2D Snake Game Using Cellular Automata Tutorial

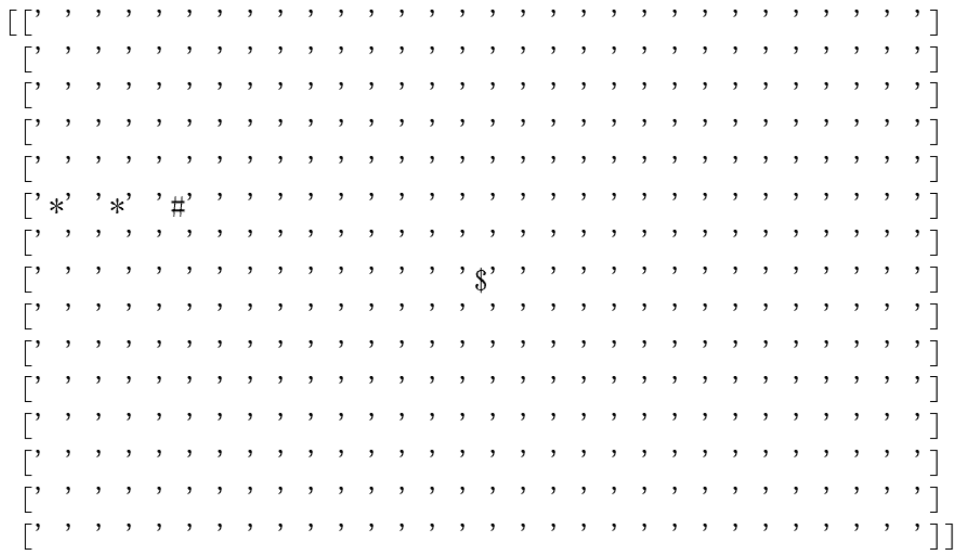
Group 59
Tianyang Cao
Yanmo Xu
Yuhao Chen

GitHub Repo Link:
https://github.gatech.edu/tzheng34/CSE6730-Group-59-Project1

1. Introduction

In this project, we implemented a 2D snake game in Jupyter NoteBook. We created a snake which automatically searched the target and approached it. This snake followed some specific target searching algorithm which helped the snake eat as much food as possible. The map was a 2D cell grid and the snake was a linked-list.

The main course concept behind this project is cellular automata (CA). Essentially, the entire game can be represented by the 2D cell grid map. This 2D cell array is a complex cellular automata system. Each cell is a pixel in the real-time display window. A cell has several states and continuously updates those state values at each time step. The states update rule for each cell is a core component of the CA system. The real-time display window just continuously displays each cell by evaluating its states at each time step, thus creating the game animation.



```
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 ['*' '*' '#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' '$' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']]
```

Figure 1 Initial Snake Game Map

- ' ' is a cell
- '$' is food or target
- '#' is snake head
- '*' is snake body

2. Code Tutorial

This section walks you through the Python code that makes our project happen. It is in great detail hpoing that you can thoroughly understand how we designed our program step by step. Let's wait no more!!

First of all, we need to import a few standard Python libraries that will be utilized in the codes later.

In [1]:
```
import numpy as np
import time
import random
from IPython.display import clear_output
```

Now, the actual project code begins. The body of the snake is a big component of the game, and it's a good idea to create a special class for it. Therefore, we defined the following class called 'SnakeNode' to represent a block of the snake. (the linked list data structure will be used to connect these separate snake nodes together)

In [2]:
```
# Class 1: Linked List Node to represent a snake body block
class SnakeNode:
    def __init__(self, parent, child, is_head, curr_moving_dir, row_idx, col_idx):
        self.parent = parent
        self.child = child
        self.row_idx = row_idx
        self.col_idx = col_idx
        # is_head: 1-yes, 2-no
        self.is_head = is_head
        # dir: 1-up, 2-down, 3-left, 4-right
        self.curr_moving_dir = curr_moving_dir
```

This class has several fields. The parent and child fields store the upstream and downstream neighbor nodes of the current node. The row and column index fields store the node's current coordinates in the 2D game map. The is_head field shows whether this node corresponds to the snake head. The last field contains the current movement direction of the snake head (relevant only for the head node).

The creation of this class will be tremendously helpful when we manipulate the snake during the game.

Next, let's create some helper functions related to the SnakeNode class!

