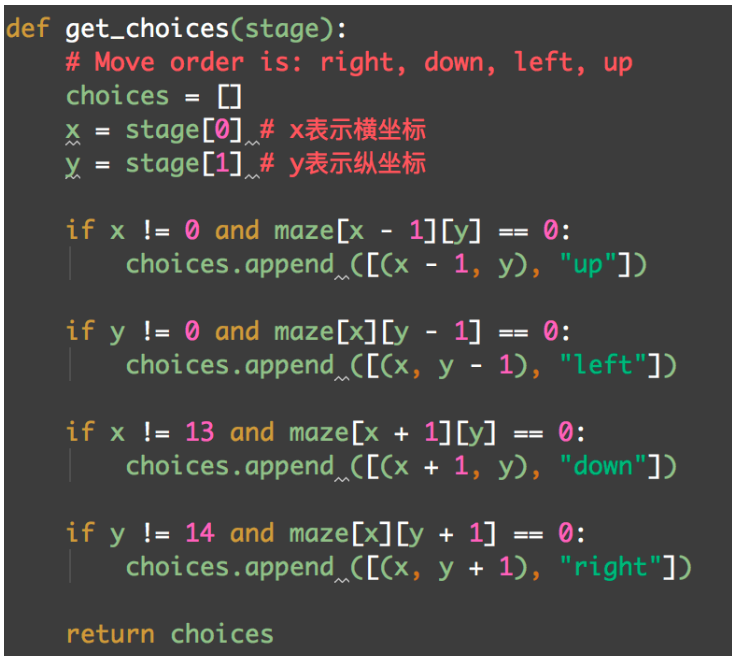
Lab 5 Report Template

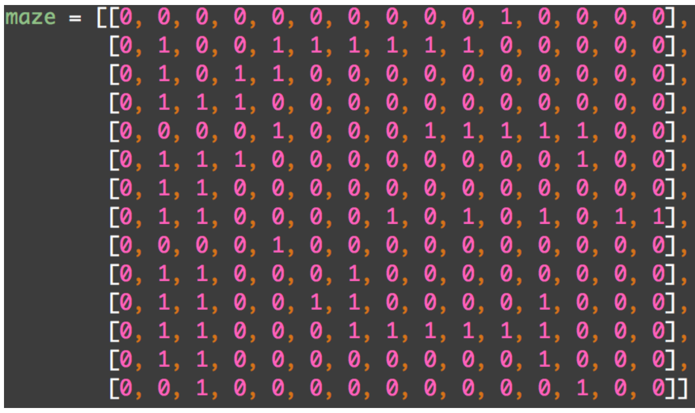
1. Use stack to solve “rate in a maze” as shown in the class

(1) To solve the “rate in a maze” problem, I use the stack to finish the depth first search. I use 2 stack to store information, the first one store all of the known states which are waiting for expanding and the other one store the path to current state during the process of searching.

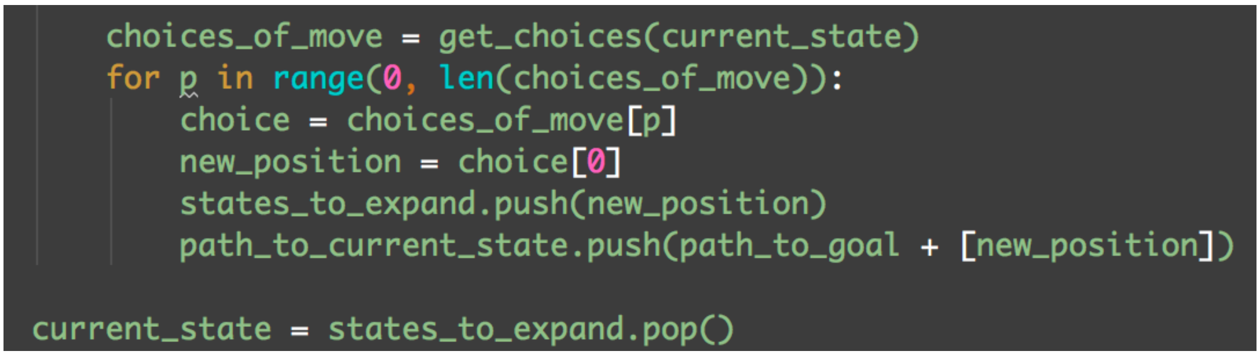
(2) For every state waiting for expanding, I use a function to get the choices of next states and they will be pushed into the stack in the order of “up, left, down, right” so that the top of the stack will in the order of “right, down, left, up” which is asked in the ppt.



```python
def get_choices(stage):
    # Move order is: right, down, left, up
    choices = []
    x = stage[0] # x表示横坐标
    y = stage[1] # y表示纵坐标

    if x != 0 and maze[x - 1][y] == 0:
        choices.append ([(x - 1, y), "up"])

    if y != 0 and maze[x][y - 1] == 0:
        choices.append ([(x, y - 1), "left"])

    if x != 13 and maze[x + 1][y] == 0:
        choices.append ([(x + 1, y), "down"])

    if y != 14 and maze[x][y + 1] == 0:
        choices.append ([(x, y + 1), "right"])

    return choices
```



```python
maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
        [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]]
```

