B. TECH FINAL SEMESTER PROJECT REPORT

NAME: ARHAN DAS

ROLL:16CS8010

1.Objectives:

Create an app that has the following features:

1. Allows a user to register their details, store them in a file and login
2. Create a board using player input for size
3. Allow player to place or remove obstacles on the board that a pawn has to navigate through to reach from one end of a diagonal to the other.

2.Requirements:

1. Python 3.7.7
2. kivy 1.11. 1

3.For the pawn to find a path from its current cell to a destination cell we will use A* heuristic search.

A* has the following properties:

- It is complete; it will always find a solution if it exists.
- It can use a heuristic to significantly speed up the process.
- It can have variable node to node movement costs. This enables things like certain nodes or paths being more difficult to traverse.
- It can search in many different directions if desired.

The heuristic can be used to control A*'s behaviour.

- At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path.
- If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower.

- If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into Greedy Best-First-Search.

Speed or accuracy?

A*'s ability to vary its behaviour based on the heuristic and cost functions can be very useful in a game. The trade-off between speed and accuracy can be exploited to make your game faster. For most games, you don't really need the best path between two points. You need something that's close. What you need may depend on what's going on in the game, or how fast the computer is. Using a function that guarantee*s* it never overestimates the cost means that it will sometimes underestimate the cost by quite a bit.

Node

A node has a positioning value (e.g. x, y), a reference to its parent and three 'scores' associated with it. These scores are how A* determines which nodes to consider first.

G score

The g score is the base score of the node and is simply the incremental cost of moving from the start node to this node.

g(n)=g(n.parent)+cost(n.parent,n)
cost(n1, n2)=the movement cost from n1 to n2

H score - the heuristic

The heuristic is a computationally easy estimate of the distance between each node and the goal. This being computationally easy is very important as the H score will be calculated at least once for every node considered before reaching the goal. The implementation of the H score can vary depending on the properties of the graph being searched, here are the most common heuristics.

F score

The f score is simply the addition of g and h scores and represents the total cost of the path via the current node.

f(n)=g(n)+h(n)

Heuristics for grid maps

On a grid, there are well-known heuristic functions to use.

Use the distance heuristic that matches the allowed movement:

- On a square grid that allows **4 directions** of movement, use Manhattan distance (L_1).

- On a square grid that allows **8 directions** of movement, use Diagonal distance (L_∞).
- On a square grid that allows **any direction** of movement, you might or might not want Euclidean distance (L_2). If A* is finding paths on the grid but you are allowing movement not on the grid, you may want to consider other representations of the map.
- On a hexagon grid that allows **6 directions** of movement, use Manhattan distance adapted to hexagonal grids.

Diagonal distance

If your map allows diagonal movement you need a different heuristic. The Manhattan distance for (4 east, 4 north) will be 8×D. However, you could simply move (4 northeast) instead, so the heuristic should be 4×D2, where D2 is the cost of moving diagonally.

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Here we compute the number of steps you take if you can't take a diagonal, then subtract the steps you save by using the diagonal. There are min(dx, dy) diagonal steps, and each one costs D2 but saves you 2×D non-diagonal steps.

When D = 1 and D2 = 1, this is called the Chebyshev distance. When D = 1 and D2 = sqrt(2), this is called the *octile distance*.

4.A_StarGraph.py

This stores the class to provide functions to store the position of barriers, return the heuristic function value of each cell, return neighbours of each cell, and the cost to move to a neighbouring cell.

In the main, we define an object of AStarGraph type to store the size of the board and barriers entered through the UI.

```
class AStarGraph(object):

    #Define a class board like grid with barriers as entered

    def __init__(self,barriers,rows,cols):

        self.barriers=[]

        self.barriers=barriers

        self.rows=rows

        self.cols=cols


    def heuristic(self, start, goal):
```

```
    #Use Chebyshev distance heuristic
        D = 1
        D2 = 1
        dx = abs(start[0] - goal[0])
        dy = abs(start[1] - goal[1])
        return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
    #gets all the neigbours of a particular cell
    def get_vertex_neighbours(self, pos):
        n = []
        for dx, dy in [(1,0),(-1,0),(0,1),(0,-1),(1,1),(-1,1),(1,-1),(-1,-1)]:
            x2 = pos[0] + dx
            y2 = pos[1] + dy
            #checks if neighbour is outside of board or not
            if x2 < 0 or x2 >= self.rows or y2 < 0 or y2 >= self.cols:
              continue
            n.append((x2, y2))
        return n #returns all the neigbours appended to n
    #returns the cost to move to b from a, if there is a barrier on b, returns 100, else 1
    def move_cost(self, a, b):
        for barrier in self.barriers:
            if (b[0]==barrier[0] and b[1]==barrier[1]):
                return 100 #Extremely high cost to enter barrier squares
        return 1 #Normal movement cost
```

5.database.py

It defines the class DataBase and all the associated functions to store data in a text file and access it.

In the main.py, we import this and define an object of type DataBase to store login details in the file "users.txt".

```
import datetime
class DataBase:
```

```
 #attaches a file to the object.
 def __init__(self, filename):
     self.filename = filename
     self.users = None
     self.file = None
     self.load()
 #stores the details of each user from file in a dictionary users.
 def load(self):
     self.file = open(self.filename, "r")
     self.users = {}
     for line in self.file:
         email, password, name, created = line.strip().split(";")
         self.users[email] = (password, name, created)
     self.file.close()

 #if an user with the given email exists, the corresponding details are returned.
 def get_user(self, email):
     if email in self.users:
         return self.users[email]
     else:
         return -1
#adds new user, taking email, password, name as input
 def add_user(self, email, password, name)
     if email.strip() not in self.users: #checks if email is present
         self.users[email.strip()] = (password.strip(), name.strip(), DataBase.get_date())
         self.save()  #calls save function to write details of new user into file
         return 1
     else:
         print("Email exists already")
         return -1

 #validates user email and passord
```

```
    def validate(self, email, password):

    #checks if email us present and then returns true if password matches that from record
        if self.get_user(email) != -1:

            return self.users[email][0] == password
        else:
            return False
#writes details of user into file, called during new user addition.
    def save(self):
        with open(self.filename, "w") as f:
            for user in self.users:
                f.write(user + ";" + self.users[user][0] + ";" + self.users[user][1] + ";" + self.users[user][2] + "\n")

   #a static method to return time date when an user is added
    @staticmethod
    def get_date():
        return str(datetime.datetime.now()).split(" ")[0]
```

Users.txt file

users.txt - Notepad

File Edit Format View Help

```
ad@gmail.com;12345;Arhan Das;2020-05-24
barun@gmail.com;BD2038!;Barun Das;2020-05-24
tim@yahoo.com;12345678;Tim Burke;2020-06-07
arhandas1998ad@gmail.com;12345;Arhan Das;2020-06-10
```

6.Kivy

We will be designing our UI using the kivy framework.

Kivy is an opensource multi-platform GUI development library for Python and can run on iOS, Android, Windows, OS X, and GNU/Linux. It helps develop applications that make use of innovative, multi-touch UI. The fundamental idea behind Kivy is to enable the developer to build an app once and use it across all devices, making the code reusable and deployable, allowing for quick and easy interaction design and rapid prototyping.