In [3]:
```
# Helper Function created for dealing with SnakeNode objects
def boundary_hit_check(SnakeNode,max_row_idx,max_col_idx):
    is_hit_boundary = 0
    if SnakeNode.row_idx < 0 or SnakeNode.row_idx > max_row_idx:
        is_hit_boundary = 1
    elif SnakeNode.col_idx < 0 or SnakeNode.col_idx > max_col_idx:
        is_hit_boundary = 1
    return is_hit_boundary
```

This first function takes in a SnakeNode object as well as the maximum row and column indices a node can have(determined by the dimension of game map). It checks if this node is outside of the game map or not and returns the result.

In [4]:
```python
# Helper Function created for dealing with SnakeNode objects
def head_body_collision_check(HeadNode,BodyNode):
    is_collision = 0
    if HeadNode.row_idx == BodyNode.row_idx and HeadNode.col_idx == BodyNode.col_idx:
        is_collision = 1
    return is_collision
```

This second function takes in the current snake head node and another body node, and checks if they are in collision with each other (i.e. have the same coordinates). The reason why this function doesn't more generally check collision between any two nodes is the following: Quite intuitively, a snake body collision can only happen between its head and a body node. It's impossible for two body nodes to collide.

In [5]:
```python
# Helper Function created for dealing with SnakeNode objects
def snake_update_one_time_step(HeadNode,TailNode,max_row_idx,max_col_idx,food_row,food_col
):
    is_game_over = 0
    is_food_eaten = 0
    old_head_row = HeadNode.row_idx
    old_head_col = HeadNode.col_idx
    ret_new_head = None

    # Update the position of HeadNode first, this saves time for head-body collision check
    # later.
    if HeadNode.curr_moving_dir == 1:
        HeadNode.row_idx = HeadNode.row_idx - 1
    elif HeadNode.curr_moving_dir == 2:
        HeadNode.row_idx = HeadNode.row_idx + 1
    elif HeadNode.curr_moving_dir == 3:
        HeadNode.col_idx = HeadNode.col_idx - 1
    else:
        HeadNode.col_idx = HeadNode.col_idx + 1

    # Check if the head hits any boundary
    is_hit_boundary = boundary_hit_check(HeadNode,max_row_idx,max_col_idx)
    if is_hit_boundary == 1:
        is_game_over = 1

    # check if the head 'eats' the food
    if HeadNode.row_idx == food_row and HeadNode.col_idx == food_col:
        HeadNode.is_head = 0
        HeadNode.row_idx = old_head_row
        HeadNode.col_idx = old_head_col

        # set the position of the new head at the food block
        new_head_row_idx = food_row
        new_head_col_idx = food_col
        new_head_moving_dir = HeadNode.curr_moving_dir

        # create the new head node
        new_HeadNode = SnakeNode(None, HeadNode, 1, new_head_moving_dir, new_head_row_idx,
new_head_col_idx)
        HeadNode.parent = new_HeadNode
        ret_new_head = new_HeadNode
```

```
            is_food_eaten = 1

    curr_node = TailNode
    while curr_node != None:
        if is_game_over == 1 or is_food_eaten == 1:
            break
        # If the current node is a body node
        if curr_node.is_head != 1:
            # update the position of the current body node
            if curr_node.parent.is_head == 1:
                curr_node.row_idx = old_head_row
                curr_node.col_idx = old_head_col
            else:
                curr_node.row_idx = curr_node.parent.row_idx
                curr_node.col_idx = curr_node.parent.col_idx
            # next, check if it collides with the head node
            if head_body_collision_check(HeadNode, curr_node) == 1:
                is_game_over = 1
                break

        curr_node = curr_node.parent

    return is_game_over, is_food_eaten, ret_new_head
```

This final function is the most important one here. On a high level, it takes in the current conditions of the snake together with the map information, and then updates the snake nodes for the next time step. Our discrete CA system evloves at each time step, and this function does this essential job.

Here's what the function does in more details. At the beginning, it updates the snake head node's coordinate fields based on its current moving direction field value. Right after this, the helper function is called to check if the new head location goes beyond the map. One thing to note here is that, for hitting-the-boundary check, it's sufficient to check only for the head node at each update time. Obviously, it's impossible for any body node to hit the boundary without the head hitting it first.

Next, the function checks whether the head node hits the food block in the map. If so, a new SnakeNode object is created and set as the new head node of the snake. Then the old head node is set back to a body node accordingly. In this case, the original snake body positions will not be moved for the next game iteration. Instead, the new head node emerges at the food block position. This creation of new head and subsequent adjustment require some careful manipulation of the snake linked list.