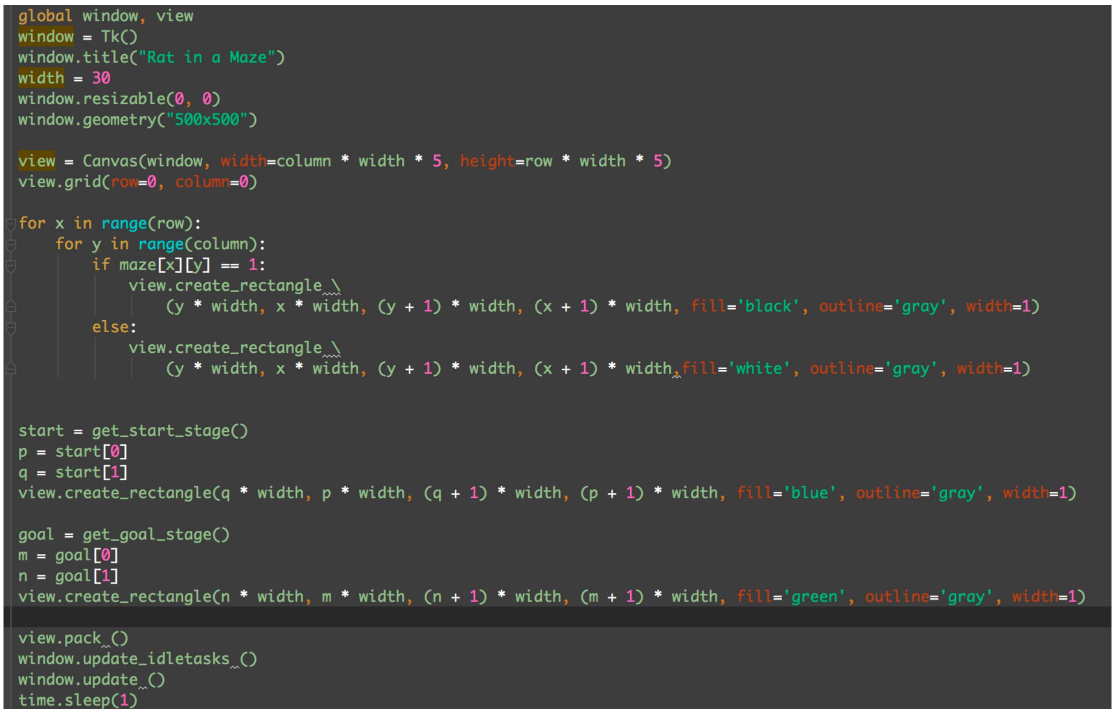


```python
        choices_of_move = get_choices(current_state)
        for p in range(0, len(choices_of_move)):
            choice = choices_of_move[p]
            new_position = choice[0]
            states_to_expand.push(new_position)
            path_to_current_state.push(path_to_goal + [new_position])

    current_state = states_to_expand.pop()
```

(3) Since I use the stack to store the choices for next step, the path of search will not change to another path until there is no choice for the current path. So the depth first search is finished.