This easy to use framework contains all the elements for building an application such as:

- Extensive input support for input devices such as mouse, keyboard, TUIO, and OS-specific multi-touch events
- A graphic library using only OpenGL ES 2
- A wide range of widgets built with multi-touch support
- An intermediate language Kv language, used to design custom widgets easily

Files with the KV extension are used by Kivy. The KV file itself contains various kinds of data, such as definitions of rules and dynamic classes, templates, and a root widget. The root widget contained inside the KV file is a core building block in Kivy.

We will be using a kv file for the “create account” and “Login” windows UI. The other windows will be done in the main file itself.

How to load KV

There are two ways to load Kv code into our application:

- By name convention:

Kivy looks for a Kv file with the same name as our App class in lowercase, minus “App” if it ends with ‘App’ e.g:

MyApp -> my.kv

If this file defines a *Root Widget* it will be attached to the App’s *root* attribute and used as the base of the application widget tree.

- Builder: We can tell Kivy to directly load a string or a file. If this string or file defines a root widget, it will be returned by the method:

  Builder.load_file('path/to/file.kv')

  or:

  Builder.load_string(kv_string)

FloatLayout

Floatlayout allows us to place the elements relatively based on the current window size and height especially in mobiles i.e. Floatlayout allow us to place the elements using something called relative position. This means rather than defining the specific position or the coordinates we will place everything using the percentage w.r.t the size of window. When we change the dimensions of the window everything placed in the window will adjust its size and position accordingly. This makes the Application more reliable and scalable to window size.

BoxLayout

BoxLayout arranges widgets in either in a vertical fashion that is one on top of another or in a horizontal fashion that is one after another. When we will not provide any size-hint then the child widgets divide the size of its parent widget equally or accordingly.

GridLayout

- The widget must be placed in a specific column/row. Each child is automatically assigned a position determined by the layout configuration and the child's index in the children list.
- Grid Layout must always contain any one of the below input constraints: GridLayout.cols or GridLayout.rows. If we do not specify cols or rows, the Layout will throw an exception.
- The GridLayout arranges children in a matrix. It takes the available space and divides it into columns and rows, then adds widgets to the resulting "cells".
- The row and coloums are just like the same as we observe in a matrix, here we can adjust the size of each grid.

ScreenManager widget:

The screen manager is a widget which is used to managing multiple screens for your application. The default ScreenManager displays only one Screen at a time and uses a TransitionBase to switch from one Screen to another. Multiple transitions are supported.

The ScreenManager and Screen class are imported. The ScreenManager will be used for the root like:

```
from kivy.uix.screenmanager import ScreenManager, Screen
```

7.my.kv

```
<CreateAccountWindow>:
    #screen name is set to "create", used by screenmanager object
    name: "create"
   #defines properties of the "create" screen
    namee: namee
    email: email
    password: passw

    FloatLayout: #Root widget of the create account window
        cols:1
        FloatLayout: #another child widget within the root layout
```

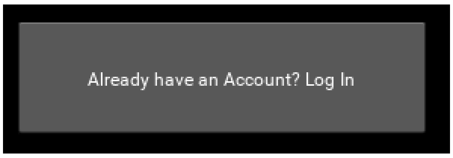
```
size: root.width, root.height/2 #sets its width to same as root, but its height is half

Label:
    text: "Create an Account"
#           size_hint: 0.8, 0.2
    pos_hint: {"x":0.1, "top":1}
    font_size: (root.width**2 + root.height**2) / 14**4
```


```
Label:
    size_hint: 0.5,0.12
    pos_hint: {"x":0, "top":0.8}
    text: "Name: "
    font_size: (root.width**2 + root.height**2) / 14**4
```

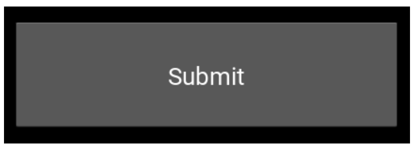

```
TextInput:   #text field to enter namee
    pos_hint: {"x":0.5, "top":0.8}
    size_hint: 0.4, 0.12
    id: namee
    multiline: False
    font_size: (root.width**2 + root.height**2) / 14**4
```

```
Label:    #creates an Email label, similar to Name
    size_hint: 0.5,0.12
    pos_hint: {"x":0, "top":0.8-0.13}
    text: "Email: "
    font_size: (root.width**2 + root.height**2) / 14**4

TextInput:  #text field to enter email
    pos_hint: {"x":0.5, "top":0.8-0.13}
    size_hint: 0.4, 0.12
    id: email
    multiline: False
```

```
        font_size: (root.width**2 + root.height**2) / 14**4

    Label:    #creates a Password label
        size_hint: 0.5,0.12
        pos_hint: {"x":0, "top":0.8-0.13*2}
        text: "Password: "
        font_size: (root.width**2 + root.height**2) / 14**4

    TextInput:    #text field to enter password
        pos_hint: {"x":0.5, "top":0.8-0.13*2}
        size_hint: 0.4, 0.12
        id: passw
        multiline: False
        password: True
        font_size: (root.width**2 + root.height**2) / 14**4

#Button to go back to login screen without entering details
Button:
    pos_hint:{"x":0.3,"y":0.25}
    size_hint: 0.4, 0.1
    font_size: (root.width**2 + root.height**2) / 17**4
    text: "Already have an Account? Log In"
    on_release:
        root.manager.transition.direction = "left"
        root.login()
```




```
#Button to go back to login screen after submitting details and creating new record
Button:
    pos_hint:{"x":0.2,"y":0.05}
    size_hint: 0.6, 0.15
    text: "Submit"
```

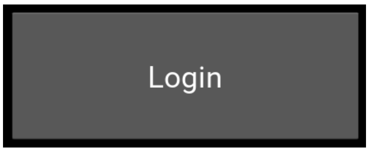


```
        font_size: (root.width**2 + root.height**2) / 14**4
        on_release:
            root.manager.transition.direction = "left"
            root.submit()

<LoginWindow>:
   # name of screen is “login”, use by screenmanager object
    name: "login"
    #defines properties of the screen
    email: email
    password: password
     #root widget
    FloatLayout:

        Label:
            text:"Email: "
            font_size: (root.width**2 + root.height**2) / 13**4
            pos_hint: {"x":0.1, "top":0.9}
            size_hint: 0.35, 0.15
```

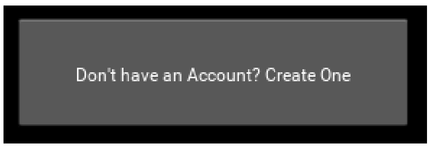

```
        TextInput:   #text field to enter email
            id: email
            font_size: (root.width**2 + root.height**2) / 13**4
            multiline: False
            pos_hint: {"x": 0.45 , "top":0.9}
            size_hint: 0.4, 0.15
```

```
        Label:
            text:"Password: "
            font_size: (root.width**2 + root.height**2) / 13**4
            pos_hint: {"x":0.1, "top":0.7}
            size_hint: 0.35, 0.15
```

```
TextInput:  #text field to enter password
    id: password
    font_size: (root.width**2 + root.height**2) / 13**4
    multiline: False
    password: True
    pos_hint: {"x": 0.45, "top":0.7}
    size_hint: 0.4, 0.15

#button to login and go to intro screen
Button:
    pos_hint:{"x":0.2,"y":0.05}
    size_hint: 0.6, 0.2
    font_size: (root.width**2 + root.height**2) / 13**4
    text: "Login"
```