Finally, it's fairly easy to update the position of each body node. Starting at the node immediately after the head and looping through until the tail, for each of them, the new coodinates are set to be the old coordinates of its parent node (before the update). Visually, this process moves each snake body block to its upstream neighbor's place. The function returns some useful information in the end.

Now, we finished creating the snake node class and some relevant functions. Next, let's shift focus to another class called 'GameConfig'.

The purpose of this class is to hold information of the game environment setup and the game state/configuration so that they can be easily accessed by other functions. Below are the class definitions and class functions. The detailed explanation is embedded in the comments inside the code block.

```
In [6]: # Class 2: Game Environment & Configuration
```

```
# For this GameConfig class, it has the following fields: a 2D integer array
# that represents the game map, the current coordinates of food block,
# and the dimension of the map. The entries of the 2D array can
# only have values 0, 1, or 2. 0 means an empty cell, 1 means a snake body occupied cell,
# and 2 means a food block.
class GameConfig:
    def __init__(self,map_num_row, map_num_col, init_food_row, init_food_col):
        self.map_array = [[0 for x in range(map_num_col)] for y in range(map_num_row)]
        self.food_row = init_food_row
        self.food_col = init_food_col
        self.num_row = map_num_row
        self.num_col = map_num_col

# The next funcion initializes the 2D game map array. It takes in the initial snake
# node coordinates (stored as tuple pairs in the list 'curr_snake_coords') and the
# food location. Then the corresponding cells in the 2D array are set to the correct value
s.
    def initalize_map(self, curr_snake_coords):
        # set initial food block in array to value 2
        self.map_array[self.food_row][self.food_col] = 2
        # set snake block blocks to value 1
        for i in range(len(curr_snake_coords)):
            curr_snake_row = curr_snake_coords[i][0]
            curr_snake_col = curr_snake_coords[i][1]
            self.map_array[curr_snake_row][curr_snake_col] = 1

# The next two functions intend to update the snake head and tail locations
# in the 2D game map array stored in a GameConfig object. This differs from
# the SnakeNode update function earlier in the sense that here we need to update the
# values in the 2D array, while the other one takes care of all the SnakeNode objects.
# In this case, the process is straightforward: just change the old head/tail cell value
# back to 0 and then set the new locations to 1. This completes the map update for
# visualization purpose.
    def update_snake_tail_in_map(self, is_food_eaten, prev_tail_row, prev_tail_col):
        if is_food_eaten == 1:
            self.map_array[prev_tail_row][prev_tail_col] = 1
        else:
            self.map_array[prev_tail_row][prev_tail_col] = 0

    def update_snake_head_in_map(self, HeadNode):
        head_row_idx = HeadNode.row_idx
        head_col_idx = HeadNode.col_idx
        self.map_array[head_row_idx][head_col_idx] = 1

# The next function aims to update the food location in the 2D game map. It's called whene
ver the
# current food is eaten by the snake and thus a new food block needs to appear. We used th
e
# 'randint' functions to randomly select a spot in the 2D map array as the new food locati
on. However,
# there is one more step. We need to check if the new food location is valid or not. If it
coincides
# with any part of snake, then the function choose another random spot until a valid one i
```

```
s found.
# Also, the 2D game map array is updated with this new food cell as well.
    def update_food_in_map(self, curr_snake_coords, map_num_row, map_num_col):
        # first, the old food block now becomes part of snake
        old_food_row = self.food_row
        old_food_col = self.food_col
        self.map_array[old_food_row][old_food_col] = 1
        # then, randomly select a new food block
        is_valid = 0
        while is_valid == 0:
            new_food_row = random.randint(0,map_num_row-1)
            new_food_col = random.randint(0,map_num_col-1)
            is_valid = is_new_food_pos_valid(new_food_row, new_food_col, curr_snake_coords
)
        # finally, update the proper fields of the gameConfig object
        self.food_row = new_food_row
        self.food_col = new_food_col
        self.map_array[new_food_row][new_food_col] = 2

# The next function does the job of creating the game animation. We designed two game disp
lay schemes:
# 1. Directly display the 2D game map array with integer values 0,1,or 2
# 2. Convert the original integer 2D array to string type, and show the food as $, the sna
ke head
# as #, and the snake body as *
# At each iteration, this function clears the previous print output and then print the new
array. This
# way, the array display becomes dynamic.
    def print_map_array(self,HeadNode,curr_snake_coords):
        clear_output(wait=True)
        # We have two game map visualization schemes:
        # a:  0 - empty cell  ,  1 - snake(head+body), 2-food
        # b: ' '- empty cell  , '#'- snake head , '*' - snake body , '$' - food

        # Scheme a:
        #print np.matrix(self.map_array)

        # Scheme b:
        display_array = [[" " for x in range(self.num_col)] for y in range(self.num_row)]
        display_array[HeadNode.row_idx][HeadNode.col_idx] = "#"
        for i in range(1,len(curr_snake_coords)):
            display_array[curr_snake_coords[i][0]][curr_snake_coords[i][1]] = "*"
        display_array[self.food_row][self.food_col] = "$"
        print np.matrix(display_array)

# This final function is not part of the GameConfig class, but just a helper function for
the earlier
# class function 'update_food_in_map'. This helper function takes in the potential new foo
d's coordinates
# with the current snake blocks' locations, and determine/return whether the new food loca
tion is
# valid.
# helper function for game configuration handling
def is_new_food_pos_valid(new_food_row, new_food_col, curr_snake_coords):
    is_valid = 1
    for i in range(len(curr_snake_coords)):
        if new_food_row == curr_snake_coords[i][0] and new_food_col == curr_snake_coords[i
][1]:
            is_valid = 0
            break
    return is_valid
```

