Cairo University
Faculty of Computers and Artificial Intelligence

CS361

Artificial Intelligence

Project ID: (PID23867930)

2nd Semester 2020 Research

GAMIFICATION

Team Members

ID	Name	Email
20170369	Nada Nasser Al-Saeed	Nadaelazab123321@gmail.com
20170351	Farah Mohamed Osama	Farahmaghraby1999@gmail.com
20170352	Hussien Hossam Idris	Hussein.h.idris@gmail.com
20170345	Yousef Adel El-Sayed	Youssefadel92@gmail.com
20170363	Omar Ashraf Labib	Omarlabib.51237@gmail.com

Contents

Introduction

In all fields and in general life there's this trend of people who keep looking for ways to change how things were made to become, they look for ways to make things better, efficient and most importantly more enjoyable. Gamification has become a creative and interesting way to change up how we do things and how we approach everything. In the recent years Gamification had gained popularity and occupied the interest of many people and many studies has been made since. Gamification has become a creative way to change up routine and old ways of things.

Gamification is the process of applying game principles and components to non-game contexts. It can also be defined as the set of actions and properties used to solve problems using characteristics of game elements.

Gamification leverages people's natural desire for socializing, competition, mastery, learning, achievements. It adds a different look at failure, it takes it as just a learning process; to learn from past experience, improve, and make it better. People might just correlate gaming to fun, while fun is a part of the experience, gamification is based on working and making an effort to earn and gain the benefits of a certain task.

Background

Gamification elements

Game design elements make the building block of gamification applications, those include points, badges, leader boards and instant feedback mechanics.

Game elements used in gamification usually includes instant feedback. Instant positive feedback motivates us to do something and makes us feel good about completing a task. Feedbacks gives people incentives to perform better, change behaviors and learn.

Some of the Gamification elements are:

1. Points
2. Badges
3. Leaderboards

1. **Points**:

Points are the basic elements of various games and gamified applications. Points are typically rewarded for successful accomplishments and for completion of tasks in the gamification environment and they usually represent the players' progress numerically. Various point systems can be used, e.g. reputation points, redeemable points, experience points as different purposes for points can be used. Points allows player's in-game behavior to be measured providing continuous and instant feedback and as a reward.

2. Badges

Badges are considered to be the visual representation of achievements and they can be earned within the gamification environment. Badges confirm players' accomplishments, symbolizes their merits and visually show the level of accomplishing goals. Earning a badge can be dependent on achieving certain number of points in a certain domain, or can be earned to symbolize the completion of a task or achievement.

Figure 1: IBM's badge for the Digital Nation Africa educational programs

Badges can serve as goals to players, if the prerequisites for achieving a badge is known to players. Just like points badges provide feedback in that they tell players how they performed so far.

Badges can affect players' choices and behaviors as the player can choose certain routes and challenges so that they earn badges that are associated with them. In addition, badges cause social influences as badges symbolizes one's membership to a group of people who earned a particular patch especially if it is a hard to get badge.

3. Leaderboards

Leaderboard ranks players according to relative success, measuring their performance against a certain criterion. Like this leaderboards can determine who performs best in a certain activity and then can be used as competitive indicators of a player's progress against other players' progress.

Leaderboards have mixed opinions regarding its motivational potential. They can be effective motivators if few points are left to reach the next rank or goal, but serve as de-motivators of players find themselves at the bottom of the leaderboard. Competition of leaderboards can lead to social pressure to increase a player's performance and participation and thus lead to failure to motivate some players. However positive effects of competition are expected if competitors are relatively at the same level of performance.

Gamification Applications

Gamification has almost been applied to every aspect of life, it was used in various fields, e.g. education, work, Health, personal lives, politics and even more.

1- Gamification in work:

Gamification in work has been applied in an attempt to improve employee performance, productivity and mental health. In general gamification in works refers to integrating gaming concepts to already existing processes or information systems and it is used to get positive employee and organizational outcomes.

2- Gamification in education:

Education and training always had interest in gamification. The need to make education more interesting for people is always at rise. Game-based learning has been created with the intent to make education more engaging and relevant to current generations, there also has been signs that gamification is particularly motivational when it comes to dyslexic students in the educational domain.

Companies used games to educate their employees on rising technologies. Gamification is used in corporate training to motivate their employers to deploy what they learned in the training to their jobs.

3- Gamification in public Health:

Applications like *Fitocracy* and *Quentiq* use gamification to encourage their users to exercise and have a general healthier life. Users get rewarded with points varying with each exercise, and they gain levels based on points gained. They are also given quests to complete and gain badges when they achieve fitness goals. Public health researchers have found gamification positive impacts on self-management and controlling of mental disorders and chronic diseases.

A game that got viral world-wide a few years back is *Pokémon Go,* it used augmented reality to show creatures from the popular cartoon show Pokémon (our generation grew up on this cartoon) so that users would "catch" them. The show was heavily focused on the main characters trying to catch all the creatures with its slogan being "gotta catch 'em all", and it being part of everyone's childhood everyone was tempted to catch 'em all no matter where they were. The characters would show anywhere and would require you to travel walking distances from your current location to "catch" them. This game got so popular that everyone was out taking walks to catch the Pokémon characters, it was a secretly best exercise app back then.

Figure 2: Pokémon Go Application

Development of the described Game

System component

1- Game board

As shown in figure, to display the board of the game, we need an array to store its values and also we need to store the position of the tile. In our implementation, we used class ***game_board*** to store the needed information and the needed operations of the board.

game_board class has the following attributes:

1	-1	0
1	0	1
0	1	-1

Figure 3 - Game Board

1. board: 2D array that store all numbers in the board.
2. tile_position: integer value used to store the current position of the tile.
3. prev_tile_position: integer value used to store the previous position of the tile in the previous move, it helps in generating the number by the min user.

Operations in ***game_board***:-

1- **Constructor:** Initialize the information of the board using a 2d array (board), tile position and by default the previous tile position = 6.

```
def __init__(self, arr,tile):
    self.prev_tile_position = 6
    self.board = arr
    self.tile_position = tile
    self.row_index = self.tile_position//3
    self.col_index = self.tile_position%3
```

2- **print_game_board :** used to print all information of the board.

```
def print_game_board(self):
    for row in self.board:
        print(row)
    print("tile position = " , self.tile_position)
```

3- **can_move_up, can_move_down, can_move_right and can_move_left** : check if the tile can move to a specific direction using its position.
For example: if the tile at upper corner so its position will be equals to 0 so it can't move up or left. Figure 4 shows how we represent the tile_position in the board.

```python
# if tile position != 0,1,2
def can_move_up(self):
    if (self.tile_position != 0
        and self.tile_position!=1
        and self.tile_position!=2):
        return True
    else:
        return False

# if tile position != 6,7,8
def can_move_down(self):
    if (self.tile_position != 6
        and self.tile_position!=7
        and self.tile_position!=8):
        return True
    else:
        return False

# if tile position != 2,5,8
def can_move_right(self):
    if (self.tile_position % 3 != 2
        and self.tile_position % 3 != 5
        and self.tile_position % 3 != 8):
        return True
    else:
        return False

# if tile position != 0,3,6
def can_move_left(self):
    if (self.tile_position != 0
        and self.tile_position != 3
        and self.tile_position != 6):
        return True
    else:
        return False
```

0	1	2
3	4	5
6	7	8

Figure 4: Indexes used to represent the tile position

4- **move_up, move_down, move_right, move_left:**

Used to update the board info after moving, by update the tile position and change the value at the previous tile position to "?" to represent the position where the Min player will put (1, 0 or -1). In the following two figures an example of moving right.

