ITI 1121. Introduction to Computing II
Winter 2019

Assignment 3
(Last modified on March 22, 2019)

Deadline: March 24, 2019, 11:30 pm

[PDF]

Learning objectives

- Designing an application utilizing event-driven programming.
- The Model-View-Controller design pattern.
- Experimenting with software evolution.

Introduction

For this assignment, we are going to build a graphical user interface for our game *Lights Out*, using some of the code developed for assignment 2. You can base this assignment on your own code, or if you prefer you can start from the solution we have provided (or a mix of both).

- YouTube Video for assignment 3

1 Solving the Game [30 Marks]

The solution developed in assignment 2 was solving the game starting from a board that is completely OFF. We first need to generalize this work and be able to compute the solutions starting from any board positions.

1.1 GameModel

The class **GameModel** is used to capture the current state of the board. Its specification is as follows:

- **GameModel(int width, int height)**: the constructor. Creates an all OFF game of width **width** and of height **height**.
- **int getHeight()**: getter for height.
- **int getWidth()**: getter for width.
- **boolean isON(int i, int j)**: returns **true** if the location at **row** i and **column** j is ON, false otherwise.
- **reset()**: resets the model to all OFF.
- **set(int i, int j, boolean value)**: sets the location (i,j) of the model to "value" (be careful, that's **column** i and **row** j).
- **String toString()**: returns a String representation of the model.

1.2 Solution

We need to adapt our class **Solution** to the new configuration. **Important:** here we want to add to our application, so all of the modifications that we make should not change any of the existing public methods or their behaviour.

The new methods of our class are the following ones:

- **boolean stillPossible(boolean nextValue, GameModel model)**: this method returns **false** if the current solution would be impossible to finalize into a working solution for a board in the state specified by the GameModel instance **model**, should it be extended from its current state with the value **nextvalue**.

  Note that the method returns **false** if extending the current solution with **nextvalue** would never yield a working solution, and **true** otherwise. Returning **true** *does not mean that the solution can necessarily be extended into a working solution*. It merely means that we do not yet know if it is still possible.

  Note as well that this method should not modify the state of the object instance of *Solution* on which it is called. It does not extend that solution with **nextvalue**, it simply indicates if such an extension would annihilate its chance of leading to a working solution.

- **public boolean finish(GameModel model)**: this method assumes that the solution is currently still extendable for a board in the state specified by the GameModel instance **model**, but only one way. It keeps extending that solution with the one correct way that it finds at each step, until the solution is complete and correct for a board in the state specified by the GameModel instance **model**, or shown to not be work. It returns **true** if and only if the solution is extended into a complete, working solution

  That method does change the state of the object instance of *Solution* on which it is called. If it returns **true**, then that instance is now a complete, working solution for a board in the state specified by the GameModel instance **model**

- **public boolean isSuccessful(GameModel model)**: this method returns **true** if the solution is completely specified and is indeed working, that is, if it will bring a board of the specified dimensions from the state specified by the GameModel instance **model** to being entirely "on".

- **int getSize()**: returns the "size" of the solution, that is, the number of positions that must be tapped. On an initially OFF 3x2 game, the solution that consists of tapping (1,1) and (3,2) is of size 2, while the solution that consists of tapping (1,1), (2,1), (1,2) and (2,2) is of size 4.

.

As you implement these new methods, you should avoid code duplication as much as possible in your class, so you may have to do a little bit of refactoring.

1.3 LightsOut

Here are the methods to add to the class **LightsOut**

- **ArrayList<Solution> solve(GameModel model)**: The **class** method **solve** finds all the solutions to the *Lights Out* game for a board in the state specified by the GameModel instance **model**, using a *Breadth-First Search* algorithm. It returns an **ArrayList** containing all the valid solutions to the problem.

- **Solution solveShortest(GameModel model)**: The **class** method **solveShortest** returns a reference to a minimum size solution to the *Lights Out* game for a board in the state specified by the GameModel instance **model**. Note that there could be more than one such minimum-size solution. The method can return a reference to any one of them

Here as well, we do not want to change any of the existing public methods and behaviour, and you should avoid code duplication as much as possible.

1.4 Queue and a Queue Implementation

In assignment 2, our application relied on an interface **SolutionQueue**, which was a queue specifically meant for instances of **Solution**. The implementation of that interface was using an ArrayList.

We can now replace both interface and implementation with our own code. You need to provide a **Queue** interface that uses generics, and a queue implementation of your own making, also using generics. Make sure that your application now uses this interface and your implementation.

