RIGA TECHNICAL UNIVERSITY
FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
INSTITUTE OF APPLIED COMPUTER SYSTEMS

Practical assignment #1

Fundamentals of Artificial Intelligence

Game Programming

Git Repository

Author: Balasuriyage Aritha Dewnith KUMARASINGHE

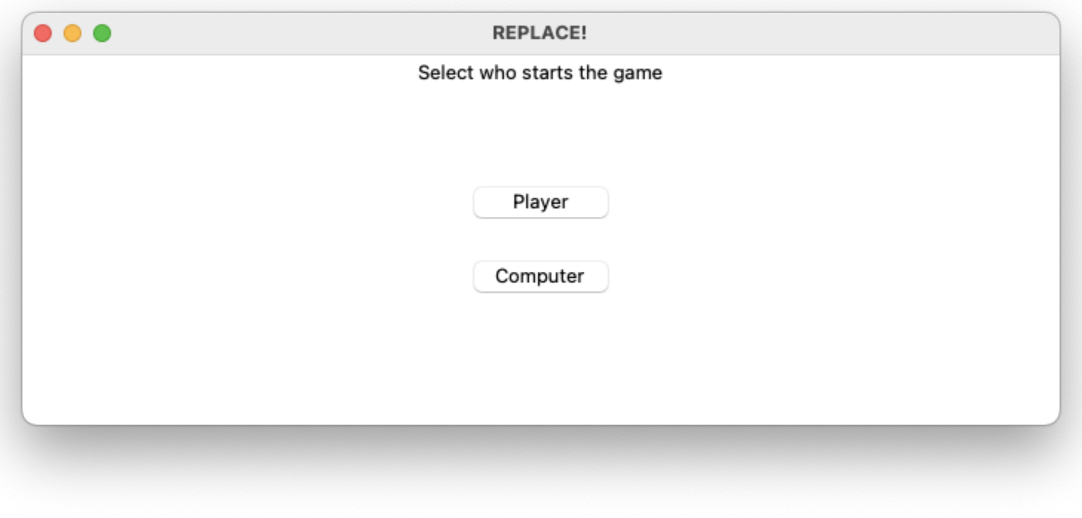
Student ID Number: 203AEB014

2021 / 2022 study year

Table of Contents

User Manual

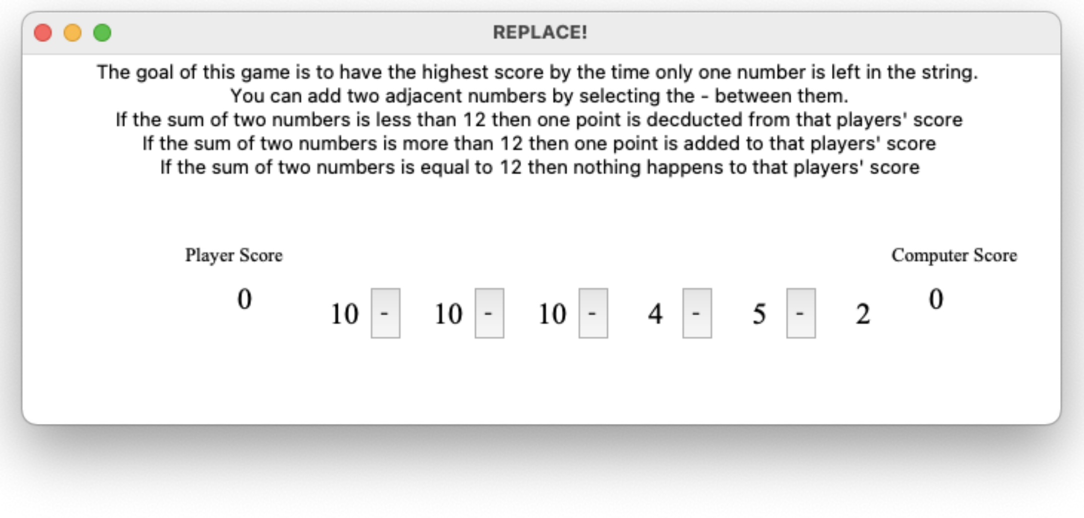
1. When you start the game, you should see a screen like this:



This is where you select who starts the game you or the computer.