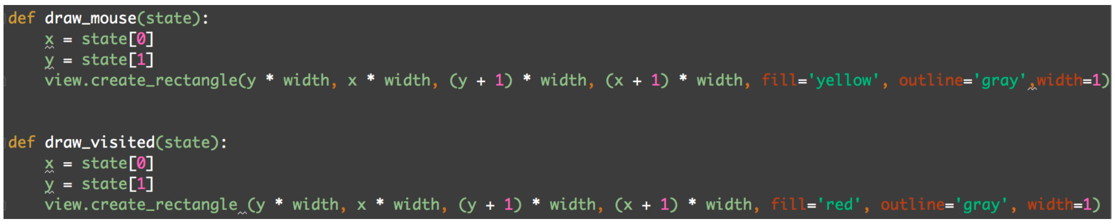
(4) Finally, I use the GUI to show the process of search.

This part shows the maze with start point and goal point.



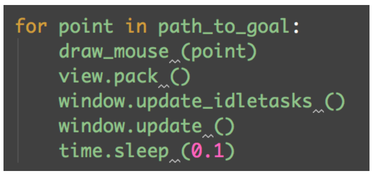
```
global window, view
window = Tk()
window.title("Rat in a Maze")
width = 30
window.resizable(0, 0)
window.geometry("500x500")

view = Canvas(window, width=column * width * 5, height=row * width * 5)
view.grid(row=0, column=0)

for x in range(row):
    for y in range(column):
        if maze[x][y] == 1:
            view.create_rectangle \
                (y * width, x * width, (y + 1) * width, (x + 1) * width, fill='black', outline='gray', width=1)
        else:
            view.create_rectangle \
                (y * width, x * width, (y + 1) * width, (x + 1) * width,fill='white', outline='gray', width=1)


start = get_start_stage()
p = start[0]
q = start[1]
view.create_rectangle(q * width, p * width, (q + 1) * width, (p + 1) * width, fill='blue', outline='gray', width=1)

goal = get_goal_stage()
m = goal[0]
n = goal[1]
view.create_rectangle(n * width, m * width, (n + 1) * width, (m + 1) * width, fill='green', outline='gray', width=1)

view.pack ()
window.update_idletasks ()
window.update ()
time.sleep(1)
```

During the search, 2 methods are used.



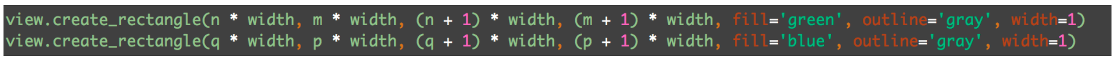
```
def draw_mouse(state):
    x = state[0]
    y = state[1]
    view.create_rectangle(y * width, x * width, (y + 1) * width, (x + 1) * width, fill='yellow', outline='gray',width=1)


def draw_visited(state):
    x = state[0]
    y = state[1]
    view.create_rectangle (y * width, x * width, (y + 1) * width, (x + 1) * width, fill='red', outline='gray', width=1)
```

Show the path found:



