** : méthode d'accès en lecture («*getter*») pour la variable d'instance **height**.
- **int getWidth()** : méthode d'accès en lecture («*getter*») pour la variable d'instance **width**.
- **boolean isON(int i, int j)** : retourne **true** si l'emplacement de la **rangée (row) i** et la **colonne (column) j** est allumée («*on*»), **false** sinon.
- **reset()** : éteint (mettre à «*off*») toutes les cases du modèle.
- **set(int i, int j, boolean value)** : change la "valeur" de l'emplacement (i,j) du modèle (soyez prudent, la **colonne** est **i** et la **rangée** est **j**).
- **String toString()** : retourne une chaîne de caractère qui représente le modèle.

1.2 Solution

Nous devons adapter la classe **Solution** à cette nouvelle situation. **Notez :** ici, nous ajoutons de nouveaux éléments à notre application et ainsi ces modifications ne changent pas les méthodes existantes et n'affectent pas leur comportement.

Voici les nouvelles méthodes de la classe **Solution** :

- **boolean stillPossible(boolean nextValue, GameModel model)** : cette méthode retourne **false** si on ne peut pas compléter cette solution et ainsi obtenir une solution qui fonctionne étant donné l'état spécifié par l'instance de **GameModel** passé en paramètre (**model**) et la prochaine valeur, **nextValue**.

  Notez que la méthode retourne **false** si la solution courante auquel on ajoute **nextValue** ne donnera jamais une solution qui fonctionne, et **true** sinon. Retourner la valeur **true ne signifie pas que la solution sera nécessairement une solution qui fonctionne**. On ne sait tout simplement pas si ce sera possible ou non (on n'a pas encore prouvé que c'est impossible).

  Notez aussi que cette méthode ne doit pas modifier l'état de la solution (l'instance de la classe **Solution**) duquel on fait l'appel à cette méthode. Nous n'ajoutons pas la valeur de **nextValue**, on indique simplement si une telle extension aurait pour effet d'annihiler les chances d'aboutir à une solution fonctionnelle.

- **public boolean finish(GameModel model)** : cette méthode suppose que la solution est encore réalisable pour le tableau de jeu représenté par l'instance de la classe **GameModel** (paramètre **model**), mais d'une seule façon. Cette méthode étend la solution d'une seule façon à chaque étape, jusqu'à ce que la solution soit complète et fonctionnelle pour le tableau représenté par l'objet **GameModel**, où que l'on ait démontré qu'elle n'est pas réalisable. Elle retourne **true** si et seulement si la solution est complète et fonctionne.

  La méthode change l'état de l'instance de la classe **Solution** de laquelle on appelle la méthode. Si elle retourne **true**, cette solution est donc complète et fonctionne pour le tableau représenté par le paramètre **model** (objet de la classe **GameModel**).

- **public boolean isSuccessful(GameModel model)** : cette méthode retourne **true** si et seulement si la solution est complète et fonctionne pour le tableau de jeu représenté par l'instance de la classe **GameModel** (paramètre **model**). (ajout du 11 mars 2019)

- **int getSize()** : retourne la taille de la solution, c'est à dire le nombre de positions qui doivent être sélectionnées. Sur un tableau de taille 3×2 dont les cellules sont éteintes, la solution qui consiste à sélectionner les positions (1,1) et (3,2) est de taille 2, alors que la solution qui consiste à sélectionner les positions (1,1), (2,1), (1,2) et (2,2) est de taille 4.

Lorsque vous implémentez ces nouvelles méthodes, vous devez éviter la duplication de code dans votre classe, vous devrez possiblement remanier un peu le code existant.

1.3 LightsOut

Voici maintenant les méthodes que vous devez ajouter à la classe **LightsOut**.