So far, we have gone through the two classes that will be used by the main function later, together with their special class functions. Next, we explore the last big part of the program -- smart computer player algorithm, in the form of a function. But first, let's quickly go over two helper functions:

In [7]:
```
# Helper Function for the computer player
def is_valid_move(move_dir, head, max_row_idx, max_col_idx,curr_snake_coords):
    is_valid = 1
    # Let the 'curr_snake_coords' array (temporarily) contain only the snake body blocks a
t the next time step
    deleted_snake_coord_entry = curr_snake_coords[-1]
    del curr_snake_coords[-1]

    # store original info of the head
    original_head_row_idx = head.row_idx
    original_head_col_idx = head.col_idx

    new_head_row = head.row_idx
    new_head_col = head.col_idx
    if move_dir == 1:
        new_head_row = new_head_row - 1
    elif move_dir == 2:
        new_head_row = new_head_row + 1
    elif move_dir == 3:
        new_head_col = new_head_col - 1
    else:
        new_head_col = new_head_col + 1

    # temporarily update HeadNode's position
    head.row_idx = new_head_row
    head.col_idx = new_head_col

    # check head-boundary collision
    if boundary_hit_check(head,max_row_idx,max_col_idx) == 1:
        is_valid = 0
        head.row_idx = original_head_row_idx
        head.col_idx = original_head_col_idx
        curr_snake_coords.insert(len(curr_snake_coords),deleted_snake_coord_entry)
        return is_valid

    # check head-body collision
    if head_body_collision_check_for_planner(head,curr_snake_coords) == 1:
        is_valid = 0
    # restore the original info and return
    curr_snake_coords.insert(len(curr_snake_coords),deleted_snake_coord_entry)
    head.row_idx = original_head_row_idx
    head.col_idx = original_head_col_idx
    return is_valid
```

This helper function above helps decide if a particular movement direction of snake head for the next game iterationis valid or not. A direction is valid if doing so won't result in either boundary collision or head-body collision. The function does the job by temporarily updating the current snake block coordinates list based on the direction-to-test. Then it runs some helper functions for collision check. Finally, it restores the original current snake block coordinates list and returns the test result.

In [8]:
```python
# Helper Function for the computer player
def head_body_collision_check_for_planner(head,body_coords_array):
    is_collision = 0
    for i in range(len(body_coords_array)):
        if head.row_idx == body_coords_array[i][0] and head.col_idx == body_coords_array[i
][1]:
            is_collision = 1
            break
    return is_collision
```

This helper function above checks for any head-body collision. It's specially designed for the computer planner since the input arguments are slight different than those of the similar function earlier. The motivation here is that this function can check through all the body nodes on a single function call, while the earlier version only checks for collision with a specific body node. The two versions are the more useful one in their own conditions.

