Version 1

Base version:

* This version has a GUI with 3 buttons, (My mazes, quit and Make Maze)
* My mazes does nothing, quit button halts the program, clicking on “Make maze” opens a menu allowing to generate maze
* Maze can be generated from size 10-100, changes height and width at the same time
* Using “non-perfect” generates a fully black canvas
* Using “perfect” generates a maze using Prim’s algorithm
* When the maze has been generated, there is a button “Quit” and “regenerate maze”, both of them work
* Regenerate maze regenerates maze of the same size and uses the same algorithm

001 import tkinter as tk

002 import random

003

004 global maze

005 maze = []

006

007 class Maze:

008 def \_\_init\_\_(self, height, width):

009 self.height = height

010 self.width = width

011 self.maze = [[0] \* width for \_ in range(height)]

012

013 def generate(self):

014 self.maze = prims\_algorithm(self.height, self.width)

015

016 def render(self, canvas):

017 for i in range(self.height):

018 for j in range(self.width):

019 color = "black" if self.maze[i][j] == 0 else "white"

020 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

021

022 def clear\_root(root):

023 for widget in root.winfo\_children():

024 widget.destroy()

025

026 def main\_menu():

027 root = tk.Tk()

028 root.title("Maze Program")

029 root.geometry("300x200")

030 tk.Button(root, text="Make Maze", command=lambda: make\_maze\_menu(root)).pack()

031 tk.Button(root, text="My Mazes", command=my\_mazes).pack()

032 tk.Button(root, text="Quit", command=root.quit).pack()

033 root.mainloop()

034

035 def make\_maze\_menu(root):

036 clear\_root(root)

037

038 height = tk.Spinbox(root, from\_=10, to=100)

039 width = tk.Spinbox(root, from\_=10, to=100)

040 density = tk.Scale(root, from\_=1, to=10, orient="horizontal")

041 maze\_type = tk.StringVar(root)

042 maze\_type.set("Perfect")

043

044 tk.OptionMenu(root, maze\_type, "Perfect", "Non-Perfect").pack()

045 size\_slider = tk.Scale(root, from\_=10, to=100, orient="horizontal", label="Maze Size")

046 size\_slider.pack()

047 tk.Button(root, text="Generate Maze", command=lambda: generate\_maze(root, int(size\_slider.get()), int(size\_slider.get()), int(density.get()), maze\_type.get())).pack()

048

049 def my\_mazes():

050 pass

051

052 def prims\_algorithm(height, width):

053 local\_maze = [[0] \* width for \_ in range(height)]

054 visited = set()

055 walls = set()

056 start = (0, 0)

057 visited.add(start)

058 local\_maze[start[0]][start[1]] = 1

059 walls.update({(0, 1), (1, 0)})

060

061 while walls:

062 wall = random.choice(list(walls))

063 x, y = wall

064

065 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

066

067 if len(neighbors) == 1:

068 nx, ny = neighbors[0]

069 local\_maze[x][y] = 1

070 visited.add((x, y))

071

072 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

073 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

074 walls.add((dx, dy))

075

076 walls.remove(wall)

077

078 return local\_maze

079

080 def generate\_maze(root, height, width, density, maze\_type):

081 global maze

082 maze = Maze(height, width)

083 clear\_root(root)

084 if maze\_type == "Perfect":

085 maze.generate()

086

087 canvas\_width = width \* 10

088 canvas\_height = height \* 10

089 root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

090 canvas = tk.Canvas(root, width=canvas\_width, height=canvas\_height)

091 root.update\_idletasks()

092 x\_pos = (root.winfo\_width() - canvas\_width) // 2

093 y\_pos = (root.winfo\_height() - canvas\_height) // 2

094 canvas.place(x=x\_pos, y=y\_pos)

095 frame = tk.Frame(root)

096 frame.place(x=x\_pos, y=y\_pos - 30)

097

098 quit\_button = tk.Button(frame, text="Quit", command=root.quit)

099 quit\_button.pack(side=tk.TOP)