- **ArrayList<Solution> solve(GameModel model)** : La méthode de classe **solve** trouve toutes les solutions du jeu **Lights Out** étant donné un tableau de jeu spécifié par le paramètre **model** (qui désigne un objet de la classe **GameModel**). Cette méthode utilise l'algorithme de **parcours en largeur** et retourne une liste (objet de la classe **ArrayList**) contenant toutes les solutions valides.

- **Solution solveShortest(GameModel model)** : La méthode de classe de classe **solveShortest** retourne la référence d'une solution de taille minimum pour le jeu **Lights Out** étant donné le modèle spécifié (objet de la classe **GameModel** désigné par le paramètre **model**). Notez qu'il pourrait y avoir plus d'une solution de taille minimale. Auquel cas, la méthode retourne l'une d'elles.

Ici aussi, on ne change pas les méthodes de visibilité **public** ou leur comportement. Tentez aussi d'éviter la duplication de code.

1.4 L'interface Queue et son implémentation

Pour le devoir 2, nous utilisions l'interface **SolutionQueue**, une file spécifiquement conçue pour des instances de la classe **Solution**. Notre implémentation de l'interface utilisait la classe **ArrayList** pour sauvegarder ces éléments.

Nous allons maintenant remplacer l'interface et son implémentation par notre propre code. Vous devez fournir l'interface **Queue** ayant un paramètre de type et son implémentation, utilisant aussi le concept de type générique. Modifiez l'application afin qu'elle utilise cette nouvelle interface et son implémentation.

Notez que la méthode **solve** de la classe **LightsOut** retourne toujours une instance de «java.util.ArrayList<Solution>».

1.5 Validation

Nous vous fournissions la classe **TestQ1** afin de valider votre travail et vous assurer que votre implémentation est conforme aux attentes. Vous trouverez un exemple d'exécution à la section A du document.

2 Interface utilisateur graphique [70 points]

Nous allons maintenant créer l'interface utilisateur graphique (IUG) («*Graphical User Interface*» GUI). Notre jeu aura plusieurs fonctionnalités. Nous vous suggérons de procéder par étapes. Dans un premier temps, développez une version fonctionnelle du jeu. Un prototype qui affiche correctement l'information. Lorsque vous aurez réalisé cette version, implémentez la fonctionnalité qui permet de remettre à zéro le jeu. Ensuite, ajoutez la fonctionnalité qui permet de générer un jeu aléatoirement. Finalement, ajoutez la possibilité de superposer une solution sur le jeu.