Note that the methods **solve** of the class **LightsOut** still returns an instance of "java.util.ArrayList<Solution>".

1.5 Testing

We provide a class **TestQ1** which can help you to ensure that your code is working as expected. A sample run is provided in Appendix A.

2 LightsOut GUI [70 Marks]

We now move on to creating our GUI for the game. Our game will have a few features. It is advisable to progress by iterations: first, get a working version of the game, on which the use can play and the correct information is presented. Once this works, add the capability to reset the game. Then, the capability to create a randomized game, and finally, the capability to superimpose a solution to the game.

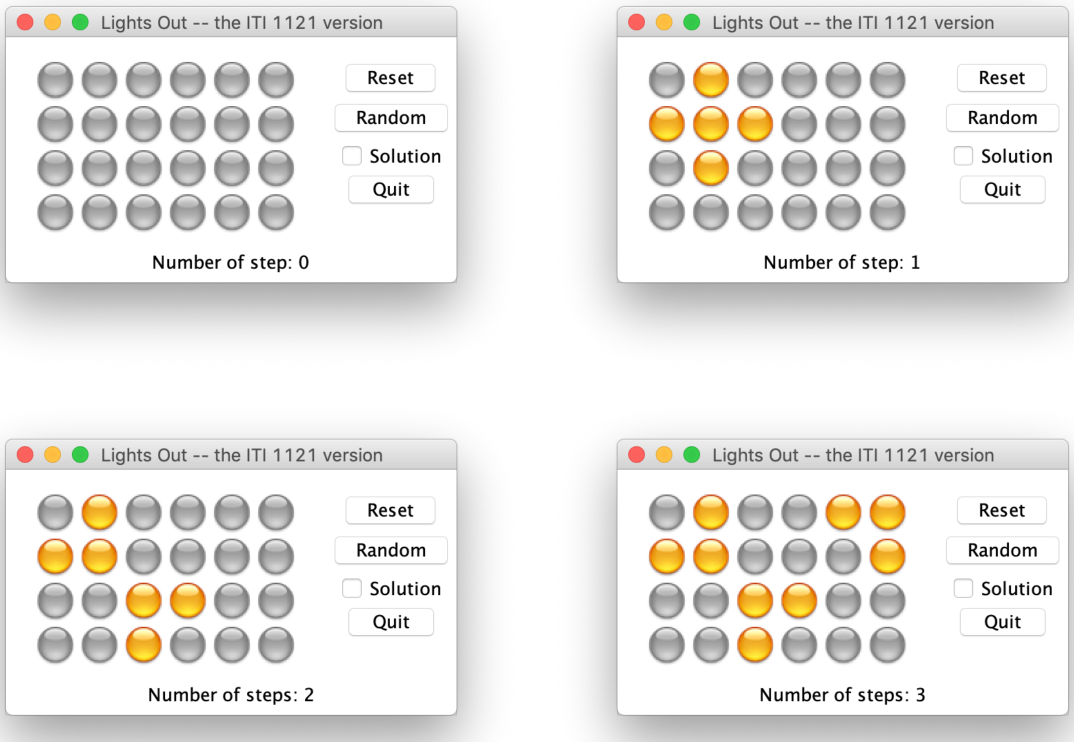


Figure 1: The GUI on Lights Out on a 6x4 game.

Model-View-Controller

Model-View-Controller (MVC) is a very common design pattern, and you will easily find lots of information about it on-line (e.g. Wikipedia, Apple, Microsoft to name a few). The general idea is to separate the roles of your classes into three categories:

- The **Model**: these are the objects that store the current *state* of your system.
- The **View** (or views): these are the objects that are representing the model to the user (the UI). The representation reflects the current state of the model. You can have several views displayed at the same time; though in our case, we will have just one.
- The **Controller**: these are the objects that provide the logic of the system, how its state evolves over time based on its interaction with the "outside" (typically, interactions with the user).

One of the great advantages of MVC is the clear separation it provides between different concerns: the model only focuses on capturing the current state, and doesn't worry about how this is displayed nor how it evolves. The view's only job is to provide an accurate representation of the current state of the model, and to provide the means to handle user inputs, and pass these inputs on to the controller if needed. The controller is the "brain" of the application, and doesn't need to worry about state representation or user interface. Thus each part can be designed, programmed, and tested independently.