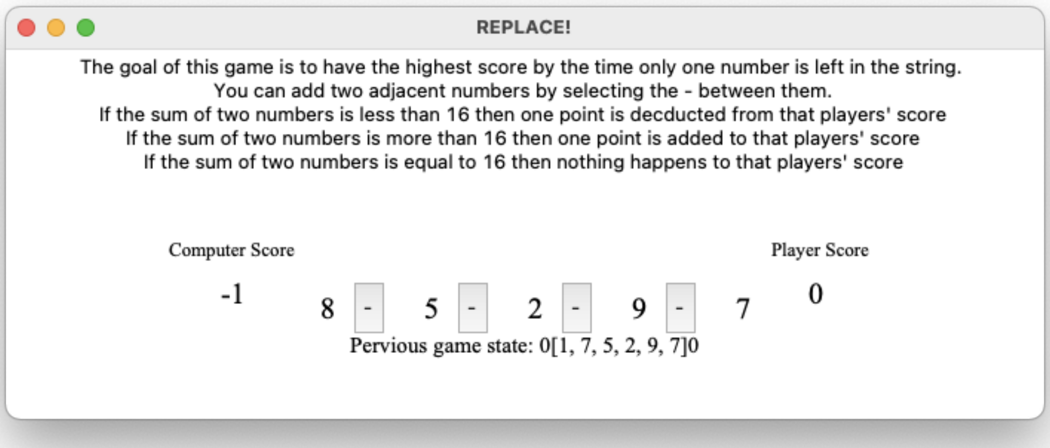
2. Once you select one of the options one of the screens shown below should appear depending on what option you selected.

   If you selected player:



The goal of this game is to have the highest score by the time only one number is left in the string.
You can add two adjacent numbers by selecting the - between them.
If the sum of two numbers is less than 12 then one point is decducted from that players' score
If the sum of two numbers is more than 12 then one point is added to that players' score
If the sum of two numbers is equal to 12 then nothing happens to that players' score

If you selected Computer:


REPLACE!

The goal of this game is to have the highest score by the time only one number is left in the string.
You can add two adjacent numbers by selecting the - between them.
If the sum of two numbers is less than 16 then one point is decducted from that players' score
If the sum of two numbers is more than 16 then one point is added to that players' score
If the sum of two numbers is equal to 16 then nothing happens to that players' score

Computer Score -1

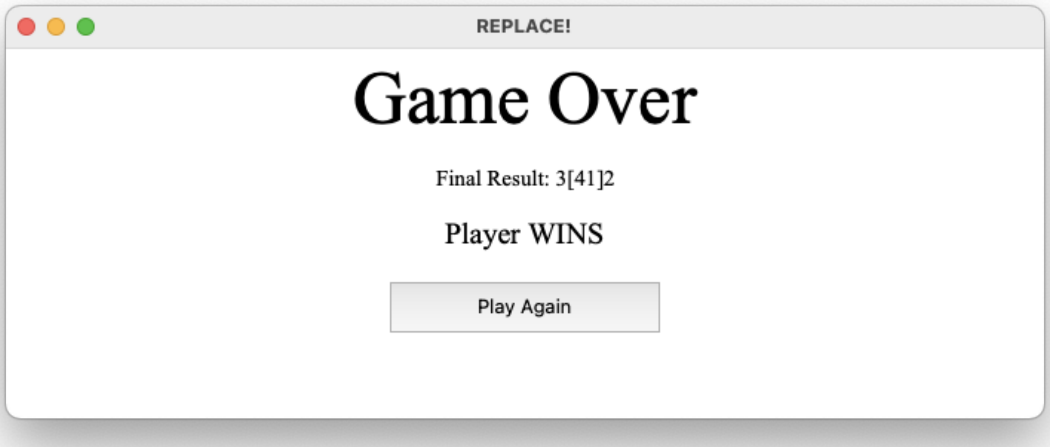
8 - 5 - 2 - 9 - 7

Player Score 0

Pervious game state: 0[1, 7, 5, 2, 9, 7]0

On the top of this window, you can see rules of the game. When the computer starts the game, you can see that there is a section called previous game state this shows up if the computer has made a move and it shows the state of the game when the computer made the move.

3. Now all you have to do is play the game until you or the computer wins.


REPLACE!