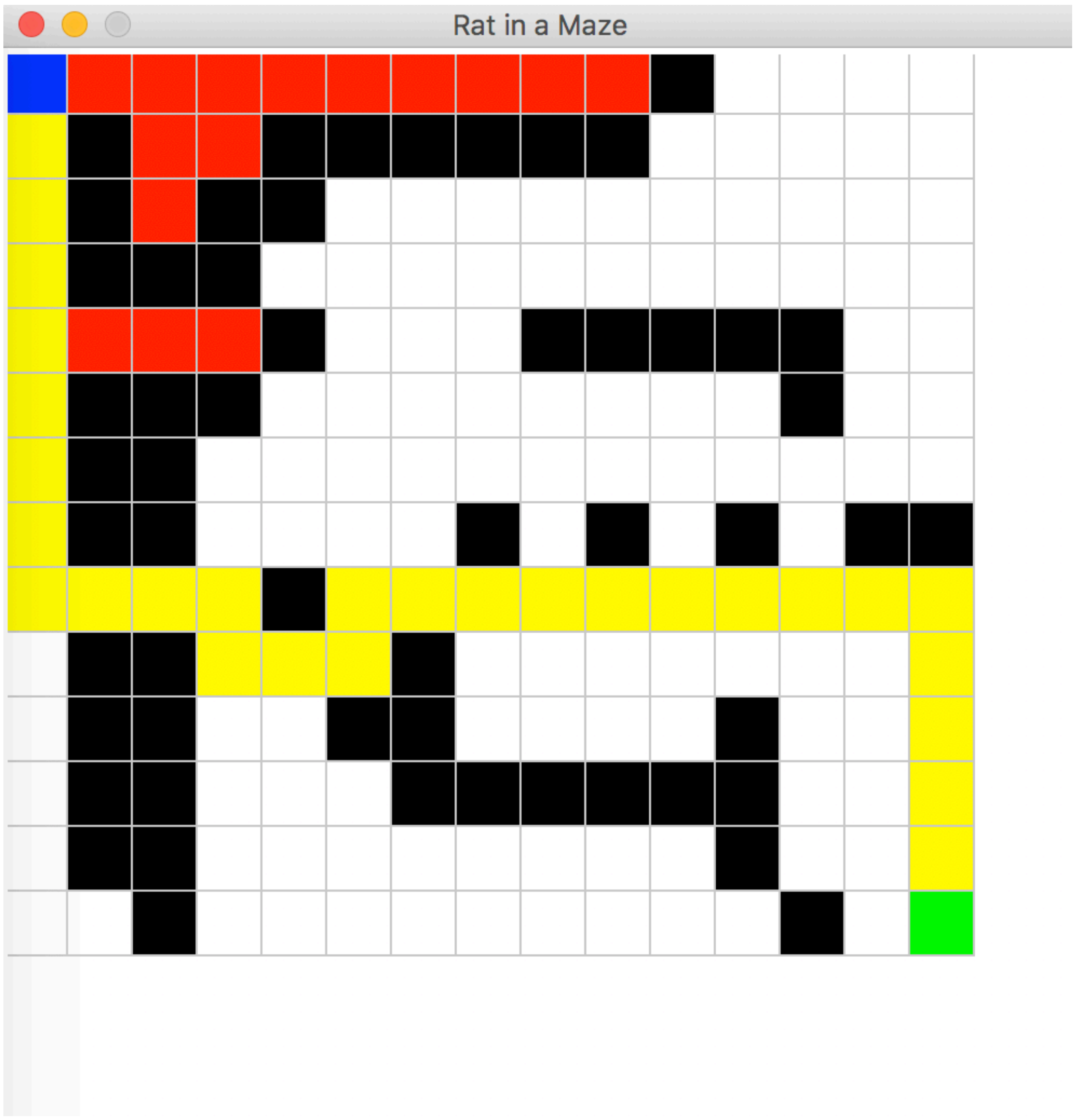
```
for point in path_to_goal:
    draw_mouse (point)
    view.pack ()
    window.update_idletasks ()
    window.update ()
    time.sleep (0.1)
```

At the end, drawing the start and end point.



```
view.create_rectangle(n * width, m * width, (n + 1) * width, (m + 1) * width, fill='green', outline='gray', width=1)
view.create_rectangle(q * width, p * width, (q + 1) * width, (p + 1) * width, fill='blue', outline='gray', width=1)
```

(5) The result of the GUI:



The black part means the block.

The white part means the unsearched available location.

The blue part means the start point.

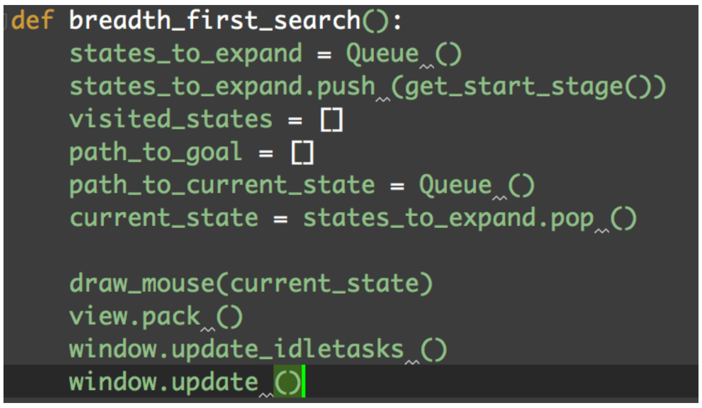
The green part means the goal point.

The yellow part means the current point and path to the goal.