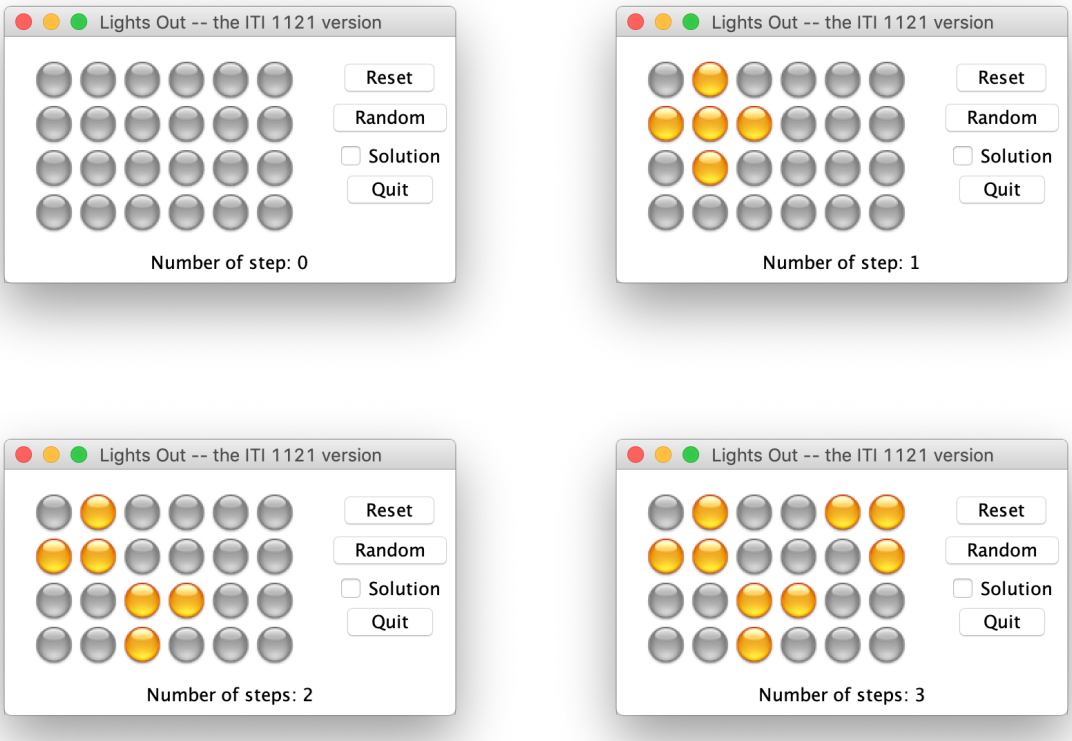

FIGURE 1 – L'interface utilisateur graphique pour le jeu **Lights Out** sur un tableau de 6×4.

Modèle-vue-contrôleur

Le patron de conception («*design pattern*») **modèle-vue-contrôleur** (MVC — «*Model-View-Controller*») est utilisé fréquemment pour la conception d'interfaces utilisateur graphiques. Vous trouverez facilement des informations complémentaires sur Internet (notamment ici : Wikipedia, Apple, Microsoft, et Oracle, pour ne nommer que ceux-ci). L'idée générale est de séparer les rôles de vos classes en trois catégories :

- Le **modèle** : ces objets représentent l'**état** de l'application.
- La **vue** (ou les vues[1]) : ces objets présentent le modèle à l'utilisateur (la portion visuelle de l'interface utilisateur graphique). Sa représentation reflète l'état courant du modèle. Il peut y avoir plusieurs vues simultanément. Cependant, pour ce devoir nous n'en aurons qu'une seule.
- Le **contrôleur** : ces objets implémentent la logique de l'application, comment l'état se transforme au cours de l'exécution en fonction des interactions externes (typiquement, les interactions avec l'utilisateur).

L'un de grands avantages du patron de conception MVC est la séparation claire entre les différentes activités de l'application : le modèle ne représente que l'état courant de l'application, sans préoccupation pour son affichage ou encore les interactions avec l'utilisateur. La vue quant à elle n'est responsable que de la représentation (visuelle ou autre) de l'état du modèle. Notamment, la vue reçoit les interactions de l'usager, mais ne les gère pas. Elle passe les requêtes au contrôleur. Finalement, le contrôleur est le «cerveau» de l'application. Il ne se préoccupe pas de représenter l'état de l'application ou sa représentation.

En plus de cette séparation claire, MVC fournit un schéma de collaboration logique entre les composantes de l'application (Figure 2).

1. Lors d'une interaction avec la vue (pour ce devoir, lorsque l'utilisateur clique sur une tuile), le contrôleur en est informé (message 1 de la Figure 2).
2. Le contrôleur traite l'information et met à jour le modèle de façon appropriée (message 2 de la Figure 2).
3. Une fois l'information traitée et le modèle mis à jour, le contrôleur informe la vue qu'elle doit être rafraîchie (message 3 de la Figure 2).
4. Finalement, chaque vue doit consulter le modèle afin de refléter fidèlement l'état courant (message 4 de la Figure 2).

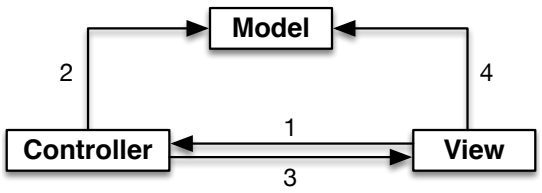

FIGURE 2 – La collaboration entre le **modèle**, la **vue**, et le **contrôleur**

- Vidéo YouTube — Le patron de conception «modèle-vue-contrôleur»

Le modèle

La première étape consiste à concevoir le modèle. À la section précédente, nous avons conçu notre point de départ, il suffit donc de compléter cette implémentation.