Now we are ready to take on the important function below -- the smart computer player algorithm. In a nutshell, its task is to analyze the current game situation, and then 'smartly' decide which direction should the snake head go next. Remember, the goal of the snake is to eat as much food as possible while staying alive. Let's dive into the function itself:

In [9]:
```python
# Helper Function: Computer Player - Snake Next Move Direction Planner
def computer_player(head,max_row_idx, max_col_idx, food_row, food_col,curr_snake_coords):
    next_snake_move_dir = 0
    target_diff_row = food_row - head.row_idx
    target_diff_col = food_col - head.col_idx
    # try to reduce the row difference first,
    # and then reduce the column difference
    if target_diff_row > 0:
        next_snake_move_dir = 2
        if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_coord
s) == 1:
            return next_snake_move_dir
        # if can't reduce row diff, try to reduce col diff
        # if can't reduce col diff either, just pick a valid direction
        else:
            if target_diff_col > 0:
                next_snake_move_dir = 4
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 1
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 3
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
            elif target_diff_col < 0:
                next_snake_move_dir = 3
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 1
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 4
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
            else:
                next_snake_move_dir = 1
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 3
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 4
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
    elif target_diff_row < 0:
        next_snake_move_dir = 1
        if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_coord
s) == 1:
            return next_snake_move_dir
        # if can't reduce row diff, try to reduce col diff
        # if can't reduce col diff either, just pick a valid direction
        else:
            if target_diff_col > 0:
                next_snake_move_dir = 4
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 2
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 3
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
            elif target_diff_col < 0:
                next_snake_move_dir = 3
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 2
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 4
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
            else:
                next_snake_move_dir = 2
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 3
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
                next_snake_move_dir = 4
                if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_sna
ke_coords) == 1:
                    return next_snake_move_dir
    else:
        # if can reduce col diff, then do it
        # otherwise, try to go the opposite direction
        # since row diff = 0 now and we try not to change row
        if target_diff_col > 0:
            next_snake_move_dir = 4
            if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_c
oords) == 1:
                return next_snake_move_dir
            next_snake_move_dir = 3
            if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_c
oords) == 1:
                return next_snake_move_dir
        else:
            next_snake_move_dir = 3
            if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_c
oords) == 1:
                return next_snake_move_dir
            next_snake_move_dir = 4
            if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_c
oords) == 1:
                return next_snake_move_dir
        # if can't change column in any way, then pick a valid change in row
        next_snake_move_dir = 1
        if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_coord
s) == 1:
            return next_snake_move_dir
        next_snake_move_dir = 2
        if is_valid_move(next_snake_move_dir,head,max_row_idx,max_col_idx,curr_snake_coord
s) == 1:
            return next_snake_move_dir
    return next_snake_move_dir
```

The algorithm itself is not complicated. To concisely put it into steps:

1. obtain the food's relative position with respect to the head node
2. try to move in a direction that would bring the head closer to food (if two such directions are present, then pick one first)
3. call helper functions to check if this direction is a valid move
4. if valid, then immediately return this direction
5. if invalid, and at the same time if there's a second direction choice exists in step 2, then test its validity again
6. if valid, return it
7. if invalid, then we are left with two other directions that would make snake head further away from food. Therefore, check them in an arbitrary order for validity.
8. if a valid direction is found, return it
9. if both remaining directions are invalid, then return the next direction value as 0 to indicate that no valid movement can be made next. This means the snake will die/ game over at the next time step.

Finally, we arrive at the program's main function part. Here, we will see how all the earlier pieces are put together to create this 2D snake game with smart computer player:

First, let's define some variables. The ones below are user specified information for the game setup: the dimension of the map and the initial food block location. Right now they are set to be a 15-by-15 2D game map with the initial food block at (7,7). You are welcome to change it to anything you want!

In [10]:
```
# Game Environment Setup
map_num_row = 15
map_num_col = 15
init_food_row = 7
init_food_col = 7
```

The variables below are for general purpose. They assist the main loop later. They keep track of whether the game is over, whether the food is eaten, and the previous-iteration tail node location at each game iteration.

In [11]:
```
# General Purpose Variables
is_game_over = 0
is_food_eaten = 0
prev_tail_row = 0
prev_tail_col = 0
```

Next, we created an object of class GameConfig. The user input is used to initialize the game environment setup. This object will be responsible for holding the latest game configuration.