In addition to the separation, MVC also provides a logical collaboration-schema between the three components (Figure 2). In our case, it works as follows:

1. When something happens on the view (in our case, mostly when the user clicks the a position on the board), the controller is informed (message 1 of Figure 2).
2. The controller processes the information and updates the model accordingly (message 2 of Figure 2).
3. Once the information is processed and the model is updated, the controller informs the view (or views) that it should refresh itself (message 3 of Figure 2).
4. Finally, each view re-reads the model to reflect the current state accurately (message 4 of Figure 2).

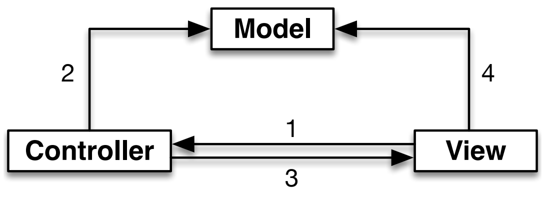


Figure 2: The collaboration between the Model, the View and the Controller.

- YouTube Video for the Model-View-Controller Design Pattern

The model

The first step is to build the model of the game. We already have a starting point with the class built for the previous section, so we only need to complete it.

First, we need to add yet another method to the class Solution: a method to see if a given position of bord is selected or not. Add the following method to the class **Solution**:

- **boolean get(int i, int j)**: returns true if the position at column i, row j is tapped in the solution, false otherwise

We then update the class **GameModel** with the following methods:

- **click(int i, int j)**: update the model after the dot at location row i and column j is clicked. It does change the state of that dot and the state of its neighborhood according to the logic of the game.
- **int getNumberOfSteps()**: the number of times the method click has been called since the last reset (or since the beginning if the game was never reset).
- **boolean isFinished()**: returns true if the model represents a completed game.
- **randomize()**: restarts the game with a solvable random board instead of an all OFF board.
- **void setSolution()**: forces the model to find a minimal size instance of Solution for the current model.
- **boolean solutionSelects(int i, int j)**: returns true if the model has been forced to find a solution and the position at row i, column j is tapped in that solution.

Note that it is not the model's responsibility to ensure that the solution it was asked to compute matches the current state of the model. If the model has an outdated solution, or was never asked to compute a solution, then the value returned by *solutionSelects* will be unreliable.

A detailed description of the classes can be found in the documentation.

The Controller

Note: as a development strategy, we suggest that you first create a very simple, temporary text-based view that prints the state of the model (via the model's **toString()** method) and asks the user to input the next square to select (by asking for a column and a line). Having this basic view will give you an opportunity to test your model and your controller independently from building the GUI.

The controller is implemented in the class **GameController**. The constructor of the class receives the width and height of the board. The instance of the model and the view are created in this constructor. It is also the controller which is the "listener" for all the components on the UI. A detailed description of the classes can be found in the documentation.

The view

We finally need to build the UI of the game. This will be be done based on the class **GameView**, which extends JFrame. It is a window which shows at the bottom the number of steps played so far, and on the right side three buttons, one to reset the game, one to randomize the game and one to quit the game. A checkbox is also available to superimpose the solution to the current board.

The board itself is made of a series of squares, h lines and w columns of them (where h is the height of the board, w is the width of the board). Figure 3 shows the full GUI.

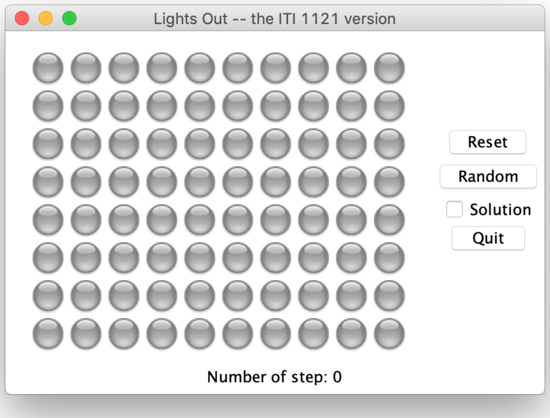


Figure 3: The initial GUI of the game with the default size.

To implement the square, we will write the class **GridButton** which extends the class JButton. GridButton is based on **Cell** [1] from Puzzler by Apple. You can review the code of the program "Puzzler" seen in lab 7 to help you with the class **GridButton**. The code for **GridButton** includes a method **getImageIcon()** which uses the current value of the instance variable **icon** to find the correct **ImageIcon** reference. In order to use **getImageIcon()**, the series of icons must be put in a subdirectory called "icons". We have provided 4 icons corrponsing to the 4 possible states of a position (selected/not selected in normal mode, and selected/not selected as part of a solution being displayed).