0	1	2
3	4	5
6	7	8

Figure 5: Before any move, tile_position = 6

0	1	2
3	4	5
?	7	8

Figure 4: after moving right, tile_position = 7

```
    #tile_position -= 3
    def move_up(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position -= 3
        self.board[self.row_index-1][self.col_index] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'

    def move_down(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position += 3
        self.board[self.row_index+1][self.col_index] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'

    def move_right(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position += 1
        self.board[self.row_index][self.col_index+1] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'

    def move_left(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position -= 1
        self.board[self.row_index][self.col_index-1] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'
```

5- **set_at_prev_tile(num:int), get_value_at_tile():int:** 2 functions just to change the value at tile position and the previous position.

```
def set_at_prev_tile(self,num):
    row = self.prev_tile_position//3
    col = self.prev_tile_position%3
    self.board[row][col] = num

def get_value_at_tile(self):
    self.row_index = self.tile_position//3
    self.col_index = self.tile_position%3
    curr_score = self.board[self.row_index][self.col_index]
    return curr_score
```

2- Game state:

After each move from the Max or Min player a new state created for the game that shows the current board (data and tile) and the current score. We represent this state using ***game_state*** class.

game_state class has the following attributes:

1. max_score: integer value used to represent minimum level goal of the game.
2. max_depth: integer value used to represent maximum number of movements in the game.
3. current_score: integer value used to represent the score that the max user get till the current game state.
4. board: game_board object that store the info of the board at the current state.
5. parent_state: game_state object used to represent the previous game state.

Operations in ***game_state*** class:

1. **Constructor**: used to initialize all attributes of game_state object

```
def __init__(self, board , max_depth , max_score ,  curr_score):
    self.parent_state = None
    self.board = copy.deepcopy(board)
    self.max_depth = max_depth
    self.max_score = max_score
    self.current_score = curr_score
```

2. **print_game_state():** print the board info and the current score at this state

```
def print_game_state(self):
        self.board.print_game_board()
        print("current_score = " , self.current_score)
        print("------------------------------------")
```

3. **Isleaf:** check if the current state of the game is a leaf or not using the max depth and max score.
   Leaf state: is a state when the Max player exceeded the level goal or when the current depth (number of moves done till this state) is greater than or equal to the maximum number of movements.

```
def isleaf(self,current_depth):
        if self.current_score >= self.max_score or current_depth >= 
self.max_depth:
            return True
        else:
            return False
```

4. **get_max_children ()**: get all possible moves that max player can move, and store them in a list then return this list.
   Figure 6 displays a children of one game state when max player move.
   In the Current state the tile position equals to 7 so the tile can only move up, right and left. So for the current state there are 3 children have different score and different board but same parent.
   NOTE: Score at child state equals to [score at the parent state + the value at new tile position after moving], for example score at *child_up* in *figure 6* equals to [score at parent + 1] while in *child_right equals to* [score at parent + 0].

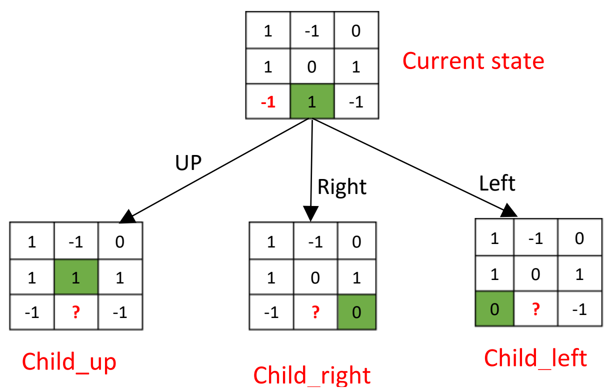


Figure 6: children Tree that represent all possible moves that max player can move

```
def get_max_children(self):
        parent = copy.deepcopy(self)

        children = []
        current_board = copy.deepcopy(self.board)

        if current_board.can_move_down():
            self.board.move_down()
            new_score = self.current_score + 
self.board.get_value_at_tile()
            child_down = create_game_state(self.board
,self.max_depth,self.max_score,new_score)
            child_down.parent_state = copy.deepcopy(parent)
            children.append(child_down)
        self.board = copy.deepcopy(current_board)
```

```
        if current_board.can_move_right():
            self.board.move_right()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_right = create_game_state(self.board
,self.max_depth,self.max_score,new_score)
            child_right.parent_state = copy.deepcopy(parent)
            children.append(child_right)
        self.board = copy.deepcopy(current_board)

        if current_board.can_move_left():
            self.board.move_left()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_left = create_game_state(self.board
,self.max_depth,self.max_score,new_score)
            child_left.parent_state = copy.deepcopy(parent)
            children.append(child_left)
        self.board = copy.deepcopy(current_board)

        if current_board.can_move_up():
            self.board.move_up()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_up =
create_game_state(self.board,self.max_depth,self.max_score,new_score)
            child_up.parent_state = copy.deepcopy(parent)
            children.append(child_up)
        self.board = copy.deepcopy(current_board)

        return children
```

5. **get_min_children() :** get all possible moves that min player generate a random number, store them in a list and return this list. The following figure displays a children of one game state when min player set a number from (0,1,-1).
Note: there is no change at score for min children.

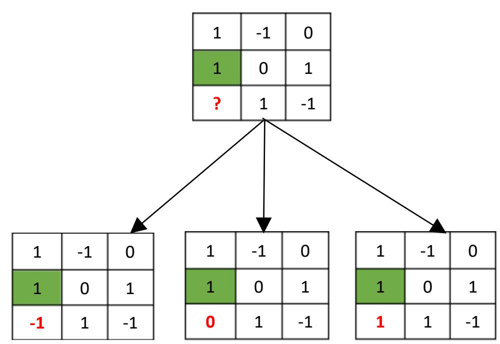


Figure 7: children Tree that represent all possible numbers that min player can generate in the previous tile position

```
def get_min_children(self):

        parent = copy.deepcopy(self)
        children = []
        current_board = copy.deepcopy(self.board)

        self.board.set_at_prev_tile(1)
        child1 =
create_game_state(self.board,self.max_depth,self.max_score,self.current_sco
re)
```

```
        child1.parent_state = copy.deepcopy(parent)
        children.append(child1)
        self.board = copy.deepcopy(current_board)

        self.board.set_at_prev_tile(0)
        child2 =
create_game_state(self.board,self.max_depth,self.max_score,self.current_sco
re)
        child2.parent_state = copy.deepcopy(parent)
        children.append(child2)
        self.board = copy.deepcopy(current_board)

        self.board.set_at_prev_tile(-1)
        child3 =
create_game_state(self.board,self.max_depth,self.max_score,self.current_sco
re)
        child3.parent_state = copy.deepcopy(parent)
        children.append(child3)
        self.board = copy.deepcopy(current_board)

        return children
```

6. **move_to_random_min_child() :** generate a random number in range of [-1,1] then use this number to create a new game state as a min state.

```
def move_to_random_min_child(self):
        value = random.randint(-1,1)
        parent = copy.deepcopy(self)
        self.board.set_at_prev_tile(value)
        self.parent_state = copy.deepcopy(parent)
```

7. **less_than_or_equal():** compare between the score at each state then return true if the current state less than or equal to state2 at the parameters

```
def less_than_or_equal(self,state2):
        score2 = state2.get_current_score()
        if self.current_score <= score2:
            return True
        else:
            return False
```

8. **get_original_state_from_leaf():** using the parent_state attribute, this function returns the first parent of a state (first move done to reach the current state). It is implemented as getting second node in a linked list from the last node.

```
def get_original_state_from_leaf(self):
        curr_state = copy.deepcopy(self)

        while curr_state.parent_state.parent_state != None:
            curr_state = copy.deepcopy(curr_state.parent_state)

        return curr_state
```

3- Alpha-beta Algorithm

Alpha beta burning algorithm, is a search technique that used to enhance the minimax algorithm be decreasing the expanded nodes in search tree.