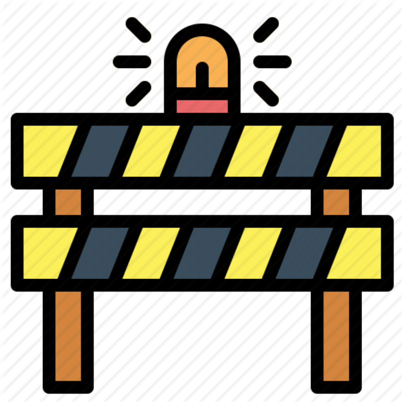



```
    on_release:
        #transition animation in the upwards direction
        root.manager.transition.direction = "up"
        #calls loginBtn() from main.py when pressed
        root.loginBtn()

#button to go to create account screen
Button:
    pos_hint:{"x":0.3,"y":0.3}
    size_hint: 0.4, 0.1
    font_size: (root.width**2 + root.height**2) / 17**4
    text: "Don't have an Account? Create One"
```




```
    on_release:
        #transition animation to the right
        root.manager.transition.direction = "right"
        #calls createBtn() from main.py when pressed
        root.createBtn()
```

8.The icon images used



player.png

barrier.png

finish_line.png

finish_player.png

The black backgrounds are for visibility, actual files have transparent backgrounds

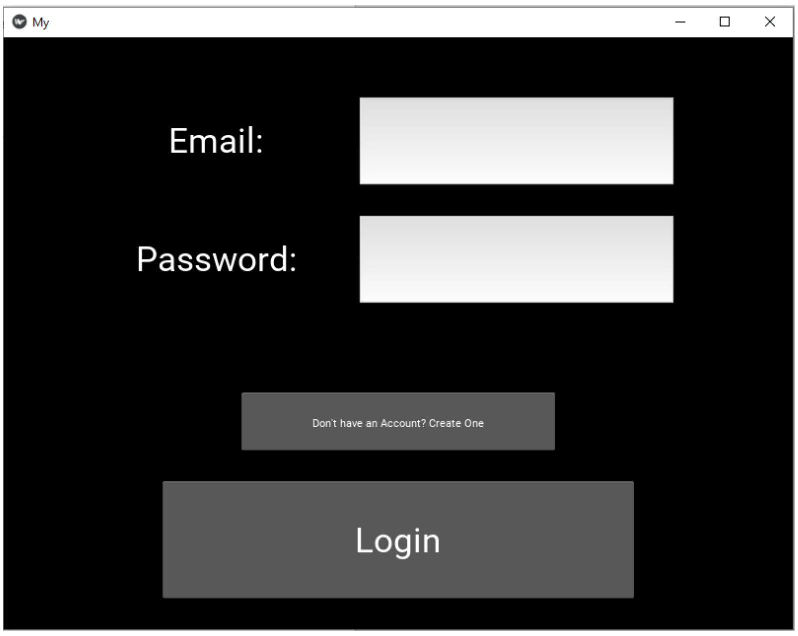
9.main.py

This is the main file to be executed. Below are all the packages that are imported

```
# re module provides support
# for regular expressions
import re
from kivy.app import App
from kivy.properties import ObjectProperty
```

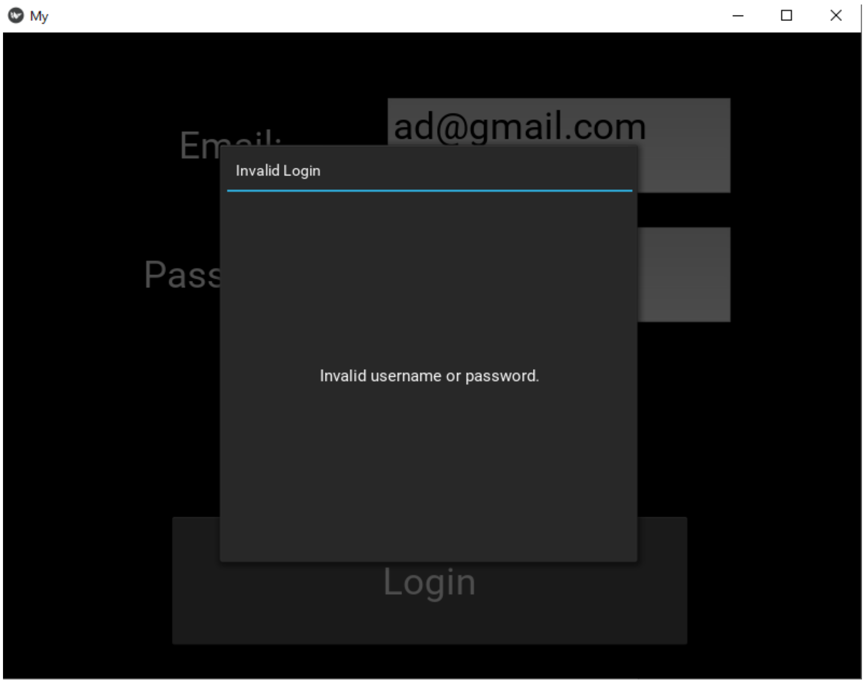
```
from kivy.uix.popup import Popup
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.uix.textinput import TextInput
from database import DataBase  #from database.py
from a_StarGraph import AStarGraph  #from a_StarGraph.py
from kivy.lang import Builder
```

I. The Login Screen



This is the first screen that opens. It has two input fields for Email and Password if you have an existing account.

If login details don't match a popup appears



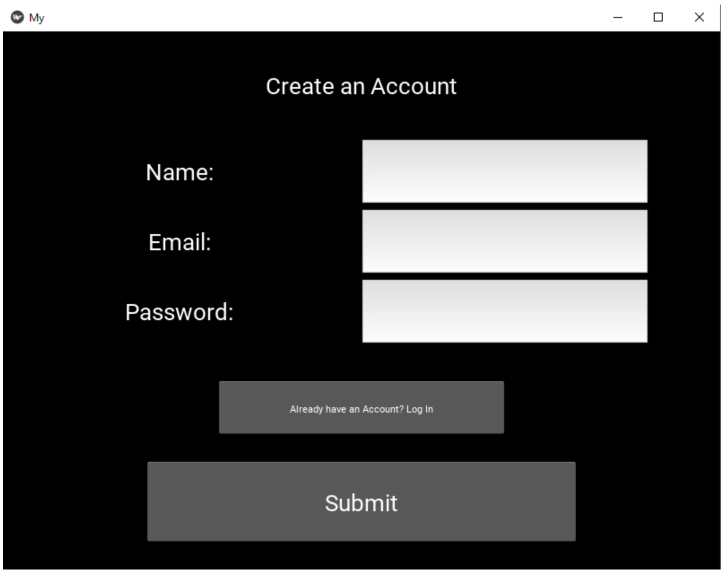
<u>Code:</u>

```
class LoginWindow(Screen):
    #sets properties to None
    email = ObjectProperty(None)
    password = ObjectProperty(None)
#on clicking the login button, this function is called
    def loginBtn(self):
#calls validate function from DataBase class in database.py
        if db.validate(self.email.text, self.password.text):
            self.reset() #calls reset() function to clear the input fields
            sm.current = "intro" # sets current screen to “intro” screen
        else:
            invalidLogin()  #calls invalidLogin() function that displays the popup
#on clicking the create button, this function is called
    def createBtn(self):
        self.reset()
        sm.current = "create" #sets current screen to “create” screen

 #sets the text and password field to blank
```