100 regenerate\_button = tk.Button(frame, text="Regenerate Maze", command=lambda: generate\_maze(root, height, width, density, maze\_type))

101 regenerate\_button.pack(side=tk.TOP)

102 canvas = tk.Canvas(frame, width=canvas\_width, height=canvas\_height)

103 canvas.pack(side=tk.BOTTOM)

104

105 maze.render(canvas)

106

107 if \_\_name\_\_ == "\_\_main\_\_":

108 main\_menu()

Version 2

Changes made:

* “My mazes” now opens and has a “Go back” button allowing you to go back to the main menu
* When clicking on “Generate maze” there is also now an option to “Go back”
* The maze is no longer centered , which I need to fix in the next version
* When the maze has been generated, the buttons “Quit” and “Regenerate maze” are now in the top-left corner, which I also need to fix in the next version

001 import tkinter as tk

002 import random

003

004 global maze

005 maze = []

006

007 class Maze:

008 def \_\_init\_\_(self, height, width):

009 self.height = height

010 self.width = width

011 self.maze = [[0] \* width for \_ in range(height)]

012

013 def generate(self):

014 self.maze = prims\_algorithm(self.height, self.width)

015

016 def render(self, canvas):

017 for i in range(self.height):

018 for j in range(self.width):

019 color = "black" if self.maze[i][j] == 0 else "white"

020 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

021

022 class MazeApplication:

023 def \_\_init\_\_(self, root):

024 self.root = root

025 self.root.title("Maze Program")

026 self.main\_menu()

027

028 def clear\_root(self):

029 for widget in self.root.winfo\_children():

030 widget.destroy()

031

032 def main\_menu(self):

033 self.clear\_root()

034 self.root.geometry("300x200")

035 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

036 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

037 self.add\_quit\_button()

038

039 def make\_maze\_menu(self):

040 self.clear\_root()

041 height = tk.Spinbox(self.root, from\_=10, to=100)

042 width = tk.Spinbox(self.root, from\_=10, to=100)

043 density = tk.Scale(self.root, from\_=1, to=10, orient="horizontal")

044 maze\_type = tk.StringVar(self.root)

045 maze\_type.set("Perfect")

046 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect").pack()

047 size\_slider = tk.Scale(self.root, from\_=10, to=100, orient="horizontal", label="Maze Size")

048 size\_slider.pack()

049 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(size\_slider.get()), int(size\_slider.get()), int(density.get()), maze\_type.get())).pack()

050 self.add\_go\_back\_button()

051

052 def my\_mazes(self):

053 self.clear\_root()

054 self.add\_go\_back\_button()

055

056 def go\_back(self):

057 self.main\_menu()

058

059 def quit\_app(self):

060 self.root.quit()

061

062 def add\_go\_back\_button(self):

063 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

064

065 def add\_quit\_button(self):

066 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

067

068 def generate\_maze(self, height, width, density, maze\_type):

069 maze = Maze(height, width)

070 self.clear\_root()

071 if maze\_type == "Perfect":

072 maze.generate()

073 canvas\_width = width \* 10

074 canvas\_height = height \* 10

075 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

076 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

077 self.root.update\_idletasks()

078 x\_pos = (self.root.winfo\_width() - canvas\_width) // 2

079 y\_pos = (self.root.winfo\_height() - canvas\_height) // 2

080 canvas.place(x=x\_pos, y=y\_pos)

081 frame = tk.Frame(self.root)

082 frame.place(x=x\_pos, y=y\_pos - 30)

083 quit\_button = tk.Button(frame, text="Quit", command=self.quit\_app)

084 quit\_button.pack(side=tk.TOP)

085 regenerate\_button = tk.Button(frame, text="Regenerate Maze", command=lambda: self.generate\_maze(height, width, density, maze\_type))

086 regenerate\_button.pack(side=tk.TOP)

087 canvas.pack(side=tk.BOTTOM)

088 maze.render(canvas)

089

090 def prims\_algorithm(height, width):

091 local\_maze = [[0] \* width for \_ in range(height)]