Alpha-beta pseudocode:

```
Minimax (current_state, current_depth,isMaximizingPlayer, alpha , beta ):
    if current state is a leaf state:
       return current_state
    if isMaximizingPlayer:
       best_value = -INF
       for each child in children of current state:
          next_state = minimax(child,current_depth+1,False ,alpha ,beta))
           best_value = max (best_value, next_state)
           alpha = max (alpha, best_value)
           if beta<=(alpha):
               break
        return best_value
    else:
       best_value = INF
       for each child in children of current state:
           next_state =minimax(child, current_depth+1, True, alpha, beta)
           best_value = min (best_value,next_state)
           beta = min (beta,best_value))
           if beta <= (alpha):
              break
        return best_value
```

In our project we use alpha-beta algorithm to make the max player is an optimal player by using the algorithm to select the best moves while the min player selects the numbers in its turn randomly. The algorithm is implemented in the ***alpha_beta*** class.

alpha_beta class has no attributes, but it has three operations:

1. **minimax(current_state: game_state ,current_depth: int , isMaximizingPlayer: Boolean , alpha: game_state , beta: game_state) : game_state**

In this function we apply alpha-beta algorithm each time max player should move. It implements using DFS to traverse the searching tree. It returns the best final score the max player can gain. Then using (**get_original_state_from_leaf ()**) function in *game_state* class to get the current step max player should do –which is the first parent for final state-.

```
def minimax(self,current_state,current_depth,isMaximizingPlayer,alpha,beta ):
        if(current_state.isleaf(current_depth)):
            return current_state
        if isMaximizingPlayer:
            best_value = copy.deepcopy(create_inf_state(current_state ,
NEG_INF))
            children = copy.deepcopy(current_state.get_max_children())
            for child in children:
                next_state = copy.deepcopy(self.minimax(child,
current_depth+1, False , alpha , beta))
                best_value = copy.deepcopy(self.max_state(best_value ,
next_state))
                alpha = copy.deepcopy(self.max_state(alpha , best_value))
                if beta.less_than_or_equal(alpha):
                    break
            return best_value
        else:
            best_value = copy.deepcopy(create_inf_state(current_state , INF))
            children = copy.deepcopy(current_state.get_min_children())
            for child in children:
                next_state = copy.deepcopy(self.minimax(child,
current_depth,True , alpha , beta))
                best_value =
copy.deepcopy(self.min_state(best_value,next_state))
                beta = copy.deepcopy(self.min_state(beta,best_value))
                if beta.less_than_or_equal(alpha):
                    break
            return best_value
```

2. **max_state(state1:game_state , state2:game_state):game_state.** In this function we compare between two game_state objects using the current_score attribute and return the state that has the maximum score.

```
def max_state(self,state1 , state2):
       score1 = state1.get_current_score()
       score2 = state2.get_current_score()
       if score1 > score2:
           return state1
       else:
           return state2
```

3. **min_state(state1:game_state , state2:game_state):game_state** In this function we compare between two game_state objects using the current_score attribute and return the state that has the minimum score.

```
def min_state(self,state1 , state2):
        score1 = state1.get_current_score()
        score2 = state2.get_current_score()
        if score1 < score2:
            return state1
        else:
            return state2
```

4- Run the game

Rand_vs_ia_agent.py and GUI_game.py these two files used to run the game.

1. **rand_vs_ai_agent.py:**

Play (depth: int, score: int, player: win_player): game_state [] .This function used to alternate between min and max player turns. In max player turn we use **create_inf_state (current_state: game_state, INF: int): game_state** to create alpha and beta game states that have INF and –INF score. Then we use them to apply alpha-beta algorithm returning the best score can max player win with, then using **get_original_state_from_leaf ()** function in *game_state* class to get the new current state. In min player turn, current_state call move_to_random_min_child () to generate a random next state.

```
def play(depth,score,player):

    max_depth = depth
    max_score = score
    random_min = True
    state = intialize_game(max_depth,max_score)

    current_depth = 0
    MAX_PLAYER = True
    INF = 2147483648
    NEG_INF = -2147483648
    current_score = 0
    game = alpha_beta()   # constructor do nothig

    all_states = []
    all_states.append(copy.deepcopy(state))
    while True:
        current_score = state.get_current_score()

        max_player_win =
is_max_win(max_depth,max_score,current_depth,current_score)
```

```
        max_player_loss =
is_max_loss(max_depth,max_score,current_depth,current_score)
        if max_player_win:
            player.name = "MAX Player"
            break
        if  max_player_loss:
            player.name = "MIN Player"
            break
        if MAX_PLAYER:
            alpha = create_inf_state(state,NEG_INF)
            beta = create_inf_state(state,INF)
            state.set_parent(None) # root of the alpha beta tree
            best_state = game.minimax(state, 1, True, alpha, beta)
            state = best_state.get_original_state_from_leaf()
            all_states.append(copy.deepcopy(state))
            current_depth += 1
            MAX_PLAYER = False
        else:
            if random_min:
                state.move_to_random_min_child()
            else:
                alpha = create_inf_state(state,NEG_INF)
                beta = create_inf_state(state,INF)
                state.set_parent(None) # root of the alpha beta tree
                best_state = game.minimax(state, 1, False, alpha, beta)
                state = best_state.get_original_state_from_leaf()
            MAX_PLAYER = True
    return all_states
```

5- User Interface:

User Interface of the project implemented in file **GUI_game.py** using standard GUI Library called **Tkinter.** UI of our project has two frames one for control and read inputs of the game, the other is for showing game states one by one.

When we run **GUI_game.py** file, the control form in figure 6 appears so you can write your inputs in textboxes and press submit button. After pressing "submit", play() function called and returns all states result from your inputs in an array – This may take time due to recursion implementation in alpha beta algorithm and its high complexity.

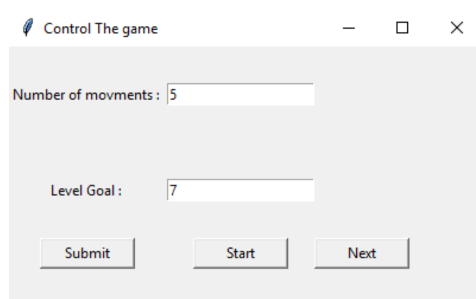


Figure 8: This Form to control the game and read inputs

After that you can start showing the states of the game using start and next buttons.

Code listing

gui_game.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed May 28 21:18:54 2020

"""
import tkinter
from rand_vs_ai_agent import play,intialize_game
from Game import game_board
from Game import game_state
from alpha_beta_algorithm import alpha_beta
from tkinter import messagebox

class states_list:
    def __init__(self,arr):
        self.states = arr
        self.index = 0
    def set_list(self,data):
        self.states = data
        self.index = 0
    def get_state(self):
        return self.states[self.index]
    def get_data(self):
        return self.states[self.index].board.board
    def inc_index(self):
        self.index+=1
    def end_game(self):
        if(self.index >= len(self.states)):
            return True
        else:
            return False


class player_win:
    name = "name"

player = player_win()
player.name = "none"

class board_cell:

    def __init__(self):
        self.cell = tkinter.Button()
        self.value = 0

    def setup_with_color(self , Window , num ,r , c,color):
        self.value = num
        self.cell = tkinter.Button(Window , width = 10 , height = 5 , text =
num , bg = color).grid(row=r,column=c)

    def setup(self , Window , num ,r , c):
        self.cell = tkinter.Button(Window , width = 10 , height = 5 , text =
num , bg="#c7c7c7").grid(row=r,column=c)
```