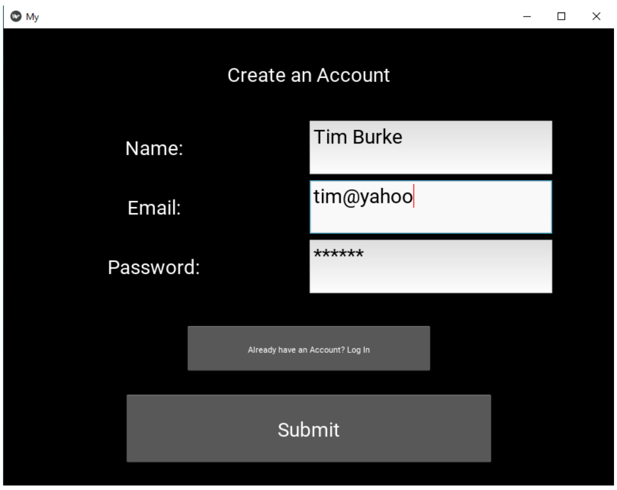
```
def reset(self):
    self.email.text = ""
    self.password.text = ""
```

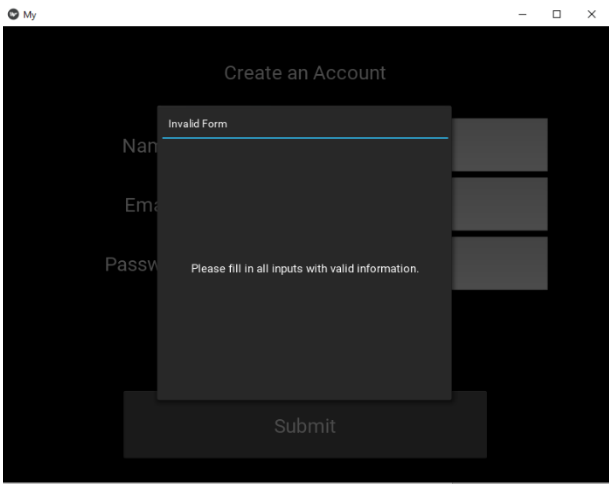
On clicking the “Don’t have an Account?” button we will be open the Create Account screen.

II. Create Account Screen

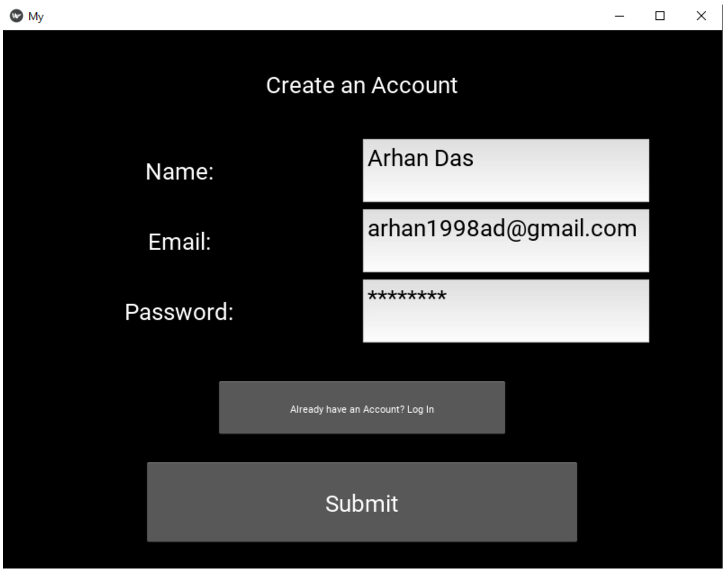


If we enter invalid input, we will get an Invalid Form popup





If we already have an account, we could click the button which will take us back to login screen. Entering all details correctly and clicking submit, stores the details in users.txt and transitions to the login screen.



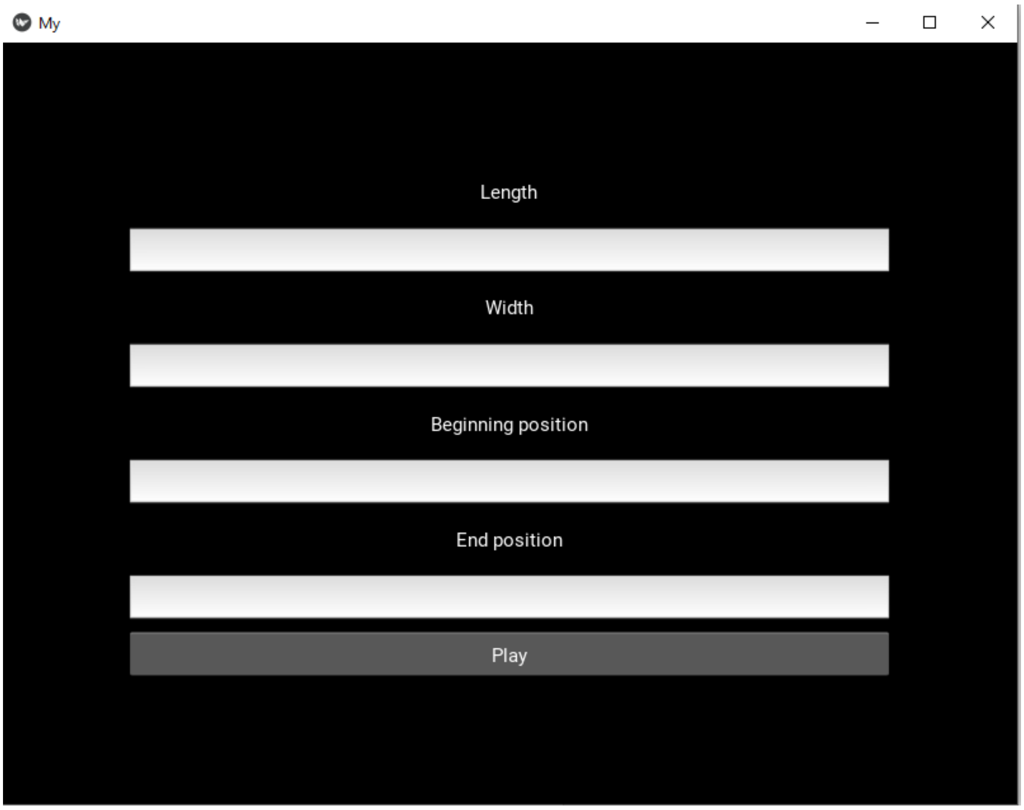
Code:

```
class CreateAccountWindow(Screen):
  #sets properties to None
   namee = ObjectProperty(None)
   email = ObjectProperty(None)
   password = ObjectProperty(None)
   # Make a regular expression
   # for validating an Email
   regex = '^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'
  #on clicking submit button, this function is executed
   def submit(self):
     #checks whether details are valid or not
      if self.namee.text != "" and re.search(regex,self.email,text):
         if self.password != "":
           #adds details to users.txt
            db.add_user(self.email.text, self.password.text, self.namee.text)
            self.reset()
            sm.current = "login"
         else: #calls the function to display popup
```

