

06/12/2017

BAALI Karim / ALGERA Pieter

ADSA Corp – IBO 1

**Shop Quest**

Maze solving game against AI

Table of contents

[Game presentation 2](#_Toc500275400)

[Maze generator 2](#_Toc500275401)

[Artificial Intelligence for maze solving 2](#_Toc500275402)

[User interface using Pygame libraries 2](#_Toc500275403)

[Improvement possibilities 2](#_Toc500275404)

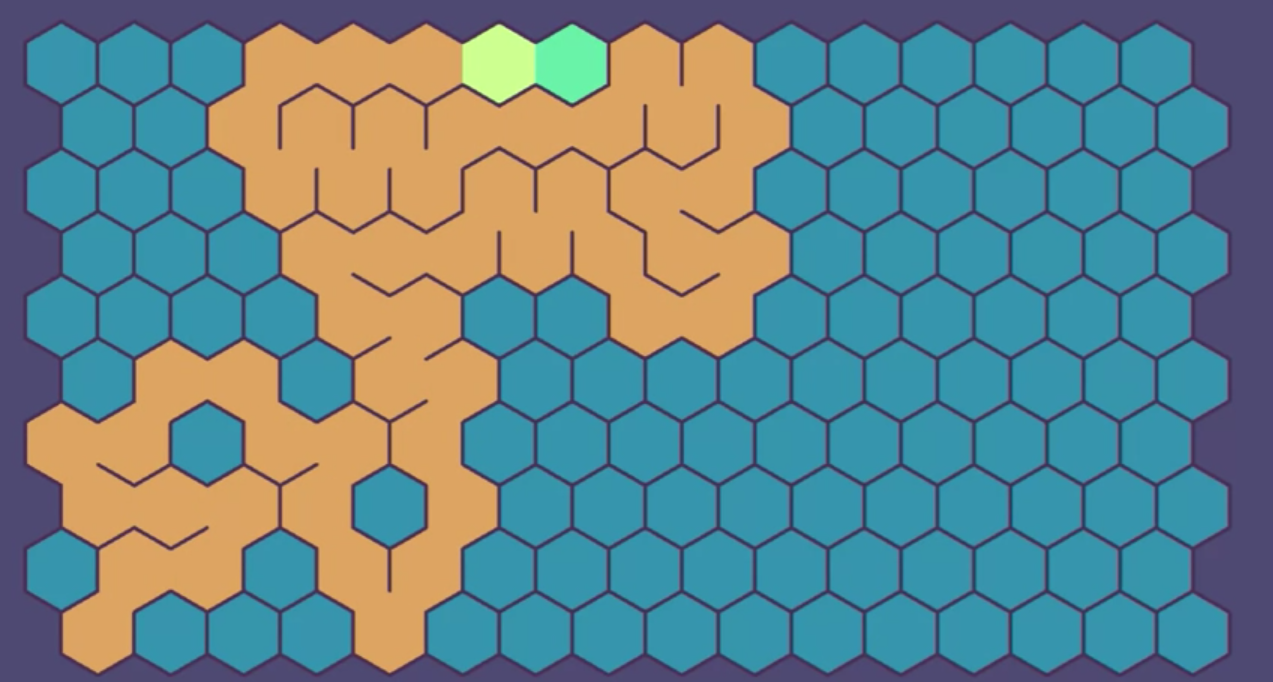
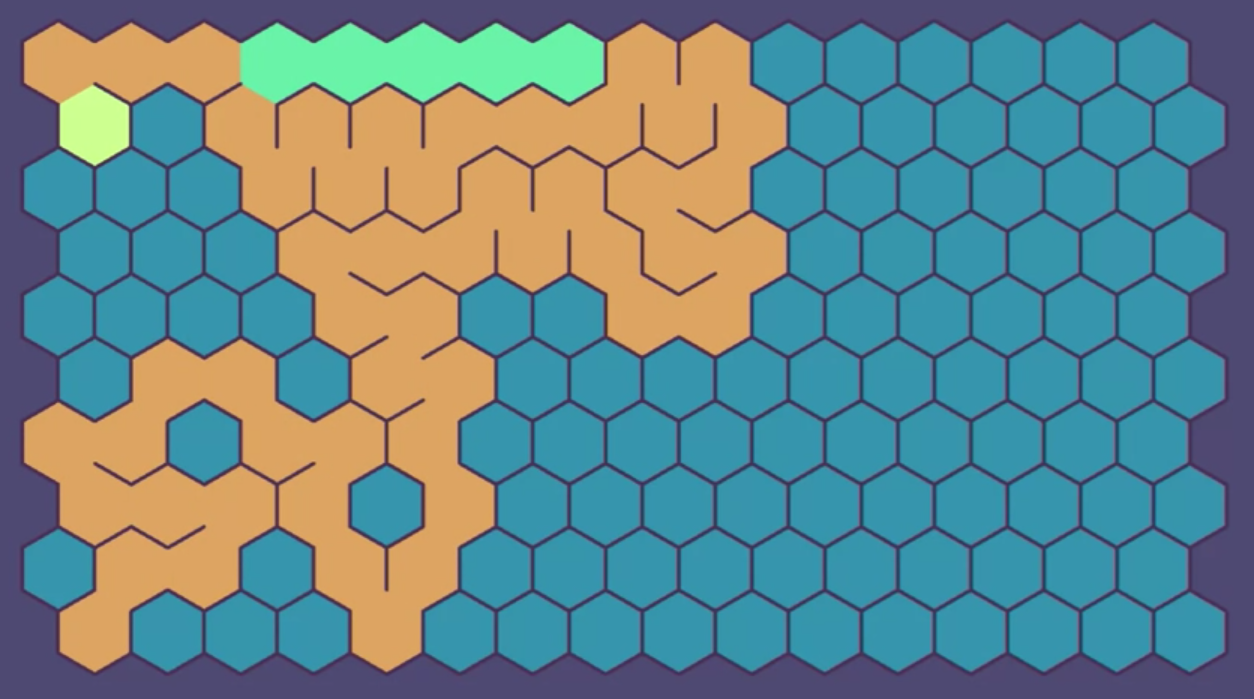
# Game introduction