The game

When the user starts the game, an completely OFF board of the specified size is shown. Throughout the game, the number of clicks is shown at the bottom of the window. When the user completes the board, a message is displayed offering to either restart a game or quit as shown Figure 4. For the dialog window that is displayed at the end of the game, have a look at the class JOptionPane. This dialog window is created by the controller.

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

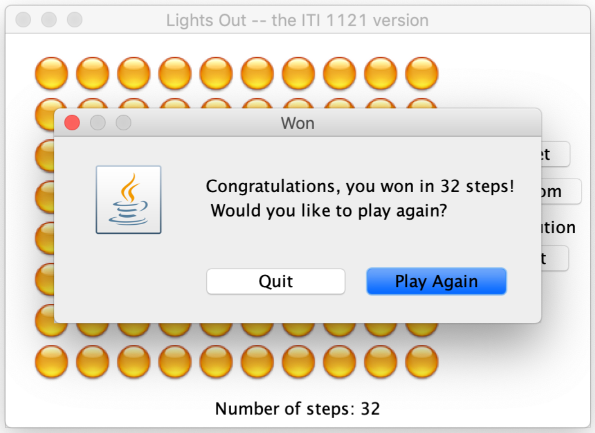

Lights Out -- the ITI 1121 version
Won
Congratulations, you won in 32 steps!
Would you like to play again?
Quit
Play Again
Number of steps: 32

Figure 4: A finished game.

When the user clicks the button "random", the game is reset and a new board is displayed. That board has been randomly created. However, you must implement this in such a way that the board is solvable. Figure 5 shows an example.

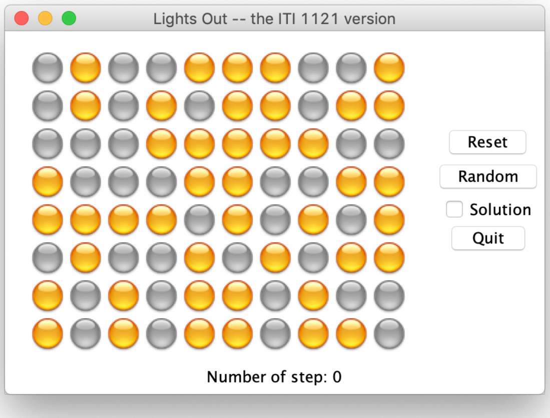


Figure 5: A randomized game.

Finally, when the "solution" checkbox is selected, a minimum size solution is superimposed to the board. The position that must be selected to solve the game are displayed using slightly different icons, as shown Figure 6.

As long as the checkbox is selected, the solution must be shown and therefore updated accordingly. See Figure 7 for an example.

The game is started from the main of the class **LightsOut**. The code is provided.

Academic Integrity

This part of the assignment is meant to raise awareness concerning plagiarism and academic integrity. Please read the following documents.

- https://www.uottawa.ca/administration-and-governance/academic-regulation-14-other-important-informat

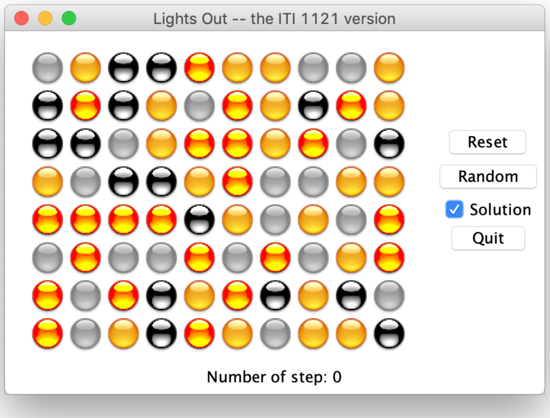


Figure 6: Showing the solution.

- https://www.uottawa.ca/vice-president-academic/academic-integrity

Cases of plagiarism will be dealt with according to the university regulations. By submitting this assignment, you acknowledge:

1. I have read the academic regulations regarding academic fraud.
2. I understand the consequences of plagiarism.
3. With the exception of the source code provided by the instructors for this course, all the source code is mine.
4. I did not collaborate with any other person, with the exception of my partner in the case of team work.
    - If you did collaborate with others or obtained source code from the Web, then please list the names of your collaborators or the source of the information, as well as the nature of the collaboration. Put this information in the submitted README.txt file. Marks will be deducted proportional to the level of help provided (from 0 to 100%).