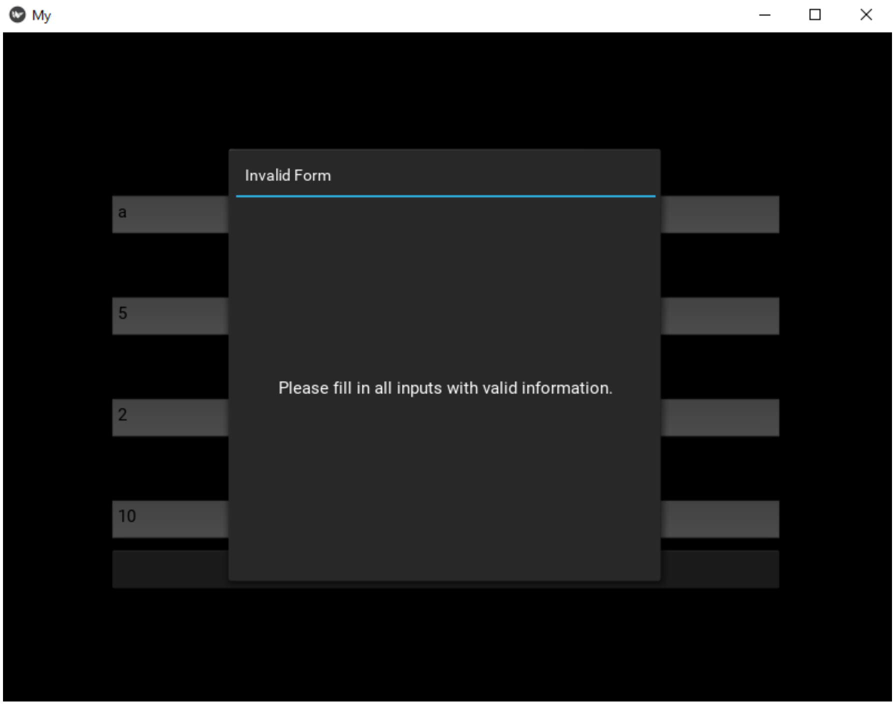
```
            invalidForm()
        else:
            invalidForm()

#called on clicking “Log in button”
  def login(self):
      self.reset()
      sm.current = "login" #changes screen to login screen

   #resets text fields
   def reset(self):
      self.email.text = ""
      self.password.text = ""
      self.namee.text = ""
```

After successful login, we are brought to the Intro Screen

III. The Intro Screen



Entering incorrect input, will make a popup appear



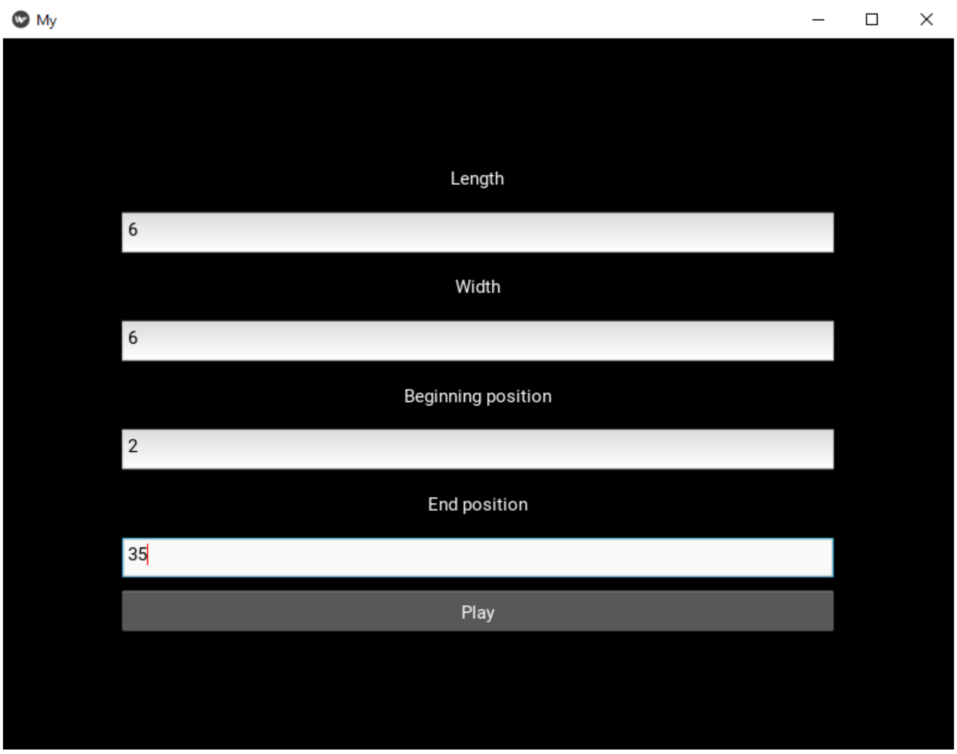
<u>Code:</u>

```
class IntroScreen(Screen):
    def __init__(self, **kw):
        super().__init__(**kw)
        #create root layout as a box layout
        self.my_root_widget = BoxLayout(spacing=10, padding=100, orientation='vertical')
        #create play button which calls start_game when pressed
        self.play_button = Button(text="Play")
        self.play_button.bind(on_press=self.start_game)
        #create length, width labels
        self.length_label = Label(text="Length")
        self.width_label = Label(text="Width")
        #create text input fields for length and width
        self.length_input = TextInput(text='', multiline=False)
        self.width_input = TextInput(text='', multiline=False)
        #create begin, end labels
        self.begin_label = Label(text="Beginning position")
```