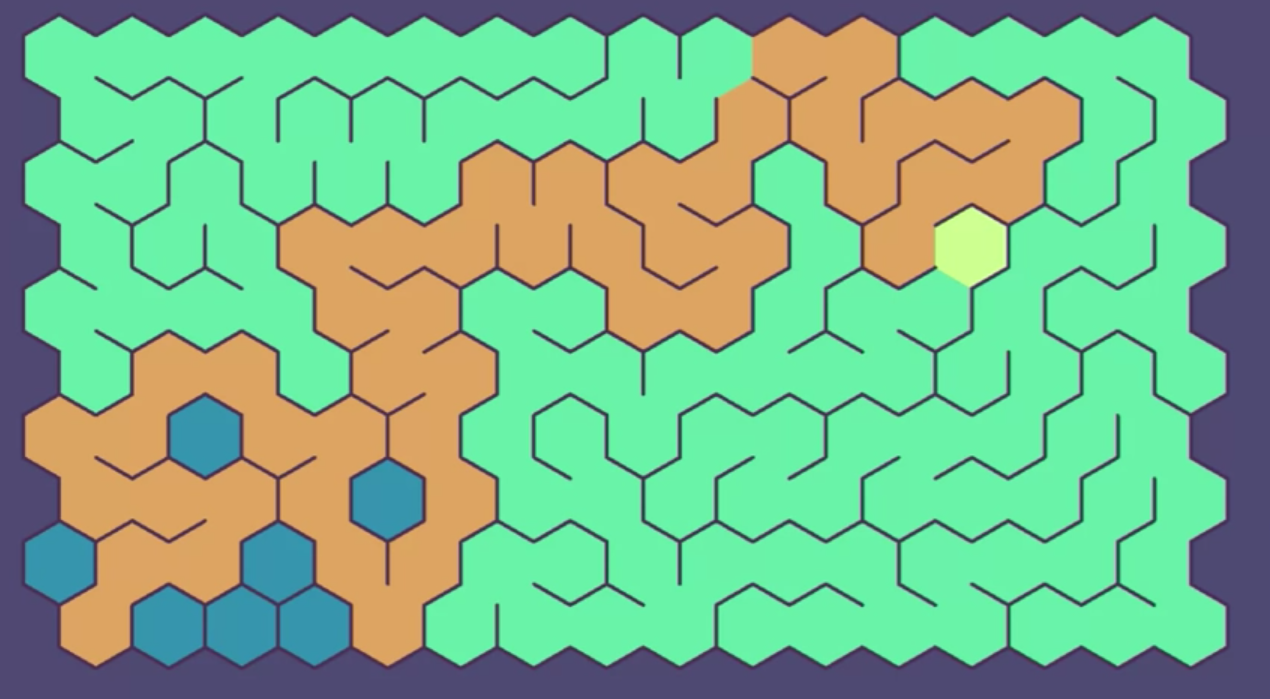
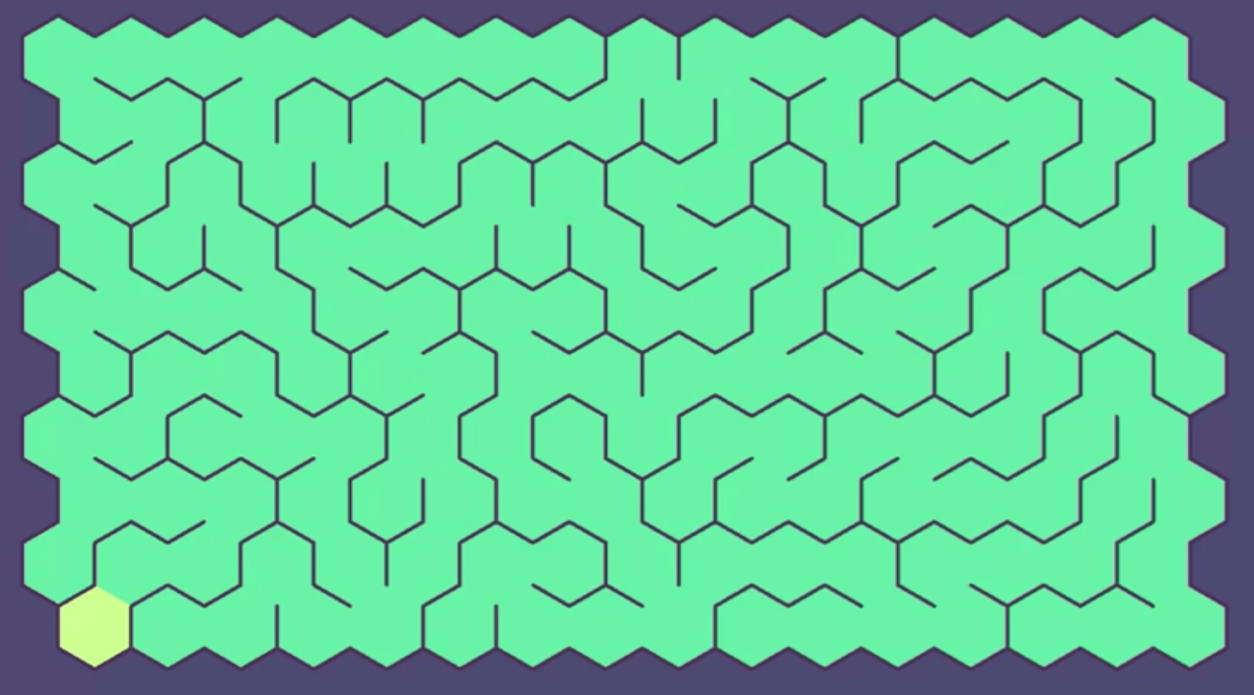
You wake up in a big shop … which looks really strange. After a few seconds, you understand that it’s a maze… a real one! Your goal? Find the wonderful banana before the AI. Who’s gonna be the faster one?