```
class game_form:
    def setup(self,arr,tile_pos,score):
        self.board = arr
        self.tile = tile_pos
        self.window = tkinter.Tk()
        self.window.title("Game Board")

        self.cells = [[board_cell(),board_cell(),board_cell()],
                      [board_cell(),board_cell(),board_cell()],
                      [board_cell(),board_cell(),board_cell()]]

        tile_row = tile_pos//3
        tile_col = tile_pos%3

        for i in range (0,3):
            for j in range(0,3):
                if tile_row == i and tile_col == j:
                    self.cells[i][j].setup_with_color(self.window
,arr[i][j],i,j,'green')
                else:
                    self.cells[i][j].setup(self.window ,arr[i][j],i,j)

        tkinter.Label(self.window, text='Score : ' , height = 5).grid(row=3 ,
column = 0)
        tkinter.Button(self.window ,width = 10 ,text = score , state =
"disable").grid(row=3, column = 1)

        self.window.geometry('%dx%d+%d+%d' % (240, 330, 50, 200))

    def draw( self):
        self.window.mainloop()

    def end(self):
        self.window.destroy()


arr = [[1 , -1, 0 ],[1 , 0 , 1 ],[0 , 1 ,-1]]

control_window =  tkinter.Tk()
control_window.title("Control The game")
control_window.geometry("400x220")
tkinter.Label(control_window, text='Number of movments : ' , height =
5).grid(row=0)
tkinter.Label(control_window, text='Level Goal : ' , height = 5).grid(row=1)
e1 = tkinter.Entry(control_window)
e2 = tkinter.Entry(control_window)
e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
f = game_form()

all_states = states_list(arr)

def submit_button():
    m = e1.get()
    s = e2.get()
```

```
    max_depth = int(m)
    max_score = int(s)
    all_states.set_list(play(max_depth,max_score,player))


def start_button():
    all_states.index = 0

f.setup(all_states.get_data(),all_states.get_state().get_tile_position(),all_
states.get_state().get_current_score())
    all_states.inc_index()

def next_button():
    if all_states.end_game():
        messagebox.showinfo("End of the Game", player.name +" Win")
    else:
        f.end()

f.setup(all_states.get_data(),all_states.get_state().get_tile_position(),all_
states.get_state().get_current_score())
        all_states.get_state().print_game_state()
        all_states.inc_index()
        f.draw()

tkinter.Button(control_window ,width = 10 ,text = "Start" , command =
start_button).grid(row=2, column = 1)
tkinter.Button(control_window ,width = 10 , text = "Next" , command =
next_button).grid(row=2 , column = 2)
tkinter.Button(control_window ,width = 10 , text = "Submit" , command =
submit_button).grid(row=2 , column = 0)


control_window.mainloop()
```

Board.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 24 03:17:56 2020

"""

class game_board:
    # board , tile_position , prev_tile_position
    prev_board = []
    def __init__(self, arr,tile):
        self.prev_tile_position = 6
        self.board = arr
        self.tile_position = tile
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3

    def print_game_board(self):
```

```
        for row in self.board:
            print(row)
        print("tile position = " , self.tile_position)

    # if tile position != 0,1,2
    def can_move_up(self):
        if (self.tile_position != 0
            and self.tile_position!=1
            and self.tile_position!=2):
            return True
        else:
            return False

    # if tile position != 6,7,8
    def can_move_down(self):
        if (self.tile_position != 6
            and self.tile_position!=7
            and self.tile_position!=8):
            return True
        else:
            return False

    # if tile position != 2,5,8
    def can_move_right(self):
        if (self.tile_position % 3 != 2
            and self.tile_position % 3 != 5
            and self.tile_position % 3 != 8):
            return True
        else:
            return False

    # if tile position != 0,3,6
    def can_move_left(self):
        if (self.tile_position != 0
            and self.tile_position != 3
            and self.tile_position != 6):
            return True
        else:
            return False

    #tile_position -= 3
    def move_up(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position -= 3
        self.board[self.row_index-1][self.col_index] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'

    def move_down(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position += 3
        self.board[self.row_index+1][self.col_index] +=
self.board[self.row_index][self.col_index]
```

```
        self.board[self.row_index][self.col_index] = '?'

    def move_right(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position += 1
        self.board[self.row_index][self.col_index+1] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'

    def move_left(self):
        self.prev_tile_position = self.tile_position
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        self.tile_position -= 1
        self.board[self.row_index][self.col_index-1] +=
self.board[self.row_index][self.col_index]
        self.board[self.row_index][self.col_index] = '?'

    def set_at_prev_tile(self,num):
        row = self.prev_tile_position//3
        col = self.prev_tile_position%3
        self.board[row][col] = num

    def get_value_at_tile(self):
        self.row_index = self.tile_position//3
        self.col_index = self.tile_position%3
        curr_score = self.board[self.row_index][self.col_index]
        return curr_score
```

Game.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 24 00:35:49 2020

"""
import copy
from Board import game_board
import random

class game_state:
#    board , max_depth , max_score
    def __init__(self, board , max_depth , max_score , curr_score):
        self.parent_state = None
        self.board = copy.deepcopy(board)
        self.max_depth = max_depth
        self.max_score = max_score
        self.current_score = curr_score

    def isleaf(self,current_depth):
```

```
        if self.current_score >= self.max_score or current_depth >=
self.max_depth:
            return True
        else:
            return False

    def print_game_state(self):
        self.board.print_game_board()
        print("current_score = " , self.current_score)
        print("-------------------------------------")

    def get_current_score(self):
        return self.current_score

    def get_tile_position(self):
        return self.board.tile_position

    def get_max_children(self):
        parent = copy.deepcopy(self)

        children = []
        current_board = copy.deepcopy(self.board)

        if current_board.can_move_down():
            self.board.move_down()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_down = create_game_state(self.board
,self.max_depth,self.max_score,new_score)
            child_down.parent_state = copy.deepcopy(parent)
            children.append(child_down)
        self.board = copy.deepcopy(current_board)

        if current_board.can_move_right():
            self.board.move_right()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_right = create_game_state(self.board
,self.max_depth,self.max_score,new_score)
            child_right.parent_state = copy.deepcopy(parent)
            children.append(child_right)
        self.board = copy.deepcopy(current_board)

        if current_board.can_move_left():
            self.board.move_left()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_left = create_game_state(self.board
,self.max_depth,self.max_score,new_score)
            child_left.parent_state = copy.deepcopy(parent)
            children.append(child_left)
        self.board = copy.deepcopy(current_board)

        if current_board.can_move_up():
            self.board.move_up()
            new_score = self.current_score + self.board.get_value_at_tile()
            child_up =
create_game_state(self.board,self.max_depth,self.max_score,new_score)
            child_up.parent_state = copy.deepcopy(parent)
            children.append(child_up)
```

```
        self.board = copy.deepcopy(current_board)

        return children


    def get_min_children(self):

        parent = copy.deepcopy(self)
        children = []
        current_board = copy.deepcopy(self.board)

        self.board.set_at_prev_tile(1)
        child1 =
create_game_state(self.board,self.max_depth,self.max_score,self.current_score
)
        child1.parent_state = copy.deepcopy(parent)
        children.append(child1)
        self.board = copy.deepcopy(current_board)

        self.board.set_at_prev_tile(0)
        child2 =
create_game_state(self.board,self.max_depth,self.max_score,self.current_score
)
        child2.parent_state = copy.deepcopy(parent)
        children.append(child2)
        self.board = copy.deepcopy(current_board)

        self.board.set_at_prev_tile(-1)
        child3 =
create_game_state(self.board,self.max_depth,self.max_score,self.current_score
)
        child3.parent_state = copy.deepcopy(parent)
        children.append(child3)
        self.board = copy.deepcopy(current_board)

        return children

    def move_to_random_min_child(self):
        value = random.randint(-1,1)
        parent = copy.deepcopy(self)
        self.board.set_at_prev_tile(value)
        self.parent_state = copy.deepcopy(parent)

    def less_than_or_equal(self,state2):
        score2 = state2.get_current_score()
        if self.current_score <= score2:
            return True
        else:
            return False

    def set_parent(self,parent):
        self.parent_state = parent

    def get_original_state_from_leaf(self):
        curr_state = copy.deepcopy(self)

        while curr_state.parent_state.parent_state != None:
```

```
            curr_state = copy.deepcopy(curr_state.parent_state)