Game Over

Final Result: 3[41]2

Player WINS

4. Then you can choose to play again. Remember that the numbers that the string of numbers are generate randomly at the start of every game to not make the game repetitive.

HAVE FUN PLAYING!!!

Data structures used

LIST

List is an ordered data structure which data that is separated by commas. This is the most commonly used data structure in the program thanks to easy of accessing data using indexes. Given below are all the most important lists used, categorized by what the data stored within them represents.

parameter/par

deceleration:

```
        parameter=[0,0,0,0,0,0,0,0,None]
```

This list contains the parameters of the game where the first elements are the score of the player that starts the game, the second element is the score of the other player, the remaining elements except for the last one represents the numbers string and the last one represents the heuristics values of that state.

path

deceleration:

```
             global path
             path=[]
```

This global list holds the current state as well as the previous states of the game in the form of nodes from the game tree. Since it can be accessed in almost any function within the program this makes the path list the go to list to find the current or previous state of the game.

move_[x]

decelerations:

```
        global move_4
        move_4=[]
        global move_3
        move_3=[]
        global move_2
        move_2=[]
        global move_1
        move_1=[]
        global move_0
        move_0=[]
```

This nested list stores all the possible moves that can be made. Where move_0 holds all the possible moves that the staring player/computer can move and move_1 holds all the possible moves that the other player can make. These list where used to assign the heuristic values of all possible moves as well as to create the nodes for the game_tree. Creating these list as global variables means that they can be accessed within a function without passing said lists as arguments for the function.

node_[x]

deceleration*s*:

```
        global node_4
        node_4=[]
        global node_3
        node_3=[]
        global node_2
        node_2=[]
        global node_1
        node_1=[]
```

These nested lists are the same as the move_[x], but contain all possible moves as nodes of the state space graph. These lists are used in the creation of the game_tree where for each of the states within the list all possible states that proceed given state are added to the game_tree.

Tree

game_tree

This data structure is a collection of nodes that originate from a single node. Each node contains of data in the form of a list that holds the parameters corresponding to the state of a game as well as the heuristic value.

Algorithms used

Creation of State Space Graph.

Defining all the possible states of the games.

```
def states(par,avg,move):
    global node_4
    node_4=[]
    global node_3
    node_3=[]
    global node_2
    node_2=[]
    global node_1
    node_1=[]
    global node_0
    node_0=[]
    global move_4
    move_4=[]
    global move_3
    move_3=[]
    global move_2
    move_2=[]
    global move_1
    move_1=[]
    global move_0
    move_0=[]
    par=tuple(par)
    def generate(par,avg,move):
        if move=='move_0':
                    start=list(par)
                    i=2
                    while i<=len(par)-3:
                        temp=list(par)

                        if par[i]+par[i+1]<avg:
                            temp[0]+=-1
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        elif par[i]+par[i+1]>avg:
                            temp[0]+=1
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        elif par[i]+par[i+1]==avg:
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)
                        move_0.append(temp)
                        i+=1

                    for w in move_0:
                        generate(w,avg,'move_1')
                    for x in move_1:
                        generate(x,avg,'move_2')
                    for y in move_2:
                        generate(y,avg,'move_3')
                    for z in move_3:
                        generate(z,avg,'move_4')

                    assign_heuristic_value()
                    start=game_tree(start,0,0)
                    path.append(start)
                    return 0

        if move=='move_1':
                  i=2
                  while i<=len(par)-3:

                        temp=list(par)

                        if par[i]+par[i+1]<avg:
                            temp[1]+=-1
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        elif par[i]+par[i+1]>avg:
                            temp[1]+=1
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        elif par[i]+par[i+1]==avg:
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        move_1.append(temp)
                        i+=1


        if move=='move_2':
                  i=2
                  while i<=len(par)-3:

                        temp=list(par)
```

```

         if par[i]+par[i+1]<avg:
             temp[0]+=-1
             temp[i]=temp[i]+temp[i+1]
             temp.pop(i+1)

         elif par[i]+par[i+1]>avg:
             temp[0]+=1
             temp[i]=temp[i]+temp[i+1]
             temp.pop(i+1)

                elif par[i]+par[i+1]==avg:
                    temp[i]=temp[i]+temp[i+1]
                    temp.pop(i+1)

                move_2.append(temp)
                i+=1

if move=='move_3':
          i=2
          while i<=len(par)-3:

                temp=list(par)

                if par[i]+par[i+1]<avg:
                    temp[1]+=-1
                    temp[i]=temp[i]+temp[i+1]
                    temp.pop(i+1)

                elif par[i]+par[i+1]>avg:
                    temp[1]+=1
                    temp[i]=temp[i]+temp[i+1]
                    temp.pop(i+1)

                elif par[i]+par[i+1]==avg:
                    temp[i]=temp[i]+temp[i+1]
                    temp.pop(i+1)

                move_3.append(temp)
                i+=1

if move=='move_4':
          i=2
          while i<=len(par)-3:

                temp=list(par)

                if par[i]+par[i+1]<avg:
```

```
                            temp[0]+=-1
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        elif par[i]+par[i+1]>avg:
                            temp[0]+=1
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        elif par[i]+par[i+1]==avg:
                            temp[i]=temp[i]+temp[i+1]
                            temp.pop(i+1)

                        player1=int(temp[0])
                        player2=int(temp[1])

                        if player1==player2:
                            temp[-1]=0

                        if player1>player2:
                            temp[-1]=1

                        if player1<player2:
                            temp[-1]=-1


                        move_4.append(temp)
                        i+=1
    generate(par,avg,move)
```