In [12]:
```
# create a game configuration object
game_config = GameConfig(map_num_row,map_num_col,init_food_row,init_food_col)
```

Moreover, we created three SnakeNode objects to initialize a length-3 snake located at some pre-determined location. We linked them properly so that now a linked list is created for the snake in the game. Also, the list 'curr_snake_coords' is created with the three initial snake block's coordinate pairs inside. This list is impotant since it will be used by many functions later.

In [13]:
```
# initially create a length-3 snake
curr_snake_coords = []
SnakeHead = SnakeNode(None, None, 1, 4, 5, 2)
SnakeBody = SnakeNode(SnakeHead, None, 0, 4, 5, 1)
SnakeHead.child = SnakeBody
SnakeTail = SnakeNode(SnakeBody, None, 0, 4, 5, 0)
curr_snake_coords.append([5,2])
curr_snake_coords.append([5,1])
curr_snake_coords.append([5,0])
```

The next section initializes the 2D game map array and then displays it. This is the starting condition for the game.

In [14]:
```
# initialize the map and print it
game_config.initalize_map(curr_snake_coords)
game_config.print_map_array(SnakeHead,curr_snake_coords)
```

```
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 ['*' '*' '#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' '$' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']]
```

Finally, we come to the main game loop. It is a while loop that keeps iterating unless the game is over. Each iteration is a time step of the CA system (or snake game dynamics). To offer a concise summary, at each game iteration, the code:

1. update the snake linked list according to the current head's movement direction
2. check if the game is over. If so, break the loop.
3. check if the food is eaten. If so, create a new food block.
4. update the 2D game map array contained in the game configuration object
5. display the current game map (replace the previous one)
6. call the computer player algorithm function to obtain the next head movement direction, update it in the head node
7. go on to the next iteration

The comments throughtout the code will help you understand what's going on.

In [15]:
```
# Main Game Loop
while is_game_over != 1:
    # record the previous tail position
    prev_tail_row = SnakeTail.row_idx
    prev_tail_col = SnakeTail.col_idx

    # Update the snake location for each SnakeNode
    is_game_over, is_food_eaten, ret_new_head = snake_update_one_time_step(SnakeHead,Snake
Tail,map_num_row-1,map_num_col-1,game_config.food_row,game_config.food_col)
    if is_food_eaten == 1:
        SnakeHead = ret_new_head
    # Update the array 'curr_snake_coords'
    if is_game_over == 1:
        break

    if is_food_eaten == 1:
        curr_snake_coords.insert(0,[game_config.food_row,game_config.food_col])
    else:
        del curr_snake_coords[-1]
        new_head_row = SnakeHead.row_idx
        new_head_col = SnakeHead.col_idx
        curr_snake_coords.insert(0,[new_head_row,new_head_col])

    # Update snake head location in the map
    game_config.update_snake_head_in_map(SnakeHead)
    # if the food is eaten, update the food in the map
    if is_food_eaten == 1:
        game_config.update_food_in_map(curr_snake_coords,map_num_row,map_num_col)

    # update snake tail location in the map
    game_config.update_snake_tail_in_map(is_food_eaten,prev_tail_row,prev_tail_col)

    # finally, print the game map after this new iteration
    game_config.print_map_array(SnakeHead,curr_snake_coords)

    # JEWEL ON THE CROWN: INTELLIGENT COMPUTER PLAYER
    # It intelligently pick the next moving direction for HeadNode
    next_move_dir = computer_player(SnakeHead,map_num_row-1,map_num_col-1,game_config.food
_row,game_config.food_col,curr_snake_coords)
    # Update the HeadNode's direction field with this new decision
    if next_move_dir == 0:
        print 'DEADLOCK: No valid movement possible'
        break
    else:
        SnakeHead.curr_moving_dir = next_move_dir

    is_food_eaten = 0
    # wait for 1 second between each game iteration
    time.sleep(0.3)

# if game is over, print
print 'Game Over'
```

```
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' '$' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '*' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '*' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '*' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '*' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '*' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' '*' '*' '*' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']]


---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-15-e5d8d5e7c63c> in <module>()
     45     is_food_eaten = 0
     46     # wait for 1 second between each game iteration
---> 47     time.sleep(0.3)
     48 
     49 # if game is over, print

KeyboardInterrupt: 
```

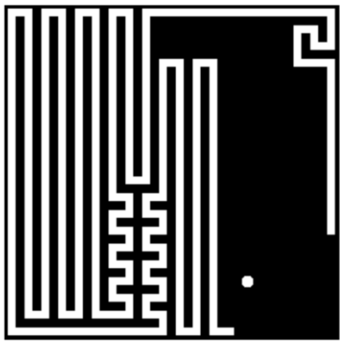
The visual animation above is the display window. As you can see, the computer planner does a pretty good job of eating food and staying alive.