Dans un premier temps, nous ajoutons une nouvelle méthode à la classe **Solution** : une méthode afin de déterminer si une position du tableau est sélectionnée ou non. Ajoutez donc cette méthode à la classe **Solution** :

- **boolean get(int i, int j)** : retourne **true** si la position à la colonne **i** et la rangée **j** est sélectionnée dans cette solution et **false** sinon.

[1] C'est l'une des grandes forces de ce patron de conception, on peut facilement associer plusieurs vues à l'application, par exemple, on peut associer une vue pour les usagers ayant un handicap visuel.

Nous passons maintenant à la classe **GameModel** et nous lui ajoutons les méthodes suivantes :

- **click(int i, int j)** : mets à jour le modèle pour indiquer la sélection de la position à la rangée **i** et la colonne **j**. Ainsi, l'état de cet élément et de ses voisins change en fonction de la logique du jeu.
- **int getNumberOfSteps()** : retourne le nombre d'appels à la méthode **click** depuis la dernière remise à zéro de l'application (ou depuis le début de la partie).
- **boolean isFinished()** : retourne **true** si ce modèle représente une solution complète du jeu.
- **randomize()** : relance le jeu avec solution aléatoire réalisable du jeu plutôt qu'un tableau où toutes les cellules sont éteintes.
- **void setSolution()** : force le modèle à trouver une solution de taille minimale pour ce modèle.
- **boolean solutionSelects(int i, int j)** : retourne **true** si le modèle a une solution courante et si dans cette solution la rangée **i** et la colonne **j** est sélectionnée.

Notez que c'est la responsabilité du modèle de s'assurer que la solution calculée correspond à l'état courant du modèle. Si le modèle a une solution périmée, où si on n'avait jamais fait le calcul d'une solution, la valeur retournée par **solutionSelects** n'est pas fiable.

Vous trouverez une une description détaillée des classes dans la documentation.

Le contrôleur

Note : comme stratégie pour réaliser votre implémentation, nous vous suggérons de créer une vue textuelle temporaire qui affiche simplement l'état du modèle (via la méthode **toString** du modèle) et qui demande à l'usager de sélectionner la prochaine position (en lui demandant d'entrer la colonne et la ligne). Cette vue textuelle permettra la validation du modèle et du contrôleur indépendamment de l'IUG.

Le contrôleur est implémenté par la classe **GameController**. Son constructeur reçoit la largeur et la hauteur du tableau de jeu. Ce dernier doit aussi créer l'instance du modèle et de la vue. C'est aussi le gestionnaire («*listener*») pour toutes les composantes de l'IUG. Vous trouverez une une description détaillée de la classe dans la documentation.

La vue

Nous construisons finalement la vue de cette application. Notamment, vous devez créer une sous-classe de JFrame que vous nommerez **GameView**. Au bas de cette fenêtre, il y a le nombre d'étapes de jeu. À droite, il y a trois boutons, un bouton pour la remise à zéro, un bouton pour générer un jeu aléatoire, et un bouton afin de quitter l'application. Une case à cocher est également disponible afin de superposer la solution au tableau actuel.

Le tableau lui-même comprend une série de tuiles, h lignes et w colonnes, où h est la hauteur du tableau et w sa largeur). La Figure 3 montre le résulta attendu.