This function is called at the start of the game, and it defines all the possible states that the games can take and adds them to lists based on which stage of the game the state is in. The arguments this function takes are the staring parameter of the game(par) and the value that is used to define the rule of adding or subtracting a value from the players score(avg) and the state the game is in.

The `generate(par,avg,move)` is an iterative function that takes the current state then moves creates all possible next moves from the current state by adding all the adjacent numbers and calculating the resulting score change due to this addition. It also assigns the heuristic values for the final states of the game (state when game is over, and winner is decided) according to the min max algorithm where a victory is equal to +1 and a defeat is equal to -1.

Once this function is executed all the necessary tasks and it calls on the function that assigns heuristics values to all the states then it calls on the function that creates the game tree and adds the root node to the path.

Assigning heuristic values to each of the states.

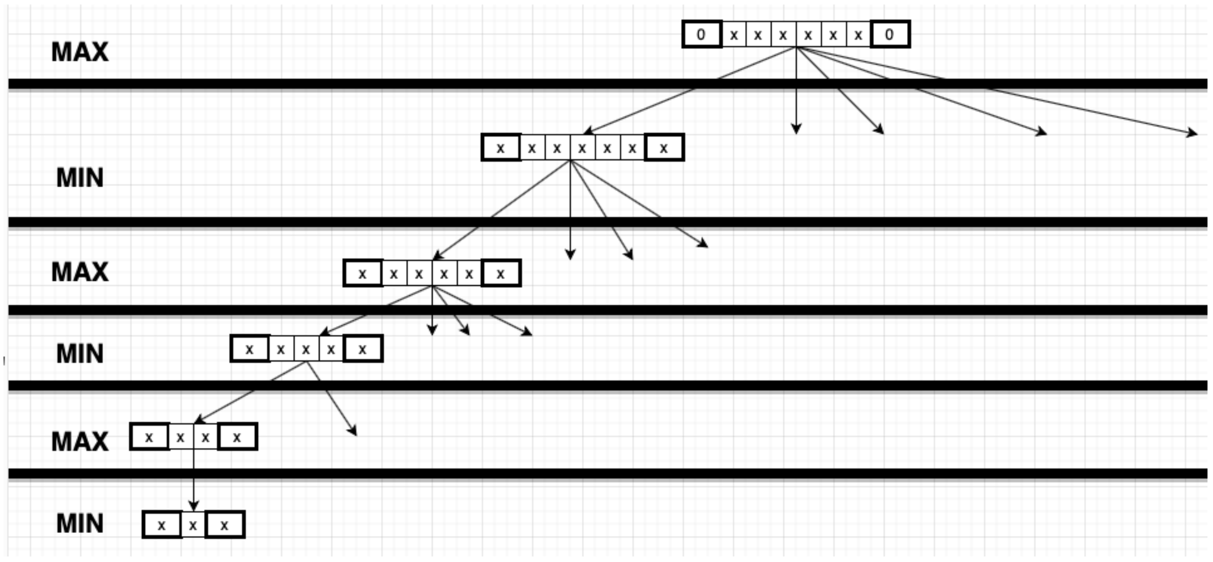
```
def assign_heuristic_values():
    for i in range(len(move_4)):
        move_3[i][-1]=move_4[i][-1]

    j=0
    while j<len(move_2):
        move_2[j][-1]=max(move_3[j][-1],move_3[j+1][-1])
        j+=1

    k=0
    while k<len(move_1):
        move_1[k][-1]=min(move_2[k][-1],move_2[k+1][-1],move_2[k+2][-1])
        k+=1

    l=0
    while l<len(move_0):
        move_0[l][-1]=max(move_1[l][-1],move_1[l+1][-1],move_1[l+2][-1],move_1[l+3][-1])
        l+=1
```