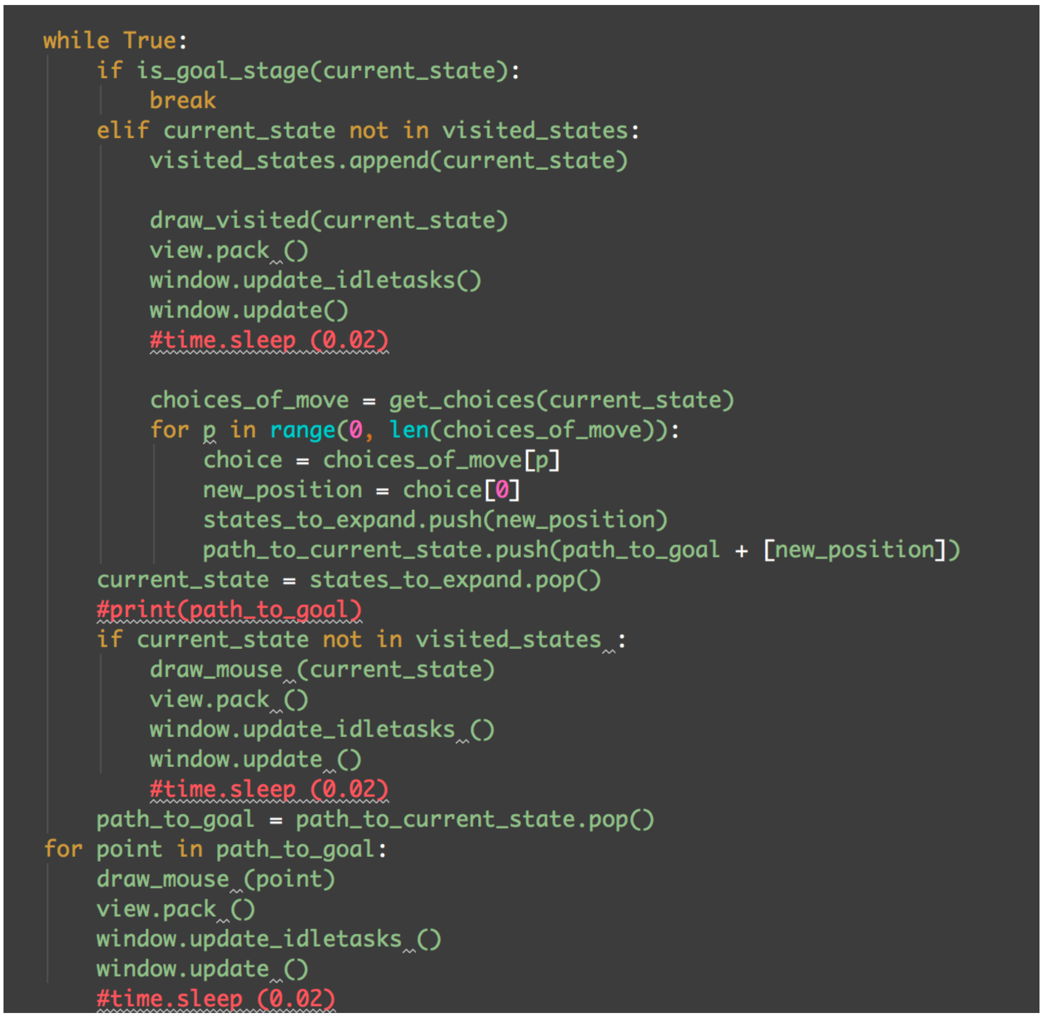
The red part means the searched points that not in the path to the goal.

2. Use queue to solve “wire routing” as shown in the class

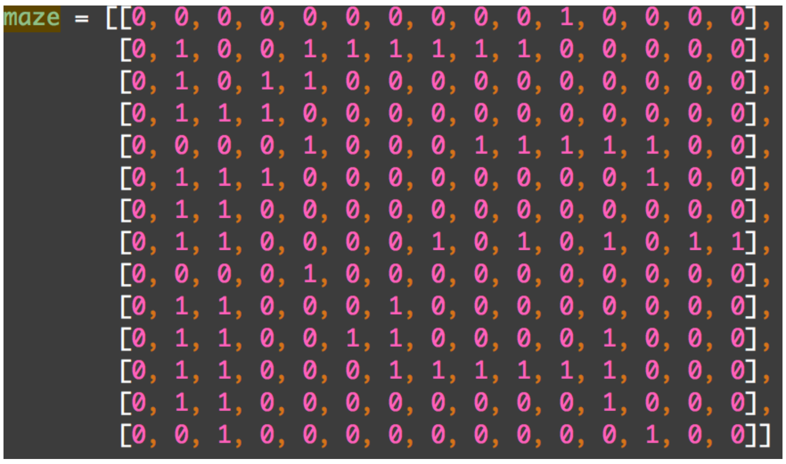
(1) In fact, this question ask me to finish the breadth first search, which has not big different with the DFS in the question1. I just change the 2 stack in the question1 into queue and let the value in the maze to store the distant form the position to the start position.



```
def breadth_first_search():
    states_to_expand = Queue ()
    states_to_expand.push (get_start_stage())
    visited_states = []
    path_to_goal = []
    path_to_current_state = Queue ()
    current_state = states_to_expand.pop ()

    draw_mouse(current_state)
    view.pack ()
    window.update_idletasks ()
    window.update ()
```