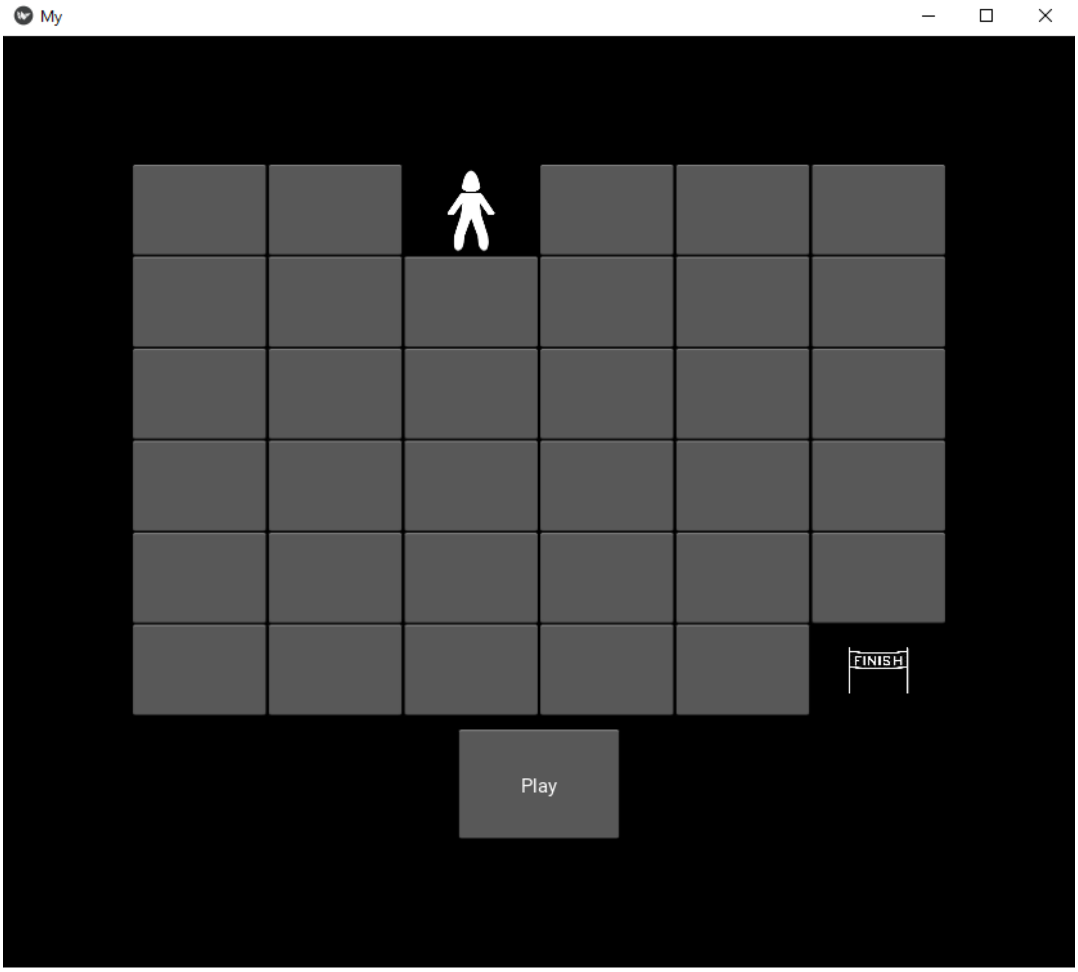
```
        self.end_label = Label(text="End position")
        #create text input fields for end and begin(1d index)
        self.begin_input = TextInput(text='', multiline=False)
        self.end_input = TextInput(text='', multiline=False)
        #add the widgets to the root layout
        self.my_root_widget.add_widget(self.length_label)
        self.my_root_widget.add_widget(self.length_input)
        self.my_root_widget.add_widget(self.width_label)
        self.my_root_widget.add_widget(self.width_input)
        self.my_root_widget.add_widget(self.begin_label)
        self.my_root_widget.add_widget(self.begin_input)
        self.my_root_widget.add_widget(self.end_label)
        self.my_root_widget.add_widget(self.end_input)
        self.my_root_widget.add_widget(self.play_button)
        self.add_widget(self.my_root_widget)

    #called when play is pressed
    def start_game(self, _):
    #initialises variables with input from text fields
        length = self.length_input.text
        width = self.width_input.text
        begin = self.begin_input.text
        end = self.end_input.text
        # validating user input is numeric
        if not length.isnumeric() or not width.isnumeric() or not begin.isnumeric() or not
end.isnumeric():
            invalidForm()
            return
        max_cell=int(length)*int(width) #max no of cells in board
        #check if begin and end are within range
        if int(begin)<0 or int(end)<0 or int(begin)>=max_cell or int(end)>=max_cell or
int(length) <= 0 or int(width) <= 0:
```

```
        invalidForm()
        return

    length = int(length)
    width = int(width)
    begin=int(begin)
    end=int(end)
    # initialise game screen and set as current
    game_screen_name = 'game'
    game_screen = sm.get_screen(game_screen_name)
    game_screen.init(length, width,begin,end)
    sm.current = game_screen_name
```

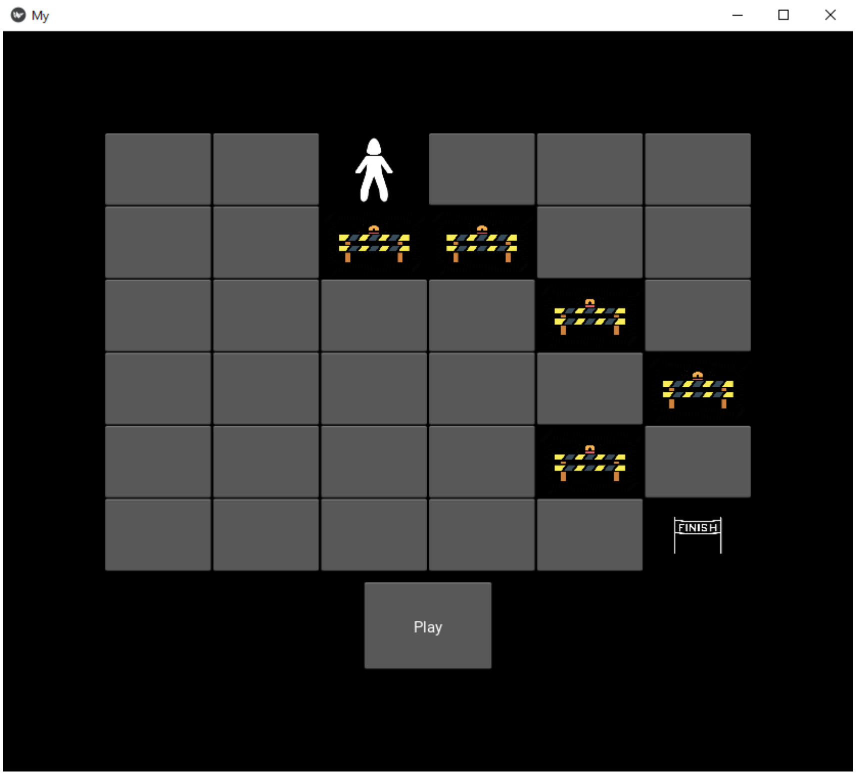
Entering the required input correctly, will bring us to the game screen