092 visited = set()

093 walls = set()

094 start = (0, 0)

095 visited.add(start)

096 local\_maze[start[0]][start[1]] = 1

097 walls.update({(0, 1), (1, 0)})

098

099 while walls:

100 wall = random.choice(list(walls))

101 x, y = wall

102

103 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

104

105 if len(neighbors) == 1:

106 nx, ny = neighbors[0]

107 local\_maze[x][y] = 1

108 visited.add((x, y))

109

110 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

111 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

112 walls.add((dx, dy))

113

114 walls.remove(wall)

115

116 return local\_maze

117

118 if \_\_name\_\_ == "\_\_main\_\_":

119 root = tk.Tk()

120 app = MazeApplication(root)

121 root.mainloop()

Version 3

Changes made:

* When clicking on “My mazes” there is now a quit button
* Implemented Recursive Backtracker algorithm to generate maze when the “perfect” option is selected
* Added a button “Find path” that doesn’t work 95% of the time when using recursive backtracker, which I will need to fix in the next version
* Fixed the maze not being centered
* The buttons are still anchored to the top left corner

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004

005 class Maze:

006 def \_\_init\_\_(self, height, width):

007 self.height = height

008 self.width = width

009 self.maze = [[0] \* width for \_ in range(height)]

010

011 def generate(self):

012 self.maze = prims\_algorithm(self.height, self.width)

013

014 def render(self, canvas):

015 for i in range(self.height):

016 for j in range(self.width):

017 color = "black" if self.maze[i][j] == 0 else "white"

018 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

019

020 class MazeApplication:

021 def \_\_init\_\_(self, root):

022 self.root = root

023 self.root.title("Maze Program")

024 self.main\_menu()

025 self.maze\_type = "Perfect"

026

027 @staticmethod

028 def a\_star\_search(maze, start, end):

029 def heuristic(a, b):

030 return abs(a[0] - b[0]) + abs(a[1] - b[1])

031

032 def get\_neighbors(pos):

033 neighbors = []

034 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

035 x, y = pos[0] + dx, pos[1] + dy

036 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

037 neighbors.append((x, y))

038 return neighbors

039

040 frontier = PriorityQueue()

041 frontier.put((0, start))

042 came\_from = {}

043 cost\_so\_far = {}

044 came\_from[start] = None

045 cost\_so\_far[start] = 0

046

047 while not frontier.empty():

048 current = frontier.get()[1]

049

050 if current == end:

051 break

052

053 for next in get\_neighbors(current):

054 new\_cost = cost\_so\_far[current] + 1

055 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

056 cost\_so\_far[next] = new\_cost

057 priority = new\_cost + heuristic(end, next)

058 frontier.put((priority, next))

059 came\_from[next] = current

060

061 current = end

062 path = []

063 while current != start:

064 if current not in came\_from:

065 return None

066 path.append(current)

067 current = came\_from[current]

068 path.append(start)

069 path.reverse()

070 return path

071

072 def find\_path(self, height, width):

073 start = (0, 0)

074 end = (height - 1, width - 1)

075 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

076 if path is not None:

077 self.show\_path(path)

078 else:

079 print("No path found!")

080

081 def show\_path(self, path):

082 if path:

083 for (x, y) in path:

084 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

085 self.canvas.update()

086 else:

087 print("No path to show.")

088

089 def update\_maze\_type(self, maze\_type):

090 self.maze\_type = maze\_type

091

092 def clear\_root(self):

093 for widget in self.root.winfo\_children():

094 widget.destroy()

095

096

097 def main\_menu(self):

098 self.clear\_root()

099 self.root.geometry("300x200")

100 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

101 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

102 self.add\_quit\_button()

103

104 def make\_maze\_menu(self):

105 self.clear\_root()

106 height = tk.Spinbox(self.root, from\_=10, to=100)

107 width = tk.Spinbox(self.root, from\_=10, to=100)

108 density = tk.Scale(self.root, from\_=1, to=10, orient="horizontal")