For each game, a new maze is generated by our program, allowing you to discover a whole new maze whenever you want. Using directional keys, your job is to go through the maze and find the holy banana. But you’re not alone! Another player, controlled by our \*very\* smart AI, will try to reach it before you, and its maze solving ability is pretty efficient.

# Maze generator

First thing first, the maze generator. To be able to provide a randomly generated maze for each game, we decided to implement a depth-first recursive backtracker algorithm. Starting at a point, the algorithm chooses a random neighbour of the current cell, if it’s unvisited, so it marks it as visited, remove the wall between the two cells, add it to the path, and move on it. If there is no more unvisited cell in the neighbours list, it’s go back to the last one on the path, and applies this logic until all the cells have been visited.

(Source : <https://en.wikipedia.org/wiki/Maze_generation_algorithm> )

In the example above, the visited cells are in orange, when the algorithm can’t find any unvisited neighbours (image 1), it comes back until it finds a new possible path. The cells which are on this backtracking path are marked in green. A the end, we can see that all the visited have been visited and there is no more unvisited cells.

To simplify the work on the generation, we decided to dissociate the logical matrix of the maze and the displayed matrix. In the logical one, each cell has data about its wall state (using a boolean array as attribute). It’s this one which is used by the algorithm to generate the maze. Then, when the generating part is over, we update the visual one, used for file printing and display. Its size is more important because each wall has a dedicated place in the matrix. So we can switch from the logical one width and height to the visual with the following formula :

visualWidth = (logicalWidth \* 2) + 1

visualHeight = (logicalHeight \* 2) + 1

Implementing the matrices this way is a bit tricky, but it fells like the easiest way on the moment. It could be useful to go through the process again to simplify it later.

When everything has been updated, the visual matrix is printed in a file, which will be read by the second part of the program to create the graphical interface.

# Artificial Intelligence for maze solving

# User interface using Pygame libraries

# Improvement possibilities