        return curr_state

    def key(self):
        return self.current_score

def create_game_state(board,max_depth,max_score,current_score):
    state =
game_state(copy.deepcopy(board),max_depth,max_score,current_score)
    return state
```

---

alpha_beta_algorithm.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 24 03:51:20 2020

"""

from Game import game_state
from Board import game_board
import copy

INF = 2147483648
NEG_INF = -2147483648

class alpha_beta:

    def minimax(self,current_state ,current_depth ,
                isMaximizingPlayer , alpha , beta ):

        if(current_state.isleaf(current_depth)):
            return current_state

        if isMaximizingPlayer:
            best_value = copy.deepcopy(create_inf_state(current_state ,
NEG_INF))
            children = copy.deepcopy(current_state.get_max_children())

            for child in children:
                next_state = copy.deepcopy(self.minimax(child,
current_depth+1, False , alpha , beta))

                best_value = copy.deepcopy(self.max_state(best_value ,
next_state))

                alpha = copy.deepcopy(self.max_state(alpha , best_value))

                if beta.less_than_or_equal(alpha):
                    break
            return best_value
```

```
        else:
            best_value = copy.deepcopy(create_inf_state(current_state , INF))
            children = copy.deepcopy(current_state.get_min_children())

            for child in children:
                next_state = copy.deepcopy(self.minimax(child,
current_depth,True , alpha , beta))

                best_value =
copy.deepcopy(self.min_state(best_value,next_state))

                beta = copy.deepcopy(self.min_state(beta,best_value))
                if beta.less_than_or_equal(alpha):
                    break
            return best_value

    def max_state(self,state1 , state2):
        score1 = state1.get_current_score()
        score2 = state2.get_current_score()
        if score1 > score2:
            return state1
        else:
            return state2

    def min_state(self,state1 , state2):
        score1 = state1.get_current_score()
        score2 = state2.get_current_score()
        if score1 < score2:
            return state1
        else:
            return state2


def create_inf_state(current_state , inf):
    inf_arr = [[inf,inf,inf] , [inf,inf,inf] , [inf,inf,inf]]
    inf_board = game_board(inf_arr,current_state.get_tile_position())
    inf_state =
game_state(inf_board,current_state.max_depth,current_state.max_score,inf)
    return inf_state

def key(obj):
    return obj.key()

def sort_max_children(children):
    children.sort(key = key , reverse=True)

def sort_min_children(children):
    children.sort(key = key)
```

---

rand_vs_ai_agent.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 24 00:46:40 2020

"""
from Game import game_board
from Game import game_state
from alpha_beta_algorithm import alpha_beta
import copy

class player_win:
    name = "name"

player = player_win()
player.name = "none"


def play(depth,score,player):

    max_depth = depth
    max_score = score
    random_min = True
    state = intialize_game(max_depth,max_score)

    current_depth = 0
    MAX_PLAYER = True
    INF = 2147483648
    NEG_INF = -2147483648
    current_score = 0
    game = alpha_beta()  # constructor do nothig

    all_states = []
    all_states.append(copy.deepcopy(state))
    while True:
        current_score = state.get_current_score()

        max_player_win = 
is_max_win(max_depth,max_score,current_depth,current_score)
        max_player_loss = 
is_max_loss(max_depth,max_score,current_depth,current_score)


        if max_player_win:
            player.name = "MAX Player"
#            print("\n\nMAX WIN")
            break

        if  max_player_loss:
            player.name = "MIN Player"
#            print("MIN Win")
            break



        if MAX_PLAYER:
```

```
#             print("MAX : ")
            alpha = create_inf_state(state,NEG_INF)
            beta = create_inf_state(state,INF)

            state.set_parent(None) # root of the alpha beta tree
            best_state = game.minimax(state, 1, True, alpha, beta)

            state = best_state.get_original_state_from_leaf()

            all_states.append(copy.deepcopy(state))

            current_depth += 1

            MAX_PLAYER = False

#             state.print_game_state()

        else:
            if random_min:
                state.move_to_random_min_child()
            else:
                alpha = create_inf_state(state,NEG_INF)
                beta = create_inf_state(state,INF)

                state.set_parent(None) # root of the alpha beta tree
                best_state = game.minimax(state, 1, False, alpha, beta)

                state = best_state.get_original_state_from_leaf()

            MAX_PLAYER = True

    return all_states

##############################################################################

def create_inf_state(current_state , inf):
    inf_arr = [[inf,inf,inf] , [inf,inf,inf] , [inf,inf,inf]]
    inf_board = game_board(inf_arr,current_state.get_tile_position())
    inf_state =
game_state(inf_board,current_state.max_depth,current_state.max_score,inf)

    return inf_state

def is_max_win(max_depth,max_score,current_depth,current_score):
    if (current_depth <= max_depth and current_score >= max_score):
        return True
    else:
        return False

def is_max_loss(max_depth,max_score,current_depth,current_score):
    if (current_depth >= max_depth and current_score < max_score):
        return True
    else:
        return False