Pour implémenter chaque tuile, nous utilisons la classe **GridButton**, une sous-classe de JButton. L'implémentation de la classe **GridButton** est inspirée de la classe **Cell**[2] du jeu Puzzler d'Apple. Nous avons avons présenté le code source de cette application lors du laboratoire 7. La classe **GridButton** comprend la méthode **getImageIcon()** qui utilise la valeur courante de la variable d'instance **icon** afin de trouver l'icône appropriée (référence d'un objet **ImageIcon**). Vous devez vous assurer que ces images sont placées dans le répertoire «*Icons*». Nous vous fournissons 4 icônes correspondant aux états possibles pour une tuile (sélectionnée ou non en mode normal, sélectionnée ou non lors qu'une solution est affichée).

Le jeu

Lorsque l'utilisateur démarre le jeu, un tableau complètement éteint et de la taille spécifiée est affiché. Tout au long de la partie, le nombre de clics est indiqué en bas de la fenêtre. Lorsque l'utilisateur complète le tableau, un message s'affiche proposant de redémarrer le jeu ou de le quitter, comme illustré Figure 4. Pour la fenêtre de dialogue qui s'affiche à la fin du jeu, jetez un coup d'oeil du côté de JOptionPane. Cette fenêtre de dialogue est créée par le contrôleur.

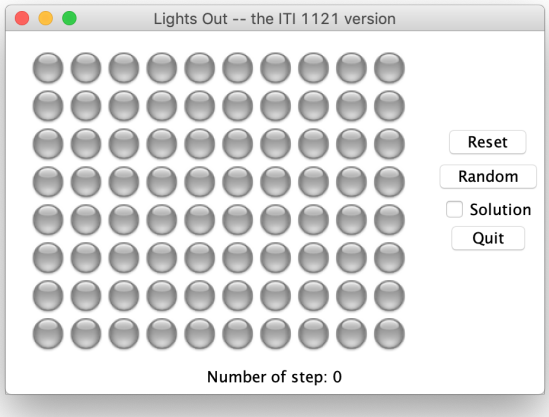

Figure 3 – Le rendu initial de l'IUG avec la taille par défaut.

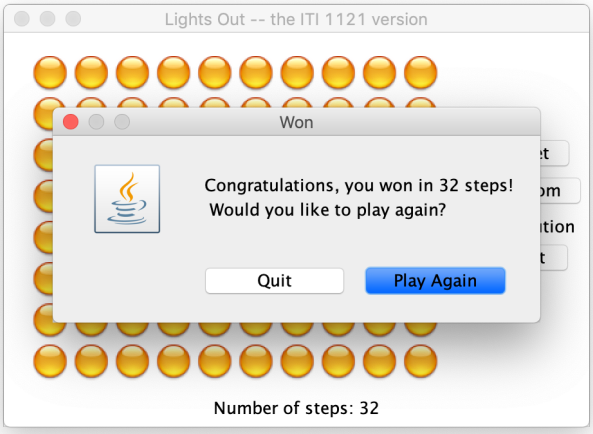

Figure 4 – Un jeu complété.

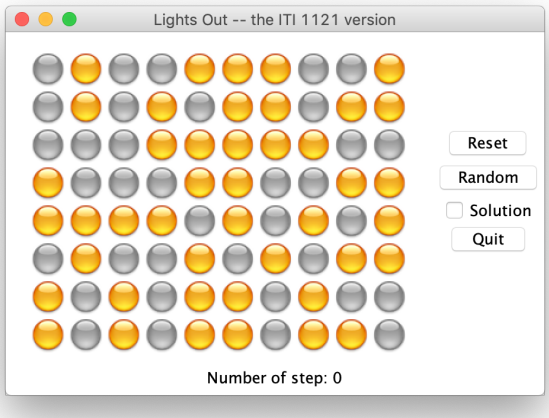

FIGURE 5 – Un jeu aléatoire.

Lorsque l'utilisateur appuie sur le bouton «*random*», le jeu est remis à zéro et le nouveau tableau de jeu est affiché. Le tableau de jeu a été initialisé avec des valeurs aléatoires. Cependant, vous devez vous assurer que ce tableau est réalisable. La Figure 5 montre un exemple.