This function takes all the possible states starting with the list containing the states before the final states(the states when the game is finished) which already have heuristic values assigned according to the min max algorithm and assigns the min/max heuristic value of the corresponding final states. The min max levels are defined as follows.



Creating game tree based on the states.

```python
def game_tree(par,level,index):
    class Node:
        def __init__(self, data):
            self.data = list(data)
            self.move1=None
            self.move2=None
            self.move3=None
            self.move4=None
            self.move5=None
        def insert(self,data):
            if self.move1==None:
                self.move1 = Node(data)
                return self.move1

            elif self.move2==None:
                self.move2 = Node(data)
                return self.move2

            elif self.move3==None:
                self.move3 = Node(data)
                return self.move3

            elif self.move4==None:
                self.move4 = Node(data)
                return self.move4

            elif self.move5==None:
                self.move5 = Node(data)
                return self.move5

    if level==0:
        start=Node(par)
        for w in move_0:
            inserted=start.insert(w)
            node_0.append(inserted)

        x1=0
        for x in node_0:
            game_tree(x,1,x1)
            x1+=1

        y1=0
        for y in node_1:
            game_tree(y,2,y1)
            y1+=1

        z1=0
        for z in node_2:
            game_tree(z,3,z1)
            z1+=1

        v1=0
        for v in node_3:
            game_tree(v,4,v1)
            v1+=1
        return start


    if level==1:
        start1=par
        i=0
        while i<4:
            inserted=start1.insert(move_1[index*4+i])
            node_1.append(inserted)
            i+=1

    if level==2:
        start2=par
        i=0
        while i<3:
            inserted=start2.insert(move_2[index*3+i])
            node_2.append(inserted)
            i+=1

    if level==3:
        start3=par
        i=0
        while i<2:
            inserted=start3.insert(move_3[index*2+i])
            node_3.append(inserted)
            i+=1

    if level==4:
        start4=par
        inserted=start4.insert(move_4[index])
        node_4.append(inserted)
```

This function has some similarities to the generate() function. It goes through the states in the current level of the games and add corresponding next states as the next nodes.

Finding Winning Path

```python
def computer_move_maximiser():
    if len(path[-1].data)==4:
        current_state(path[-1].data)
    else:
        possible_moves=[]
        if path[-1].move1:
            possible_moves.append(path[-1].move1)
        if path[-1].move2:
            possible_moves.append(path[-1].move2)
        if path[-1].move3:
            possible_moves.append(path[-1].move3)
        if path[-1].move4:
            possible_moves.append(path[-1].move4)
        best_move=possible_moves[0]
        for i in possible_moves:
            if i.data[-1]>best_move.data[-1]:
                best_move=i
        path.append(best_move)
        current_state(best_move.data)
```

```python
def computer_move_minimiser():
    if len(path[-1].data)==4:
        current_state(path[-1].data)
    else:
        possible_moves=[]
        if path[-1].move1:
            possible_moves.append(path[-1].move1)
        if path[-1].move2:
            possible_moves.append(path[-1].move2)
        if path[-1].move3:
            possible_moves.append(path[-1].move3)
        if path[-1].move4:
            possible_moves.append(path[-1].move4)
        best_move=possible_moves[0]
        for i in possible_moves:
            if i.data[-1]<best_move.data[-1]:
                best_move=i
        path.append(best_move)
        current_state(best_move.data)
```

The two functions are identical except of the fact one chooses to make the move that produces the state with the maximum heuristic value(if computer is maximiser) and the other chooses the move that produces the state with the minimum heuristic value(if computer is minimiser).