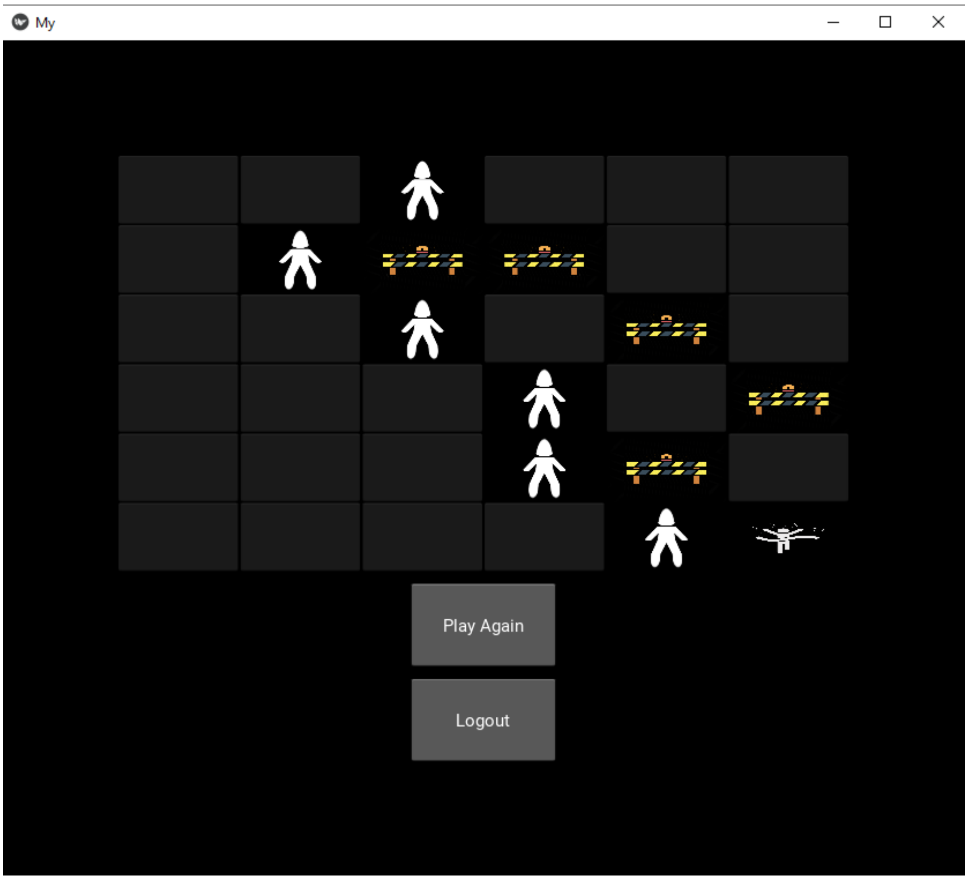
IV. Game Screen



We can set the obstacles by clicking on the tiles. Selecting an obstacle will remove it.



By clicking play, a star search algorithm is executed and the path with minimum cost is selected. The player icon on the tiles signify nodes in its path to the goal. Change in goal tile icon signifies goal has been reached



Clicking the "Play Again" button brings us to Intro screen to enter new input and run the game again.

Clicking the "Logout" button brings us to the Login screen.

Code:

```
#This class is use to create clickable tiles in the game board
class Tile(Button):
    def __init__(self, game, index, is_obstacle_flag=False,**kwargs):
        super().__init__(**kwargs)
        self.game = game
        self.tile_index = index #stores index of tile in tile_list
        self.is_obstacle_flag = is_obstacle_flag
  #function to set obstacle flag as True or False at tile
    def set_obstacles(self,flag):
```

```
        self.is_obstacle_flag = flag
  #returns whether tile is an obstacle or not
    def is_obstacle(self):
        return self.is_obstacle_flag
    #if a tile button is pressed, on_press_callback() function in GameScreen class is called
    def on_press(self):
        self.game.on_press_callback(self.tile_index)
    #returns index of tile
    def get_tile_index(self):
        return self.tile_index


class GameScreen(Screen):
    def __init__(self, **kw):
        super().__init__(**kw)
        self.my_root_widget = None
        self.tile_list = None
        self.no_rows = None
        self.no_cols = None
        self.beg=None
        self.finish=None

 #initialises the object, beg and finish are 1-d position values
    def init(self, length, width,beg,finish):
        self.no_rows = length
        self.no_cols = width
        self.beg=beg
        self.finish=finish
        # create root layout as a box with vertical orientation
        self.my_root_widget = BoxLayout(spacing=10, padding=100, orientation='vertical')
        #create a sub layout grid of size length and width as entered by user
        self.table = GridLayout(cols=self.no_cols,rows=self.no_rows)
        #add table to root
```

```
        self.my_root_widget.add_widget(self.table)
        #create a button called play_button
        self.play_button = Button(text="Play",size_hint=(.2,.2),pos_hint={'x':0.4,'y':0})
        #on pressing it, start_game() is called
        self.play_button.bind(on_press=self.start_game)
        # create list of tiles (objects of Tile class)
        self.tile_list = [Tile(self,i) for i in range(self.no_cols * self.no_rows)]
        # populate the table layout
        for tile in self.tile_list:
            self.table.add_widget(tile)
        #adds button to root layout
        self.my_root_widget.add_widget(self.play_button)
        self.add_widget(self.my_root_widget)
        #set beginning tile unclickable
        self.tile_list[beg].disabled=True
        self.tile_list[beg].background_disabled_normal='player.png'
        #set finishing tile unclickable
        self.tile_list[finish].disabled=True
        self.tile_list[finish].background_disabled_normal='finish_line.png'
       #when a tile is pressed, if there is a barrier, it sets flag to false and changes the icon,
       #else sets flag to True and changes icon to barrier
    def on_press_callback(self, index):
        flag=self.tile_list[index].is_obstacle()
        if flag:
            self.tile_list[index].set_obstacles(False)
            self.tile_list[index].background_normal=
'atlas://data/images/defaulttheme/button_disabled'
        else:
            self.tile_list[index].set_obstacles(True)
            self.tile_list[index].background_normal='barrier.png'
```

```
#on pressing the play button this function is called. This runs the A star search to find the path to goal.
def start_game(self, _):
    #the play button is removed
    self.my_root_widget.remove_widget(self.play_button)
    barriers = []
    no_rows=self.no_rows
    no_cols=self.no_cols
    #change 1d index to 2d index
    start=(int(self.beg/no_cols),self.beg%no_cols)
    end=(int(self.finish/no_cols),self.finish%no_cols)
    #disables all tiles making them unclickable, while keeping barrier icons
    for i in range (no_rows):
        for j in range (no_cols):
            index=i*no_cols+j
            self.tile_list[index].disabled=True

            if self.tile_list[index].is_obstacle():
                barriers.append((i,j))
                self.tile_list[index].background_disabled_normal='barrier.png'
    #an object of class AstarGraph is created
    graph = AStarGraph(barriers,no_rows,no_cols)
    G = {} #Actual movement cost to each position from the start position
    F = {} #Estimated movement cost of start to end going via this position

    #Initialize starting values
    G[start] = 0
    F[start] = graph.heuristic(start, end)

    closedVertices = set()
    openVertices = set([start])
    cameFrom = {}
```

```
while len(openVertices) > 0:
        #Get the vertex in the open list with the lowest F score
    current = None
    currentFscore = None
    for pos in openVertices:
            if current is None or F[pos] < currentFscore:
                    currentFscore = F[pos]
                    current = pos

        #Check if we have reached the goal
    if current == end:
          #Retrace our route backward
            path = [current]
            while current in cameFrom:
                    current = cameFrom[current]
                    path.append(current)
            path.reverse()
            print(path)
            #remove goal node from path
            path.pop()
            print(path)
            #sets all tiles in path except goal with player icon
            for node in path:
                    self.tile_list[node[0]*no_cols+node[1]].background_disabled_normal='p
                    layer.png'
            #changes goal tile to denote player has reached
            self.tile_list[self.finish].background_disabled_normal='finish_player.png'
            #create Play Again button, which calls again_game() when pressed
            self.play_again_button = Button(text="Play
           Again",size_hint=(.2,.2),pos_hint={'x':0.4,'y':0})
            self.play_again_button.bind(on_press=self.again_game)
            self.my_root_widget.add_widget(self.play_again_button)
```

```
        #create Logout button which calls logout() when pressed
        self.logout_button
        =Button(text="Logout",size_hint=(.2,.2),pos_hint={'x':0.4,'y':0})
        self.logout_button.bind(on_press=self.logout)
        self.my_root_widget.add_widget(self.logout_button)
        return

      #Mark the current vertex as closed
    openVertices.remove(current)
    closedVertices.add(current)

      #Update scores for vertices near the current position
    for neighbour in graph.get_vertex_neighbours(current):
        if neighbour in closedVertices:
            continue #We have already processed this node exhaustively
        candidateG = G[current] + graph.move_cost(current, neighbour)
        if neighbour not in openVertices:
            openVertices.add(neighbour) #Discovered a new vertex
        elif candidateG >= G[neighbour]:
            continue #This G score is worse than previously found

            #Adopt this G score
        cameFrom[neighbour] = current
        G[neighbour] = candidateG
        H = graph.heuristic(neighbour, end)
        F[neighbour] = G[neighbour] + H

  raise RuntimeError("A* failed to find a solution")
#If play again is clicked, remove the root widget of current screen, and set current
#screen to intro
def again_game(self,_):
    self.remove_widget(self.my_root_widget)
```

```
        sm.current = 'intro'
    #If logout is clicked, set current screen to login screen

    def logout(self,_):
        sm.current = "login"
```