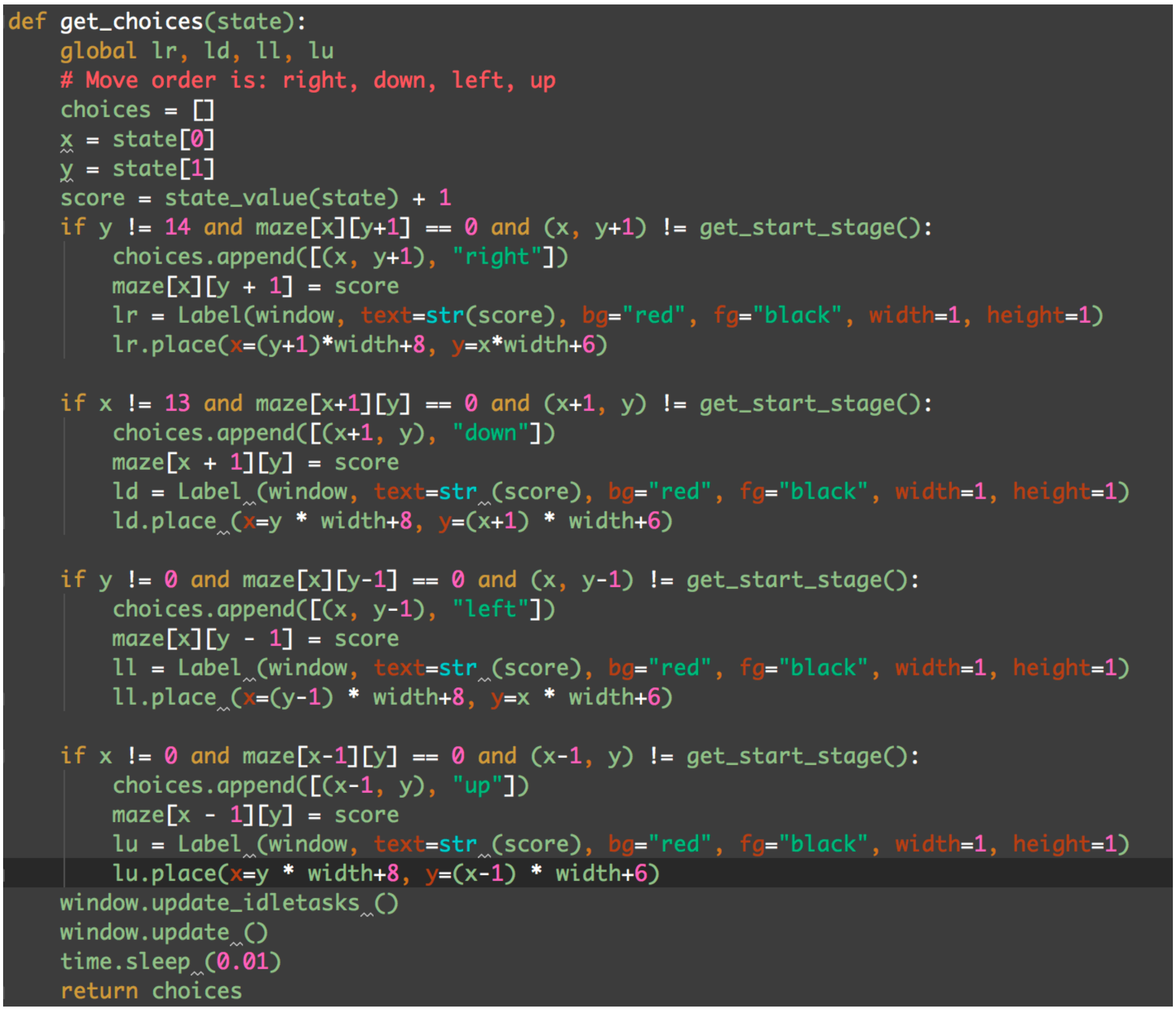


```
    while True:
        if is_goal_stage(current_state):
            break
        elif current_state not in visited_states:
            visited_states.append(current_state)

            draw_visited(current_state)
            view.pack ()
            window.update_idletasks()
            window.update()
            #time.sleep (0.02)

            choices_of_move = get_choices(current_state)
            for p in range(0, len(choices_of_move)):
                choice = choices_of_move[p]
                new_position = choice[0]
                states_to_expand.push(new_position)
                path_to_current_state.push(path_to_goal + [new_position])
        current_state = states_to_expand.pop()
        #print(path_to_goal)
        if current_state not in visited_states :
            draw_mouse (current_state)
            view.pack ()
            window.update_idletasks ()
            window.update ()
            #time.sleep (0.02)
        path_to_goal = path_to_current_state.pop()
    for point in path_to_goal:
        draw_mouse (point)
        view.pack ()
        window.update_idletasks ()
        window.update ()
        #time.sleep (0.02)
```