Rules and regulation

- Follow all the directives available on the assignment directives web page.
- Submit your assignment through the on-line submission system virtual campus.
- You must preferably do the assignment in teams of two, but you can also do the assignment individually.
- You must use the provided template classes below.
- If you do not follow the instructions, your program will make the automated tests fail and consequently your assignment will not be graded.
- We will be using an automated tool to compare all the assignments against each other (this includes both, the French and English sections). Submissions that are flagged by this tool will receive the grade of 0.
- It is your responsibility to make sure that BrightSpace has received your assignment. Late submissions will not be graded.

Files

You must hand in a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **a3_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The name of the folder starts with the letter "a" (lowercase), followed by the number of the assignment, here 3. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a3_3000000_3000001.zip contains the files that you can use as a starting point. Your submission must contain the following files.

- README.txt
  - A text file that contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- 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

For all your questions, please visit the Piazza Web site for this course:

- https://piazza.com/uottawa.ca/winter2019/iti1121/home

Last modified: March 22, 2019

A Running TestQ1

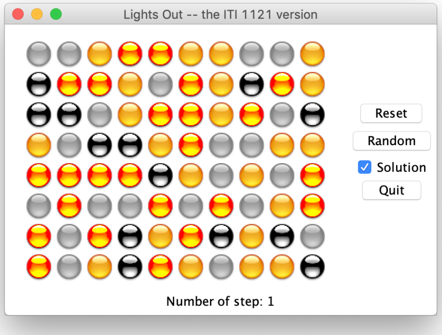
This is the output of TestQ1 on our own implementation. You run should not generate any failure, obviously. The trace that you get might be different, and the "shortest" solution might be different (but of the same size).

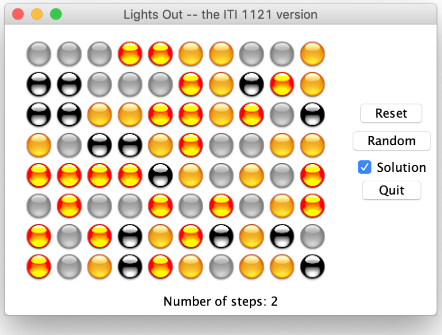
```
$ java Q1Tests
**********************************************************
*                                                        *
*                                                        *
*                                                        *
*                                            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


testSolver
```

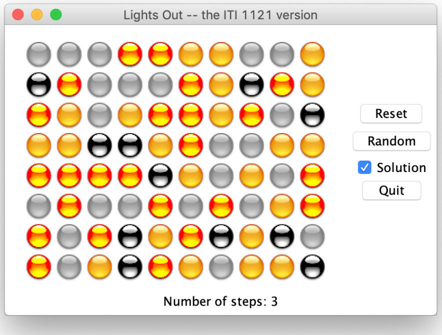
```
Solution found in 0 ms
Success!
Starting from :
[false,false]
[false,false]

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) :
```

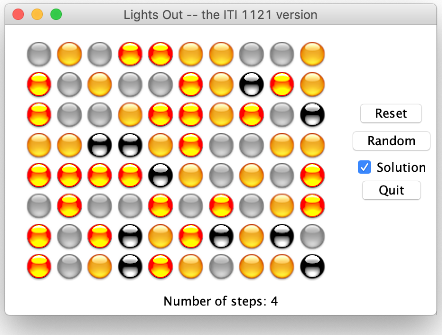
```
Solution found in 0 ms
Success!
Starting from :
[false,false,false]
[false,false,false]
[false,false,false]

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]]
```

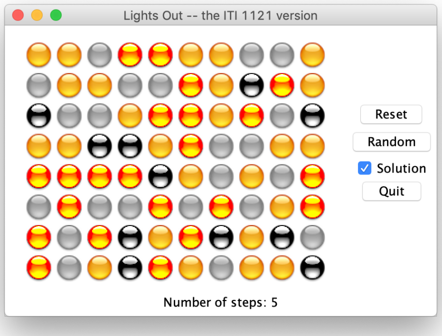
```
For model :
[true,true,false]
[true,false,false]

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
 $
```











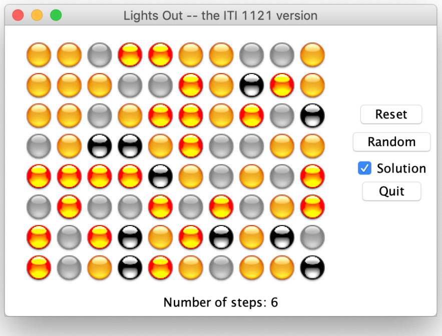


Figure 7: Playing while showing the solution.