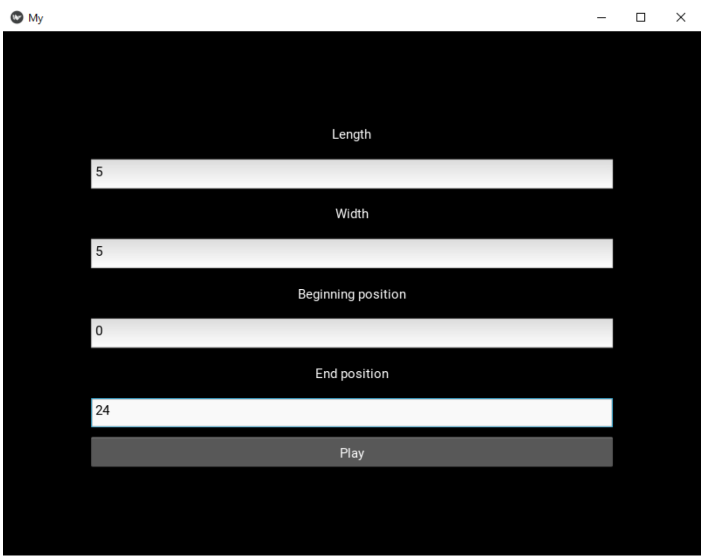
V. Other functions in main.py

```
#invalidLogin() is called to create a popup and open it on invalid login
def invalidLogin():
    pop = Popup(title='Invalid Login',
                content=Label(text='Invalid username or password.'),
                size_hint=(None, None), size=(400, 400))
    pop.open()

#invalidForm() is called to create a popup and open it on invalid input
def invalidForm():
    pop = Popup(title='Invalid Form',
                content=Label(text='Please fill in all inputs with valid information.'),
                size_hint=(None, None), size=(400, 400))

    pop.open()
#create an empty class
class WindowManager(ScreenManager):
    pass
#loads our kv file
kv = Builder.load_file("my.kv")

#create an instance of class WindowManager
sm = WindowManager()
#create an object of class DataBase and pass filename “users.txt”
db = DataBase("users.txt")
```

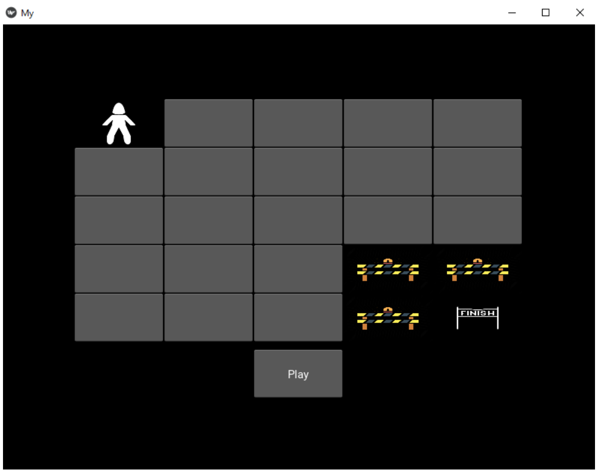
```
#creates a list of screens

screens = [LoginWindow(name='login'), CreateAccountWindow(name='create'),IntroScreen(name='intro'),GameScreen(name='game')]

#adds all the screens to sm object which we will use to navigate between different screens

for screen in screens:

    sm.add_widget(screen)

#sets current screen to login

sm.current = 'login'


class MyApp(App):

    def build(self):

        return sm



if __name__ == '__main__':

    MyApp().run()
```

10.Issues

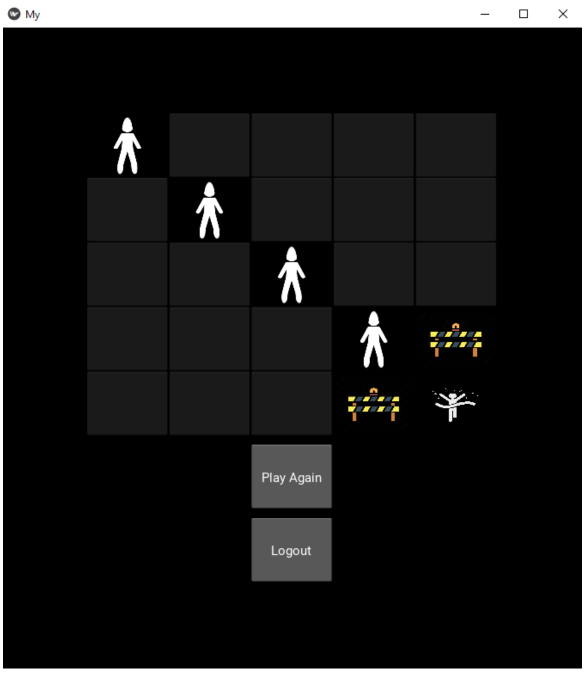
I. Due to the usage of a high finite move cost for barrier, if the goal is blocked of by barriers, the search continues over the barriers.



For a board of 5X5 and start= (0,0) and end= (4,4)



Barriers are set at (3,3), (3,4) and (4,3).



The player goes through barrier at (3,3) to reach goal.

II. Storing user details in a text file is not secure.

11.Conclusion

In games we often want to find paths from one location to another. We're not only trying to find the shortest distance; we also want to take into account travel time.

To find this path we use a graph search algorithm, which works when the map is represented as a graph. **A*** is a popular choice for graph search.

This app allows us to easily create an interactable grid of custom size, with variable start and goal positions. By interactively placing barriers on this map we can find shortest path for multiple combinations of input easily.