109 maze\_type = tk.StringVar(self.root)

110 maze\_type.set("Perfect")

111 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

112 size\_slider = tk.Scale(self.root, from\_=10, to=100, orient="horizontal", label="Maze Size")

113 size\_slider.pack()

114 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(size\_slider.get()), int(size\_slider.get()), int(density.get()), maze\_type.get())).pack()

115 self.add\_go\_back\_button()

116 self.add\_quit\_button()

117

118 def my\_mazes(self):

119 self.clear\_root()

120 self.add\_go\_back\_button()

121 self.add\_quit\_button()

122

123 def go\_back(self):

124 self.main\_menu()

125

126 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

127 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

128

129 def quit\_app(self):

130 self.root.quit()

131

132 def add\_go\_back\_button(self):

133 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

134

135 def add\_quit\_button(self):

136 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

137

138 def generate\_maze(self, height, width, density, maze\_type):

139 self.maze = Maze(height, width)

140 self.clear\_root()

141 if maze\_type == "Perfect":

142 self.maze.maze = recursive\_backtracker(height, width)

143 else:

144 self.maze.maze = prims\_algorithm(height, width)

145

146 canvas\_width = width \* 10

147 canvas\_height = height \* 10

148 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

149

150 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

151 self.canvas = canvas

152 self.root.update\_idletasks()

153 x\_pos = (self.root.winfo\_width() - canvas\_width) // 2

154 y\_pos = (self.root.winfo\_height() - canvas\_height) // 2 - 30

155

156 button\_frame = tk.Frame(self.root)

157 button\_frame.place(x=x\_pos, y=max(y\_pos, 0))

158

159 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

160 quit\_button.pack()

161

162 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(height, width, density, maze\_type))

163 regenerate\_button.pack()

164

165 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

166 find\_path\_button.pack()

167

168 self.add\_go\_back\_button\_generate\_maze(button\_frame)

169

170 canvas.pack()

171 self.maze.render(canvas)

172

173

174 def recursive\_backtracker(height, width):

175 def carve\_passage\_from(cx, cy, grid):

176 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

177 random.shuffle(directions)

178

179 for (nx, ny) in directions:

180 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

181 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

182 grid[nx][ny] = 1

183 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

184 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

185

186 maze = [[0] \* width for \_ in range(height)]

187 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

188 maze[start\_x][start\_y] = 1

189 carve\_passage\_from(start\_x, start\_y, maze)

190 return maze

191

192

193 def prims\_algorithm(height, width):

194 local\_maze = [[0] \* width for \_ in range(height)]

195 visited = set()

196 walls = set()

197 start = (0, 0)

198 visited.add(start)

199 local\_maze[start[0]][start[1]] = 1

200 walls.update({(0, 1), (1, 0)})

201

202 while walls:

203 wall = random.choice(list(walls))

204 x, y = wall

205

206 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

207

208 if len(neighbors) == 1:

209 nx, ny = neighbors[0]

210 local\_maze[x][y] = 1

211 visited.add((x, y))

212

213 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

214 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

215 walls.add((dx, dy))

216

217 walls.remove(wall)

218

219 return local\_maze

220

221 if \_\_name\_\_ == "\_\_main\_\_":

222 root = tk.Tk()

223 app = MazeApplication(root)

224 root.mainloop()

Version 4

Changes made:

* Added 3 sliders for “Maze height”, “Maze width” from 10-100, and “Maze density” from 1 to 10 which does nothing
* This allows for rectangle mazes to be generated
* Finding the path still doesn’t work properly with Recursive Backtracker, which I will try fixing in the next version
* When the maze has been generated the buttons are still in the top left corner, which I will fix in a later version

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004

005 class Maze:

006 def \_\_init\_\_(self, height, width):

007 self.height = height

008 self.width = width

009 self.maze = [[0] \* width for \_ in range(height)]

010

011 def generate(self):

012 self.maze = prims\_algorithm(self.height, self.width)

013

014 def render(self, canvas):

015 for i in range(self.height):