Finalement, lorsque la case à cocher est sélectionnée, une solution de taille minimale est superposée sur le tableau. Les positions qui doivent être sélectionnées pour compléter la solution sont affichées à l'aide de nouvelles icônes, telles que présentées pour la Figure 6.

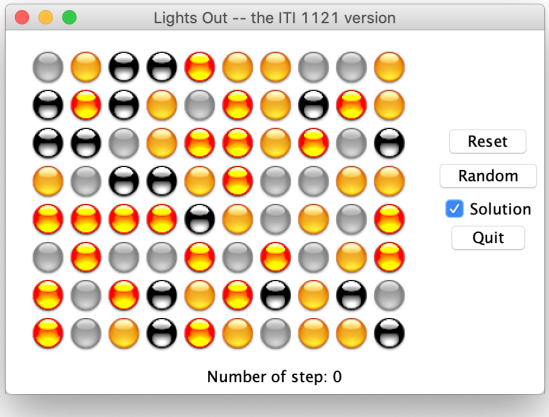

FIGURE 6 – Affichage d'une solution.

Tant que la case à cocher est sélectionnée, la solution doit être affichée et donc mise à jour en conséquence. Consultez la Figure 7 pour un exemple.

On lance cette application à partir de la méthode **main** de la classe **LightsOut**. Le code est fourni.

[2] http://www.site.uottawa.ca/~turcotte/teaching/iti-1521/lectures/09/puzzler-src.zip