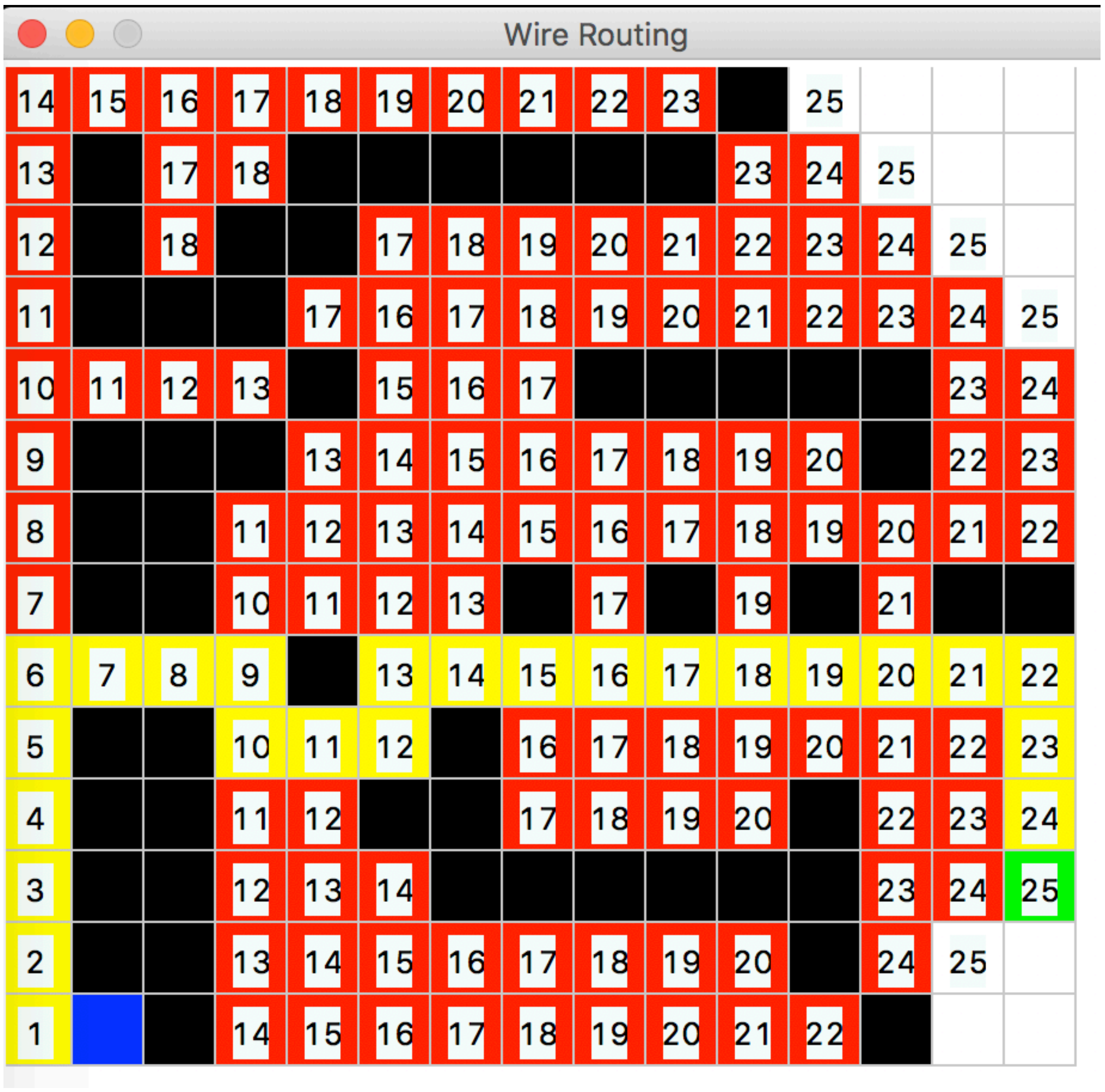
```python
maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
        [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]]
```