016 for j in range(self.width):

017 color = "black" if self.maze[i][j] == 0 else "white"

018 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

019

020 class MazeApplication:

021 def \_\_init\_\_(self, root):

022 self.root = root

023 self.root.title("Maze Program")

024 self.main\_menu()

025 self.maze\_type = "Perfect"

026

027 @staticmethod

028 def a\_star\_search(maze, start, end):

029 def heuristic(a, b):

030 return abs(a[0] - b[0]) + abs(a[1] - b[1])

031

032 def get\_neighbors(pos):

033 neighbors = []

034 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

035 x, y = pos[0] + dx, pos[1] + dy

036 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

037 neighbors.append((x, y))

038 return neighbors

039

040 frontier = PriorityQueue()

041 frontier.put((0, start))

042 came\_from = {}

043 cost\_so\_far = {}

044 came\_from[start] = None

045 cost\_so\_far[start] = 0

046

047 while not frontier.empty():

048 current = frontier.get()[1]

049

050 if current == end:

051 break

052

053 for next in get\_neighbors(current):

054 new\_cost = cost\_so\_far[current] + 1

055 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

056 cost\_so\_far[next] = new\_cost

057 priority = new\_cost + heuristic(end, next)

058 frontier.put((priority, next))

059 came\_from[next] = current

060

061 current = end

062 path = []

063 while current != start:

064 if current not in came\_from:

065 return None

066 path.append(current)

067 current = came\_from[current]

068 path.append(start)

069 path.reverse()

070 return path

071

072 def find\_path(self, height, width):

073 start = (0, 0)

074 end = (height - 1, width - 1)

075 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

076 if path is not None:

077 self.show\_path(path)

078 else:

079 print("No path found!")

080

081 def show\_path(self, path):

082 if path:

083 for (x, y) in path:

084 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

085 self.canvas.update()

086 else:

087 print("No path to show.")

088

089 def update\_maze\_type(self, maze\_type):

090 self.maze\_type = maze\_type

091

092 def clear\_root(self):

093 for widget in self.root.winfo\_children():

094 widget.destroy()

095

096

097 def main\_menu(self):

098 self.clear\_root()

099 self.root.geometry("300x200")

100 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

101 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

102 self.add\_quit\_button()

103

104 def make\_maze\_menu(self):

105 self.clear\_root()

106 self.root.geometry("300x400")

107

108 height\_label = tk.Label(self.root, text="Maze Height:")

109 height\_label.pack()

110 height\_slider = tk.Scale(self.root, from\_=10, to=100, orient="horizontal", label="10 to 100 cells")

111 height\_slider.pack()

112

113 width\_label = tk.Label(self.root, text="Maze Width:")

114 width\_label.pack()

115 width\_slider = tk.Scale(self.root, from\_=10, to=100, orient="horizontal", label="10 to 100 cells")

116 width\_slider.pack()

117

118 density\_label = tk.Label(self.root, text="Maze Density:")

119 density\_label.pack()

120 density = tk.Scale(self.root, from\_=1, to=10, orient="horizontal")

121 density.pack()

122

123 maze\_type = tk.StringVar(self.root)

124 maze\_type.set("Perfect")

125 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

126

127 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(height\_slider.get()), int(width\_slider.get()), int(density.get()), maze\_type.get())).pack()

128

129 self.add\_go\_back\_button()

130 self.add\_quit\_button()

131

132 def my\_mazes(self):

133 self.clear\_root()

134 self.add\_go\_back\_button()

135 self.add\_quit\_button()

136

137 def go\_back(self):

138 self.main\_menu()

139

140 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

141 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

142

143 def quit\_app(self):

144 self.root.quit()

145

146 def add\_go\_back\_button(self):

147 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

148

149 def add\_quit\_button(self):

150 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

151

152 def generate\_maze(self, height, width, density, maze\_type):

153 self.maze = Maze(height, width)

154 self.clear\_root()

155 if maze\_type == "Perfect":

156 self.maze.maze = recursive\_backtracker(height, width)