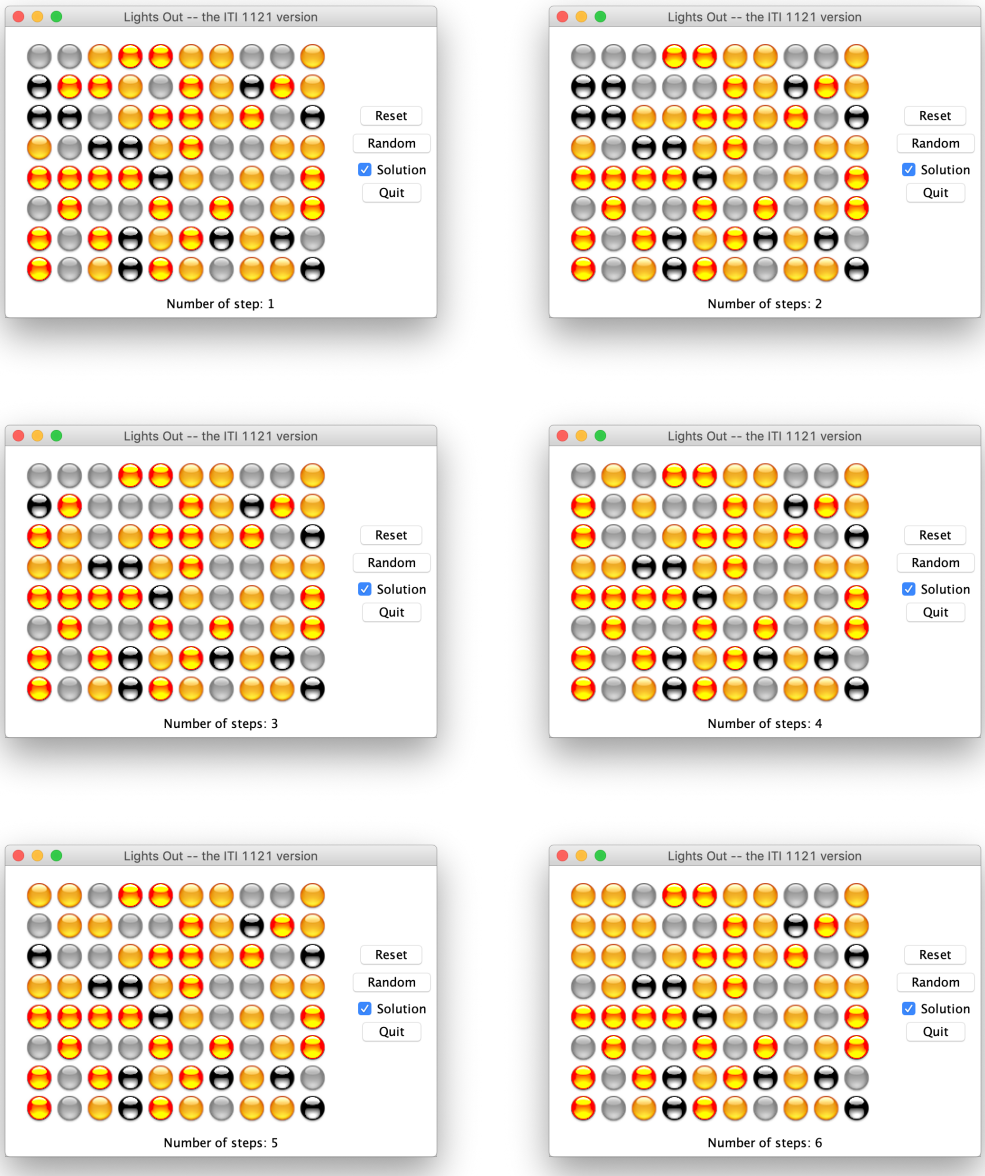

FIGURE 7 – Jouer tout en affichant la solution.

Intégrité dans les études

Cette partie du devoir a pour but de sensibiliser les étudiants face au problème de fraude scolaire (plagiat). Lisez les deux documents qui suivent.

- www.uottawa.ca/administration-et-gouvernance/reglement-scolaire-14-autres-informations-importantes
- www.uottawa.ca/vice-recteur-etudes/integrite-etudes

Les règlements de l'université seront appliqués pour tout cas de plagiat. En soumettant ce devoir, je certifie que :

1. J'ai lu les règlements sur la fraude scolaire.
2. Je comprends les conséquences de la fraude scolaire.
3. Sauf pour le code source fourni par les instructeurs du cours, tout le code source soumis est le mien.
4. Je n'ai pas collaboré avec d'autres personnes, à l'exception de ma coéquipière dans le cas d'un travail en équipe (de deux personnes).
    - Si vous avez collaboré avec d'autres personnes ou si vous avez obtenu du code source du Web, veuillez soit donner les noms de vos collaborateurs ou vos sources, ainsi que la nature de la collaboration. Mettez ces informations dans le fichier README.txt. Des points seront retranchés proportionnellement à l'aide reçue (0 à 100%).

Consignes

- Veuillez suivre les consignes que vous trouverez sur la page des consignes aux devoirs.
- Tous les devoirs doivent être soumis à l'aide de uottawa.brightspace.com.
- Vous devez préférablement faire le travail en équipe de deux, mais vous pouvez aussi faire le travail seul.
- Vous devez utiliser le gabarit ci-dessous.
- Les soumissions qui ne sont pas conformes aux exigences ne fonctionneront pas avec les outils que nous utilisons pour valider vos classes et en conséquence ne seront pas corrigées.
- Nous utilisons un outil informatique pour détecter les cas de plagiat. Les soumissions de toutes les sections (anglaises et française) sont comparées. Les soumissions identifiées par cet outil recevront la note 0.
- Vous devez vous assurer que Brightspace a bien reçu votre soumission. Vous ne pourrez pas soumettre des documents après l'échéance.
- Les soumissions en retard ne sont pas acceptées.