(2) Also, I change the method "get_choices". Then, it would give the distant from current state to the start state and draw the distant in the relevant location in the GUI.



```python
def get_choices(state):
    global lr, ld, ll, lu
    # Move order is: right, down, left, up
    choices = []
    x = state[0]
    y = state[1]
    score = state_value(state) + 1
    if y != 14 and maze[x][y+1] == 0 and (x, y+1) != get_start_stage():
        choices.append([(x, y+1), "right"])
        maze[x][y + 1] = score
        lr = Label(window, text=str(score), bg="red", fg="black", width=1, height=1)
        lr.place(x=(y+1)*width+8, y=x*width+6)

    if x != 13 and maze[x+1][y] == 0 and (x+1, y) != get_start_stage():
        choices.append([(x+1, y), "down"])
        maze[x + 1][y] = score
        ld = Label (window, text=str (score), bg="red", fg="black", width=1, height=1)
        ld.place (x=y * width+8, y=(x+1) * width+6)

    if y != 0 and maze[x][y-1] == 0 and (x, y-1) != get_start_stage():
        choices.append([(x, y-1), "left"])
        maze[x][y - 1] = score
        ll = Label (window, text=str (score), bg="red", fg="black", width=1, height=1)
        ll.place (x=(y-1) * width+8, y=x * width+6)

    if x != 0 and maze[x-1][y] == 0 and (x-1, y) != get_start_stage():
        choices.append([(x-1, y), "up"])
        maze[x - 1][y] = score
        lu = Label (window, text=str (score), bg="red", fg="black", width=1, height=1)
        lu.place(x=y * width+8, y=(x-1) * width+6)
    window.update_idletasks ()
    window.update ()
    time.sleep (0.01)
    return choices
```

(3) The result of the GUI:



14	15	16	17	18	19	20	21	22	23	⬛	25			
13	⬛	17	18	⬛	⬛	⬛	⬛	⬛	⬛	23	24	25		
12	⬛	18	⬛	⬛	17	18	19	20	21	22	23	24	25	
11	⬛	⬛	⬛	17	16	17	18	19	20	21	22	23	24	25
10	11	12	13	⬛	15	16	17	⬛	⬛	⬛	⬛	⬛	23	24
9	⬛	⬛	⬛	13	14	15	16	17	18	19	20	⬛	22	23
8	⬛	⬛	11	12	13	14	15	16	17	18	19	20	21	22
7	⬛	⬛	10	11	12	13	⬛	17	⬛	19	⬛	21	⬛	⬛
6	7	8	9	⬛	13	14	15	16	17	18	19	20	21	22
5	⬛	⬛	10	11	12	⬛	16	17	18	19	20	21	22	23
4	⬛	⬛	11	12	⬛	⬛	17	18	19	20	⬛	22	23	24
3	⬛	⬛	12	13	14	⬛	⬛	⬛	⬛	⬛	⬛	23	24	25
2	⬛	⬛	13	14	15	16	17	18	19	20	⬛	24	25	
1	🟦	⬛	14	15	16	17	18	19	20	21	22	⬛		

The black part means the block.

The white part means the unsearched available location.

The blue part means the start point.

The green part means the goal point.

The yellow part means the current point and path to the goal.

The red part means the searched points that not in the path to the goal.

The number means the distant to the start point.