157 else:

158 self.maze.maze = prims\_algorithm(height, width)

159

160 canvas\_width = width \* 10

161 canvas\_height = height \* 10

162 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

163

164 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

165 self.canvas = canvas

166 self.root.update\_idletasks()

167 x\_pos = (self.root.winfo\_width() - canvas\_width) // 2

168 y\_pos = (self.root.winfo\_height() - canvas\_height) // 2 - 30

169

170 button\_frame = tk.Frame(self.root)

171 button\_frame.place(x=x\_pos, y=max(y\_pos, 0))

172

173 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

174 quit\_button.pack()

175

176 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(height, width, density, maze\_type))

177 regenerate\_button.pack()

178

179 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

180 find\_path\_button.pack()

181

182 self.add\_go\_back\_button\_generate\_maze(button\_frame)

183

184 canvas.pack()

185 self.maze.render(canvas)

186

187

188 def recursive\_backtracker(height, width):

189 def carve\_passage\_from(cx, cy, grid):

190 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

191 random.shuffle(directions)

192

193 for (nx, ny) in directions:

194 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

195 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

196 grid[nx][ny] = 1

197 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

198 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

199

200 maze = [[0] \* width for \_ in range(height)]

201 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

202 maze[start\_x][start\_y] = 1

203 carve\_passage\_from(start\_x, start\_y, maze)

204 return maze

205

206

207 def prims\_algorithm(height, width):

208 local\_maze = [[0] \* width for \_ in range(height)]

209 visited = set()

210 walls = set()

211 start = (0, 0)

212 visited.add(start)

213 local\_maze[start[0]][start[1]] = 1

214 walls.update({(0, 1), (1, 0)})

215

216 while walls:

217 wall = random.choice(list(walls))

218 x, y = wall

219

220 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

221

222 if len(neighbors) == 1:

223 nx, ny = neighbors[0]

224 local\_maze[x][y] = 1

225 visited.add((x, y))

226

227 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

228 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

229 walls.add((dx, dy))

230

231 walls.remove(wall)

232

233 return local\_maze

234

235 if \_\_name\_\_ == "\_\_main\_\_":

236 root = tk.Tk()

237 app = MazeApplication(root)

238 root.mainloop()

Version 5

Changes made:

* Reduced the amount of times the maze cannot be generated by increasing the lower bound on the height and width from 10 to 15 (15-100 now)
* Find path has been fixed, and it now correctly shows the path from top left to bottom right corner
* The buttons are now correctly horizontally centered below the maze instead of being anchored to the top left corner

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004

005 class Maze:

006 def \_\_init\_\_(self, height, width):

007 self.height = height

008 self.width = width

009 self.maze = [[0] \* width for \_ in range(height)]

010

011 def generate(self):

012 self.maze = prims\_algorithm(self.height, self.width)

013

014 def render(self, canvas):

015 for i in range(self.height):

016 for j in range(self.width):

017 color = "black" if self.maze[i][j] == 0 else "white"

018 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

019

020 class MazeApplication:

021 def \_\_init\_\_(self, root):

022 self.root = root

023 self.root.title("Maze Program")

024 self.main\_menu()

025 self.maze\_type = "Perfect"

026

027 @staticmethod

028 def a\_star\_search(maze, start, end):

029 def heuristic(a, b):

030 return abs(a[0] - b[0]) + abs(a[1] - b[1])

031

032 def get\_neighbors(pos):

033 neighbors = []

034 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

035 x, y = pos[0] + dx, pos[1] + dy

036 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

037 neighbors.append((x, y))

038 return neighbors

039

040 frontier = PriorityQueue()

041 frontier.put((0, start))

042 came\_from = {}

043 cost\_so\_far = {}

044 came\_from[start] = None

045 cost\_so\_far[start] = 0

046

047 while not frontier.empty():

048 current = frontier.get()[1]

049

050 if current == end:

051 break

052

053 for next in get\_neighbors(current):

054 new\_cost = cost\_so\_far[current] + 1

055 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

056 cost\_so\_far[next] = new\_cost

057 priority = new\_cost + heuristic(end, next)

058 frontier.put((priority, next))

059 came\_from[next] = current

060

061 current = end

062 path = []

063 while current != start:

064 if current not in came\_from:

065 return None

066 path.append(current)

067 current = came\_from[current]

068 path.append(start)

069 path.reverse()

070 return path

071

072 def find\_path(self, height, width):

073 start = (0, 0)

074 end = (height - 1, width - 1)

075 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

076 if path is not None:

077 self.show\_path(path)

078 else:

079 print("No path found!")