def intialize_game(max_depth,max_score):
    board_values = [[1 , -1, 0 ]
```

```
                 ,[1 , 0 , 1 ],
                  [0 , 1 ,-1]]
    board = game_board(board_values,6)
    state = game_state(board,max_depth,max_score,0)
    return state
```

Test cases (output of the game play)

To run the game you need to follow these steps:

1- Put all files in same folder
2- Run “GUI_game.py” file
3- UI appears so you can write your inputs
4- Submit the inputs via “Submit button” then press “Start button” to see the game states one by one using “Next button”.

NOTE: after pressing submit it may take time to be able to press “start button” because of the high complexity of alpha-beta and it also depends on ‘max number of movements’ you entered in the game.

<u>Test case #1:</u>

Input:

1- Minimal Level Goal (score) = 7
2- Maximum number of moves = 5

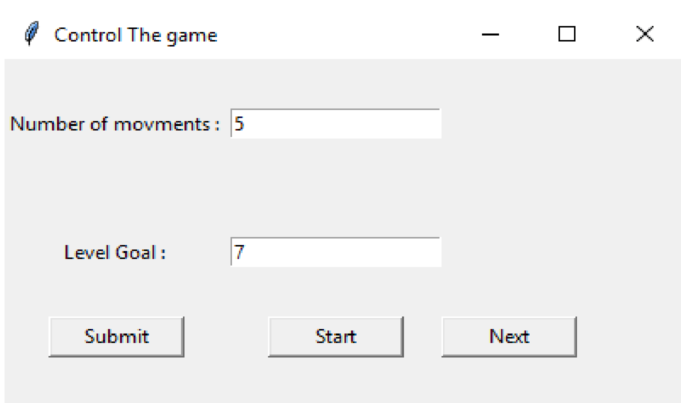


Figure 8: Inputs in GUI form

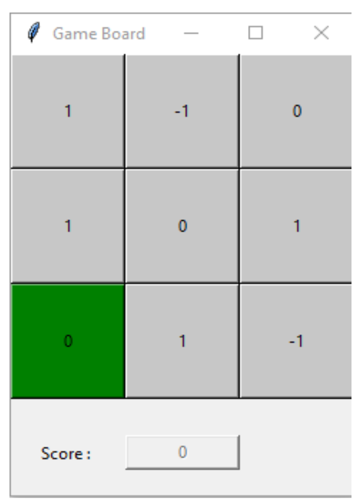


Figure 9 : Initial state of the game

Output

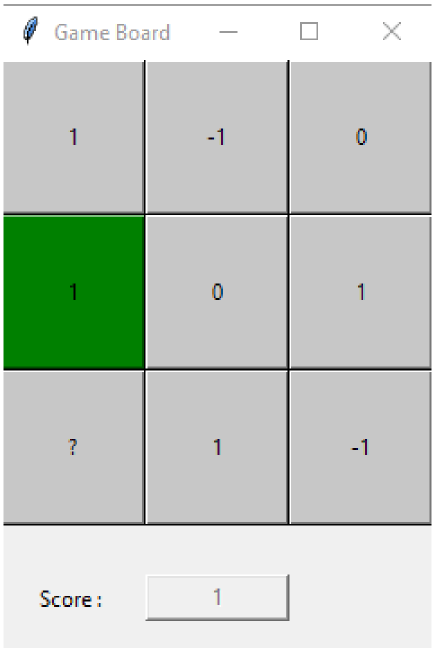


Figure 1: max user moved up with score =1

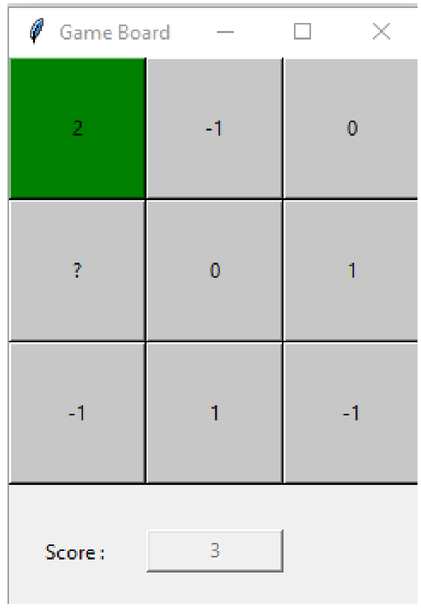


Figure 2 Min player generate -1 at previous tile position and max player moved up again. Score

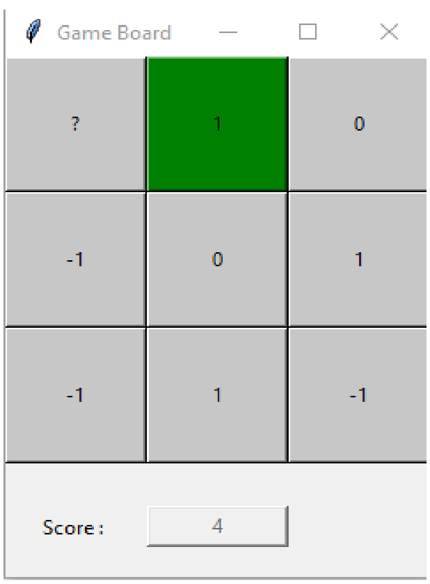


Figure3: Min player generated -1 at previous tile position and max player moved right.

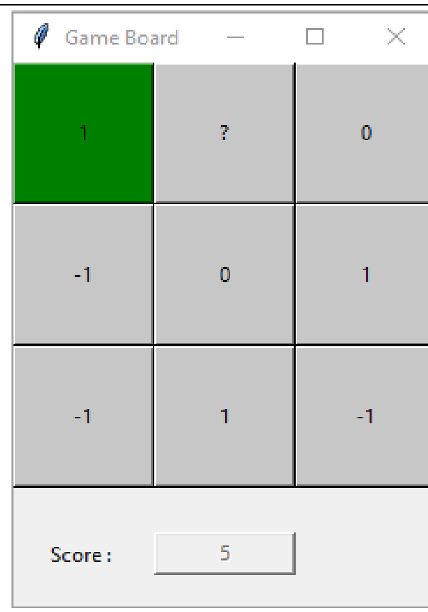


Figure 4 Min Player generated 1 at previous tile position, then Max Player moved left

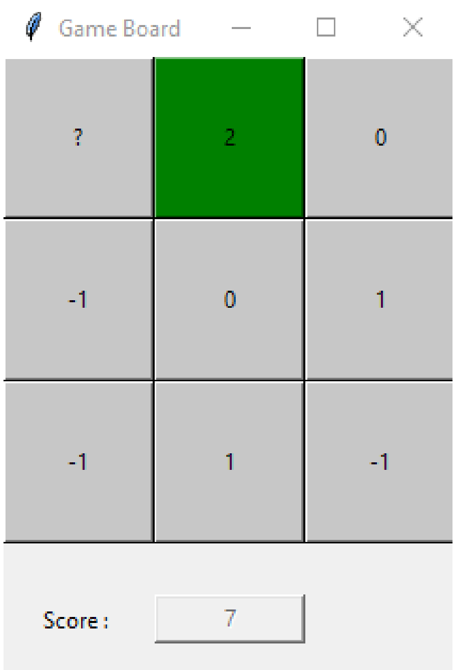


Figure 5: Min player generated 1 at previous tile position, then Max Player moved up.