This is the end of the program. We hope you understand how this game was created. Cellular Automata is a very powerful modeling tool, and we just showed a small fun application of it. Thank you so much for reading through the code!

3. Algorithm Improvement

Sometimes, we can clearly see that there are still ways to approach the food but our snake chooses the wrong way and dies. So we need to improve our algorithm.

We can find some inspiration from the picture below. You can see that the snake can grow up to fill all the girds.



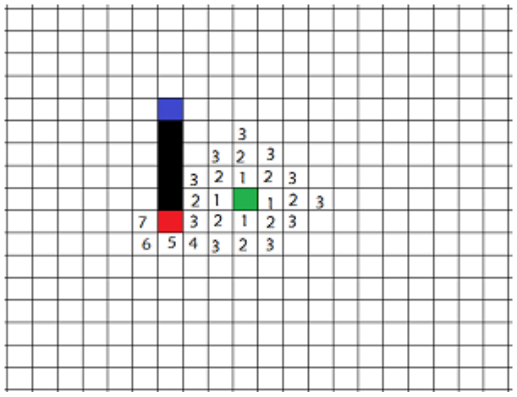
Before we start, we have two basic principles.

- First, the snake moves following the tail. Because the tail can create enough space for the head constantly, it will not die.
- Second, we update the position of the snake step by step, cell by cell.

We can classify the states of the snake into three categories:

1.There are paths between food and head. And there are paths between head and tail.

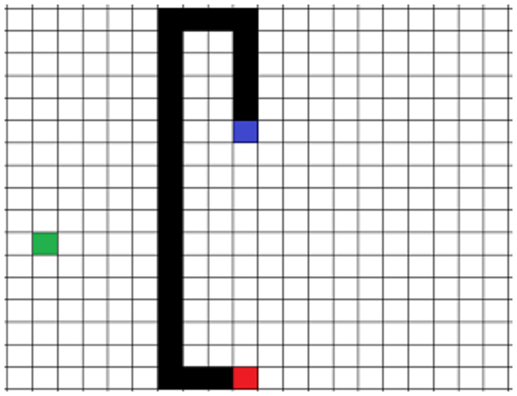
In this case, the snake can go directly to the food and chase the tail.



Case 1

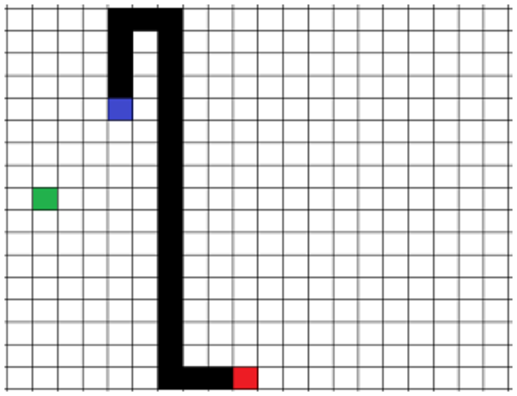
2.There are paths between head and tail. But no paths between head and food.

In this case, the snake cannot approach the tail, so we let it chase the tail.


Case 2

3.There is no paths neither between head and food nor between head and tail.

In this case, the head will choose an arbitrary direction until it goes case 2.


Case 3

Pseudocode:

```
if  Could eat food:
        if  The virtual snake follows the rule shortest to eat food, can find the tail:
               Real snake moves one more step;
               Judge again;
else if  The virtual snake goes along the irregular shortest to eat food to find the tail:
               Real snake moves one more step;
               Judge again;
else if The virtual know that I can reach my tail and move one step so that I can reach my tail:
               Choose to move farthest position from the food;
               Judge again;
Else:
               DFS moves one step to the deepest path
```

4. Work Distribution

Tianyang Cao

- Wrote the codes for the snake and game configuration class methods
- Wrote the markdown annotation (code tutorial) in Jupyter NoteBook

Yanmo Xu

- Wrote the codes for the snake smart path searching algorithm
- Organized/Integrated all the paper work, including this final PDF file

Yuhao chen

- Wrote the codes for the main game setup and dynamic loop
- Wrote the tutorial for improving the snake path searching algorithm