080

081 def show\_path(self, path):

082 if path:

083 for (x, y) in path:

084 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

085 self.canvas.update()

086 else:

087 print("No path to show.")

088

089 def update\_maze\_type(self, maze\_type):

090 self.maze\_type = maze\_type

091

092 def set\_window\_size(self, width, height):

093 canvas\_width = width \* 10

094 canvas\_height = height \* 10

095 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

096

097 def clear\_root(self):

098 for widget in self.root.winfo\_children():

099 widget.destroy()

100

101

102 def main\_menu(self):

103 self.clear\_root()

104 self.root.geometry("300x200")

105 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

106 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

107 self.add\_quit\_button()

108

109 def make\_maze\_menu(self):

110 self.clear\_root()

111 self.root.geometry("300x400")

112

113 height\_label = tk.Label(self.root, text="Maze Height:")

114 height\_label.pack()

115 self.height\_slider = tk.Scale(self.root, from\_=15, to=100, orient="horizontal", label="15 to 100 cells")

116 self.height\_slider.pack()

117

118 width\_label = tk.Label(self.root, text="Maze Width:")

119 width\_label.pack()

120 self.width\_slider = tk.Scale(self.root, from\_=15, to=100, orient="horizontal", label="15 to 100 cells")

121 self.width\_slider.pack()

122

123 maze\_type = tk.StringVar(self.root)

124 maze\_type.set("Perfect")

125 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

126

127 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

128

129 self.add\_go\_back\_button()

130 self.add\_quit\_button()

131

132 def update\_size(self, event=None):

133 height = int(self.height\_slider.get())

134 width = int(self.width\_slider.get())

135 self.set\_window\_size(width, height)

136

137 def my\_mazes(self):

138 self.clear\_root()

139 self.add\_go\_back\_button()

140 self.add\_quit\_button()

141

142 def go\_back(self):

143 self.main\_menu()

144

145 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

146 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

147

148 def quit\_app(self):

149 self.root.quit()

150

151 def add\_go\_back\_button(self):

152 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

153

154 def add\_quit\_button(self):

155 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

156

157 def display\_maze(self, height, width):

158 self.clear\_root()

159 canvas\_width = width \* 10

160 canvas\_height = height \* 10

161

162 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

163 self.canvas = canvas

164

165 canvas.pack(side=tk.TOP)

166

167 self.maze.render(canvas)

168

169 button\_frame = tk.Frame(self.root)

170 button\_frame.pack(after=canvas)

171

172 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

173 quit\_button.pack(side=tk.LEFT)

174

175 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

176 regenerate\_button.pack(side=tk.LEFT)

177

178 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

179 find\_path\_button.pack(side=tk.LEFT)

180

181 self.add\_go\_back\_button\_generate\_maze(button\_frame)

182

183 def generate\_maze(self, height, width, maze\_type):

184 self.current\_height = height

185 self.current\_width = width

186 self.current\_maze\_type = maze\_type

187

188 if not hasattr(self, 'initial\_size\_set') or not self.initial\_size\_set:

189 self.set\_window\_size(width, height)

190 self.initial\_size\_set = True

191

192 valid\_maze = False

193 maze\_generation\_attempts = 0

194 while not valid\_maze:

195 maze\_generation\_attempts += 1

196 self.maze = Maze(height, width)

197 if maze\_type == "Perfect":

198 self.maze.maze = recursive\_backtracker(height, width)

199 else:

200 self.maze.maze = prims\_algorithm(height, width)

201

202 start = (0, 0)

203 end = (height - 1, width - 1)

204 if MazeApplication.a\_star\_search(self.maze.maze, start, end):

205 valid\_maze = True

206 else:

207 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

208

209 if maze\_generation\_attempts > 100:

210 print("Too many attempts to generate a valid maze. Stopping the process.")