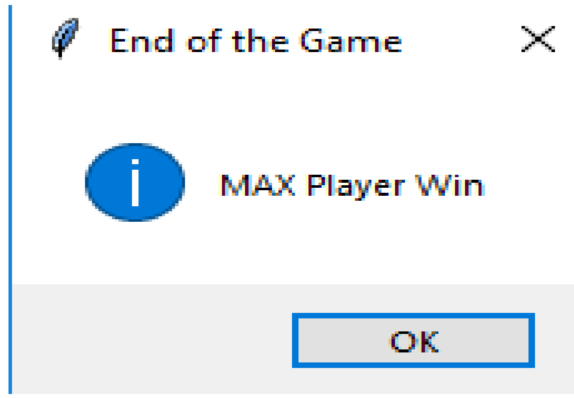


Figure 6: Max Player get Score equal 7 in 5 steps.

Test Case #2

Input:

1. Minimal Level Goal (score) = 10
2. Maximum number of moves = 3

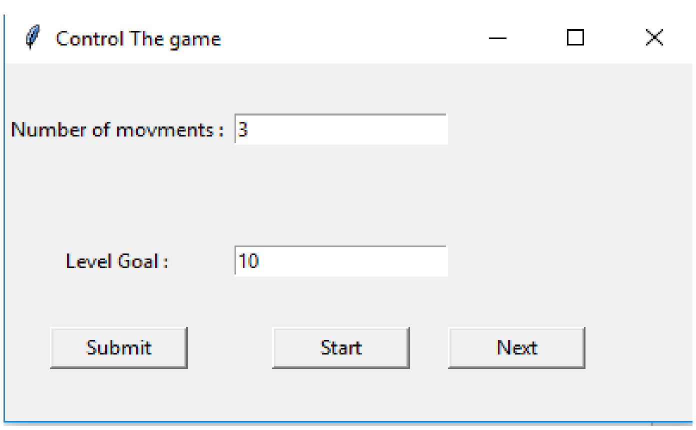


Figure 10: Inputs in GUI

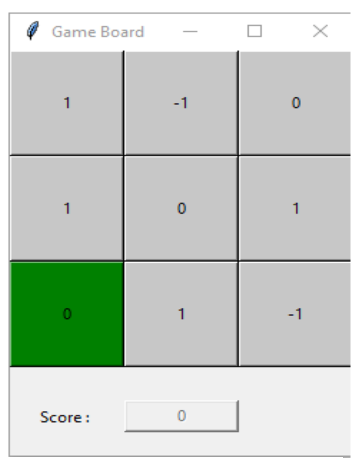
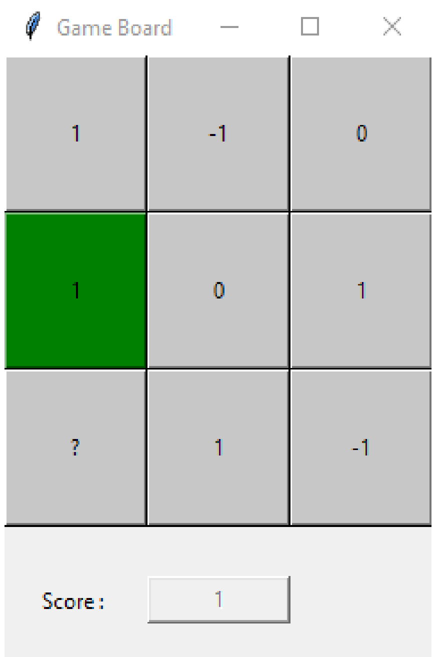
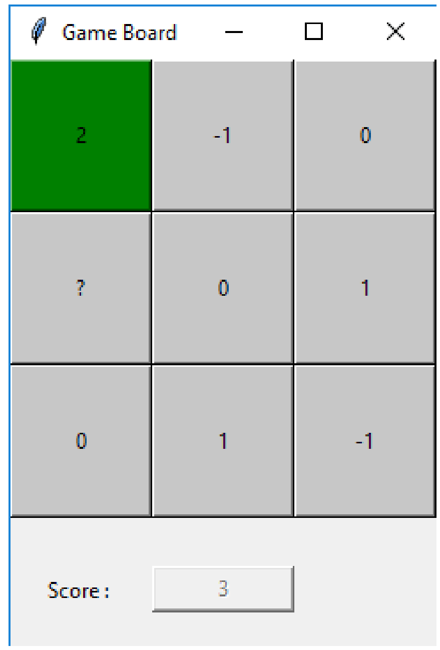


Figure 11: Initial state of the game

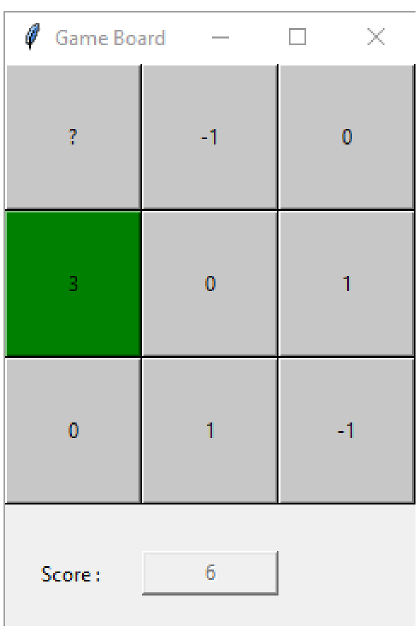
Output



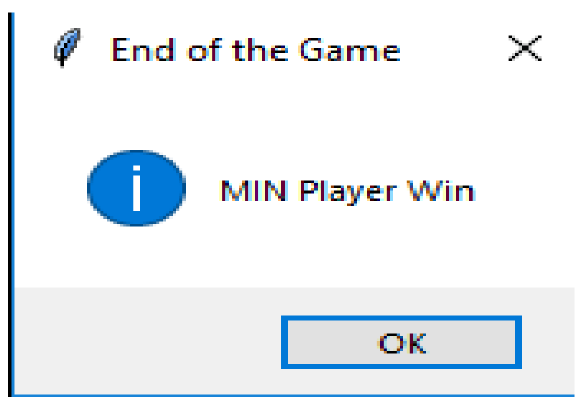
Max Player Move up



Min player generate 1 at previous tile position, then max player moved up again.



Min player generated 1 at previous tile position, then max player moved down.



Max Player cannot get score 10 in 3 steps, so Min player win

Test Case #3

Input:

1. Minimal Level Goal (score) = 7
2. Maximum number of moves = 5

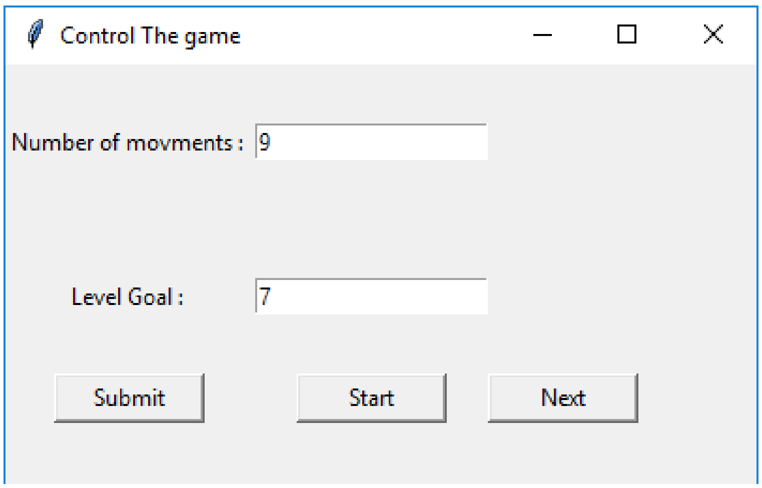


Figure 13: Inputs in GUI form

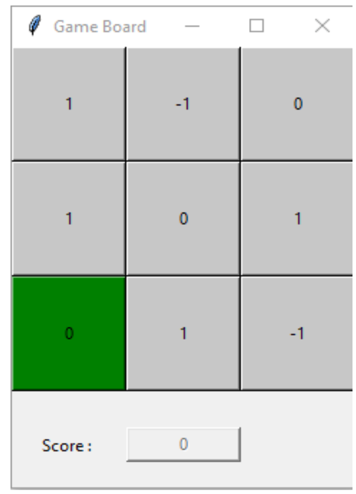
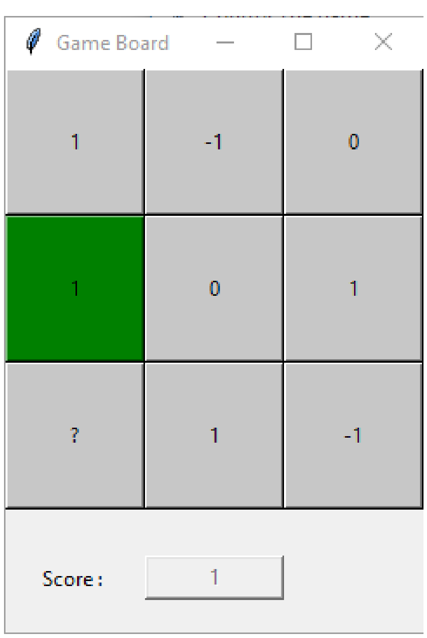
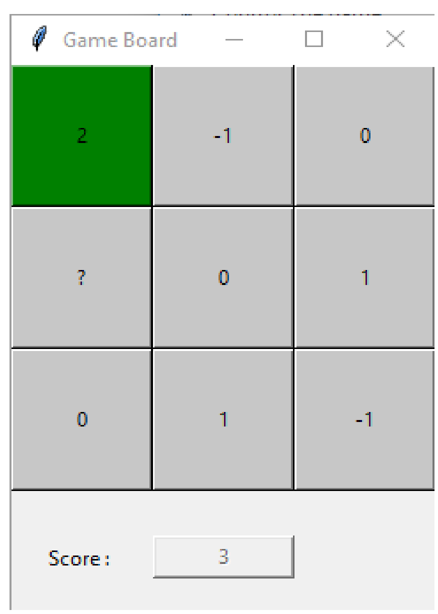


Figure 13 : Initial state of the game

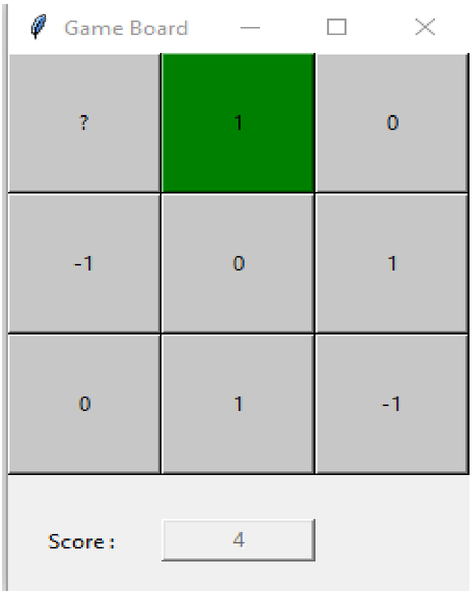
Output



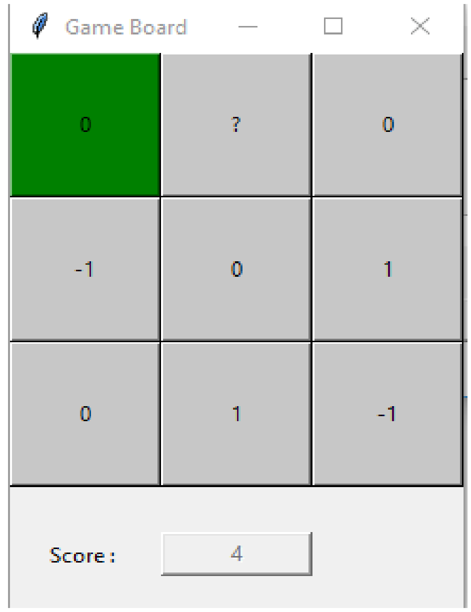
Max Player Move up



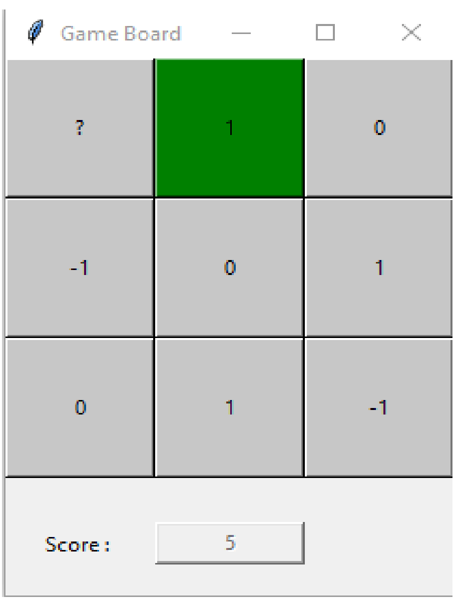
Min player generate 0 at previous tile position, then max player moved up again.



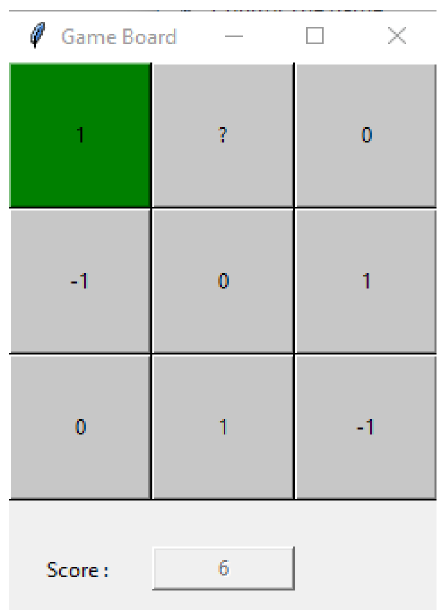
Min player generate -1 at previous tile position, then max player moved right.



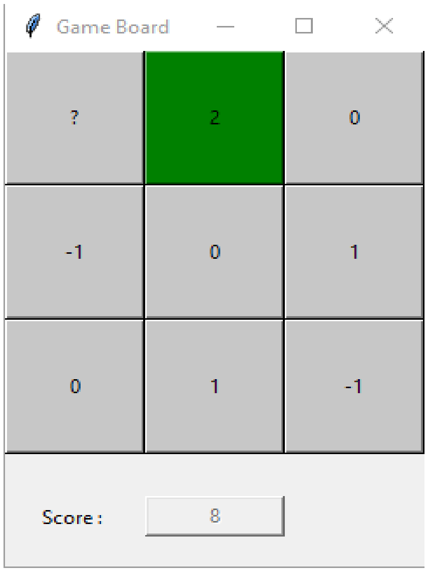
Min player generate -1 at previous tile position, then max player moved left



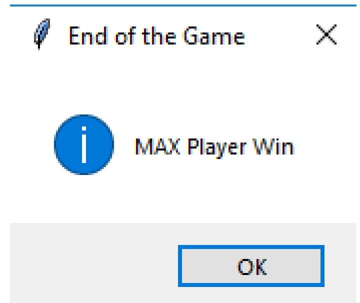