Fichiers

Vous devez remettre un fichier zip (aucun autre format ne sera accepté). Le répertoire principal a pour nom **a3_3000000_30000001**, où 300000 et 300001 sont les numéros d'étudiant des deux membres du groupe. Si vous travaillez seul, répétez votre numéro d'étudiant deux fois. Le nom du répertoire débute par la lettre minuscule **a** suivie du numéro du devoir, **3**. Les séparateurs sont les symboles soulignés («*underscore*») et non le tiret. Il n'y a pas d'espace dans le nom du répertoire. L'archive suivante, a3_3000000_3000001.zip, contient les fichiers que vous utiliserez comme point de départ. Votre soumission doit contenir les fichiers suivants :

- README.txt
    - C'est un fichier texte contenant le nom des membres de l'équipe, leur numéro d'étudiant, section, et une courte description du travail (une ou deux lignes suffisent).
- 01/GameModel.java
- 01/LightsOut.java
- 01/Q1Tests.java
- 01/Queue.java
- 01/QueueImplementation.java
- 01/Solution.java
- 01/StudentInfo.java
- 02/GameController.java

- 02/GameModel.java
- 02/GameView.java
- 02/GridButton.java
- 02/LightsOut.java
- 02/Queue.java
- 02/QueueImplementation.java
- 02/Solution.java
- 02/StudentInfo.java
- 02/Icons/Light-0.png
- 02/Icons/Light-1.png
- 02/Icons/Light-2.png
- 02/Icons/Light-3.png

Questions

Pour toutes vos questions, nous vous invitons à visiter le site Piazza du cours :

- https://piazza.com/uottawa.ca/winter2019/iti1521/home

A Exécuter TestQ1

Voici la sortie de **TestQ1** pour notre propre implémentation. L'exécution du code ne devrait pas générer d'échec, évidemment. La trace que vous obtenez peut être différente, et la la solution la plus courte peut être différente (mais de la même taille).

```
$ java Q1Tests
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************


testSolver
Solution found in 0 ms
Success!
Starting from :
[false,false]
[false,false]

Solution(s) :
[[true,true],
[true,true]]
Solution found in 0 ms
Success!
Starting from :
[false,false]
[true,false]

Solution(s) :
[[false,true],
[false,false]]
Solution found in 0 ms
Success!
Starting from :
[true,true]
```

```
[true,true]

Solution(s) :
[[false,false],
[false,false]]
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Success!
Starting from :
[false,false]
[false,false]
[false,false]

Solution(s) :
[[true,true],
[true,true],
[false,false]]
[[true,false],
[false,false],
[false,true]]
[[false,true],
[false,false],
[true,false]]
[[false,false],
[true,true],
[true,true]]
Success!
Starting from :
[true,false]
[false,false]
[false,false]

Solution(s) :
Solution found in 0 ms
Success!
Starting from :
[false,false,false]
[false,false,false]
[false,false,false]

Solution(s) :
[[true,false,true],
[false,true,false],
[true,false,true]]
Solution found in 0 ms
Success!
Starting from :
[true,false,false]
[false,true,false]
[false,false,true]

Solution(s) :
[[false,false,true],
[false,false,false],
[true,false,false]]
```

```
testShortest
For model :
[false,false]
[false,false]

Solution found in 0 ms
Success. Size found: 4
shortest solution found:
[[true,true],
[true,true]]
For model :
[false,false,false]
[false,false,false]

Solution found in 1 ms
Solution found in 1 ms
Solution found in 1 ms
Solution found in 1 ms
Success. Size found: 2
shortest solution found:
[[true,false,false],
[false,false,true]]
For model :
[false,false,true]
[true,true,false]

Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Success. Size found: 2
shortest solution found:
[[false,true,false],
[false,false,true]]
For model :
[true,true,false]
[true,false,false]

Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Success. Size found: 1
shortest solution found:
[[false,false,false],
[false,false,true]]
For model :
[false,false,false,false]
[false,false,false,false]
[false,false,false,false]
[false,false,false,false]

Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
```

```
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Solution found in 0 ms
Success. Size found: 4
shortest solution found:
[[false,true,false,false],
[false,false,false,true],
[true,false,false,false],
[false,false,true,false]]

testModel
Success
$
```

Version du 1er juin 2019