Min player generate 1 at previous tile position, then max player moved right



Min player generate 0 at previous tile position, then max player moved left



Min player generate 1 at previous tile position, then max player moved right.



Max Player get Score equals 8 in 7 steps.

Test Case #4

Input:

1- Minimal Level Goal (score) = 7

2- Maximum number of moves = 5

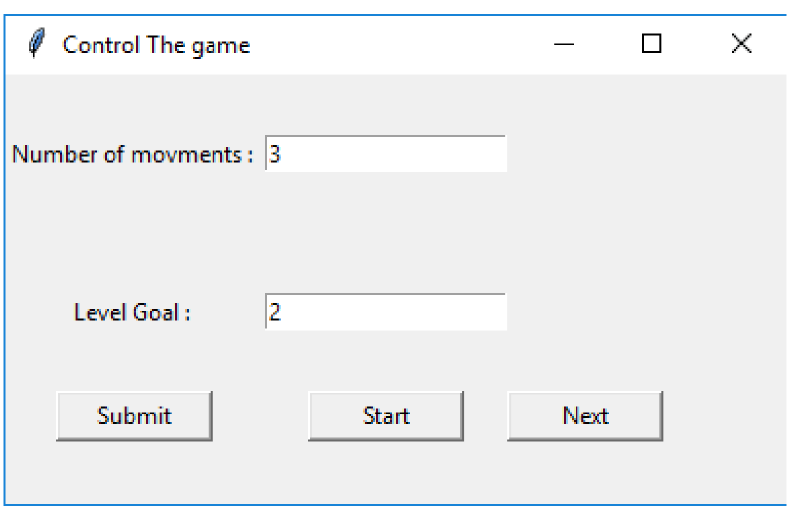


Figure 15: Inputs in GUI form

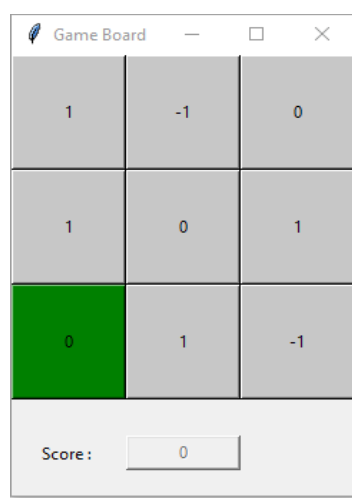
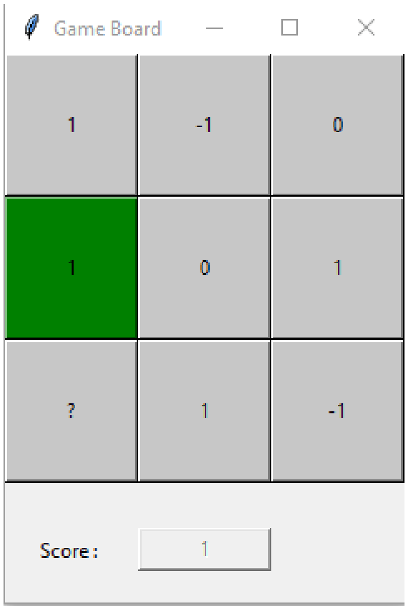
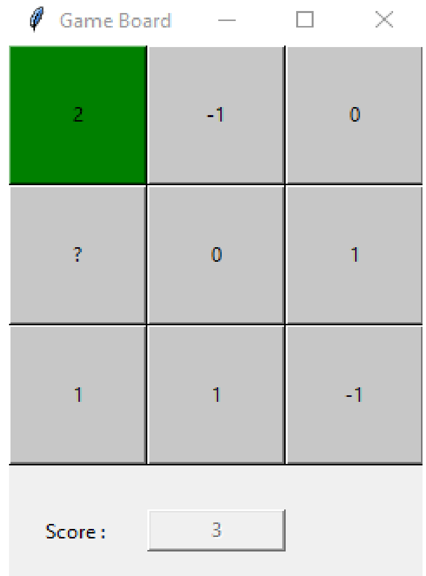


Figure 15 : Initial state of the game

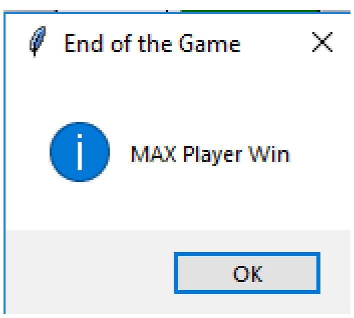
Output



Max Player Move up



Min player generate 1 at previous tile position, then max player moved up again.



Max Player get Score equal 3 in 2 steps.

References

[1] M. Guta, “What is Gamification and How Can It Help My Business?,” *Small Business Trends*, 07-Jul-2017. [Online]. Available: https://smallbiztrends.com/2017/07/what-is-gamification.html

[2] Y. A. R. O. S. L. A. V. LEHENCHUK, “Get the Most in Life and Work with Gamification,” Product Tribe, 07-Sep-2019. [Online]. Available: https://producttribe.com/productivity/life-gamification

[3] TechnologyAdvice, “Gamification in work space,” 31-Oct-2014. [Online]. Available: https://www.youtube.com/watch?time_continue=44&v=vInCVUIiq6g&feature=emb_title

[4] Mark Rober, “The Super Mario effects,” TEDx Talks, 31-May-2018. [Online]. Available: https://www.youtube.com/watch?v=9vJRopau0g0

[5] Yu-Kai Chou, “Gamification to improve our work” TEDx Talks, 26-Feb-2014. [Online]. Available: https://www.youtube.com/watch?v=v5Qjuegtiyc