211 break

212

213 if valid\_maze:

214 self.display\_maze(height, width)

215 else:

216 print("Could not generate a valid maze. Please try again.")

217

218

219 def recursive\_backtracker(height, width):

220 def carve\_passage\_from(cx, cy, grid):

221 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

222 random.shuffle(directions)

223

224 for (nx, ny) in directions:

225 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

226 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

227 grid[nx][ny] = 1

228 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

229 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

230

231 maze = [[0] \* width for \_ in range(height)]

232 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

233 maze[start\_x][start\_y] = 1

234 carve\_passage\_from(start\_x, start\_y, maze)

235 return maze

236

237

238 def prims\_algorithm(height, width):

239 local\_maze = [[0] \* width for \_ in range(height)]

240 visited = set()

241 walls = set()

242 start = (0, 0)

243 visited.add(start)

244 local\_maze[start[0]][start[1]] = 1

245 walls.update({(0, 1), (1, 0)})

246

247 while walls:

248 wall = random.choice(list(walls))

249 x, y = wall

250

251 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

252

253 if len(neighbors) == 1:

254 nx, ny = neighbors[0]

255 local\_maze[x][y] = 1

256 visited.add((x, y))

257

258 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

259 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

260 walls.add((dx, dy))

261

262 walls.remove(wall)

263

264 return local\_maze

265

266 if \_\_name\_\_ == "\_\_main\_\_":

267 root = tk.Tk()

268 app = MazeApplication(root)

269 root.mainloop()

Version 6

Changes made:

* Reduced the amount of times the program crashes when trying to generate a maze by lowering the higher bound on how big the maze can be (15 to 75 cells now)

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004

005 class Maze:

006 def \_\_init\_\_(self, height, width):

007 self.height = height

008 self.width = width

009 self.maze = [[0] \* width for \_ in range(height)]

010

011 def generate(self):

012 self.maze = prims\_algorithm(self.height, self.width)

013

014 def render(self, canvas):

015 for i in range(self.height):

016 for j in range(self.width):

017 color = "black" if self.maze[i][j] == 0 else "white"

018 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

019

020 class MazeApplication:

021 def \_\_init\_\_(self, root):

022 self.root = root

023 self.root.title("Maze Program")

024 self.main\_menu()

025 self.maze\_type = "Perfect"

026

027 @staticmethod

028 def a\_star\_search(maze, start, end):

029 def heuristic(a, b):

030 return abs(a[0] - b[0]) + abs(a[1] - b[1])

031

032 def get\_neighbors(pos):

033 neighbors = []

034 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

035 x, y = pos[0] + dx, pos[1] + dy

036 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

037 neighbors.append((x, y))

038 return neighbors

039

040 frontier = PriorityQueue()

041 frontier.put((0, start))

042 came\_from = {}

043 cost\_so\_far = {}

044 came\_from[start] = None

045 cost\_so\_far[start] = 0

046

047 while not frontier.empty():

048 current = frontier.get()[1]

049

050 if current == end:

051 break

052

053 for next in get\_neighbors(current):

054 new\_cost = cost\_so\_far[current] + 1

055 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

056 cost\_so\_far[next] = new\_cost

057 priority = new\_cost + heuristic(end, next)

058 frontier.put((priority, next))

059 came\_from[next] = current

060

061 current = end

062 path = []

063 while current != start:

064 if current not in came\_from:

065 return None

066 path.append(current)

067 current = came\_from[current]

068 path.append(start)

069 path.reverse()

070 return path

071

072 def find\_path(self, height, width):

073 start = (0, 0)

074 end = (height - 1, width - 1)

075 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

076 if path is not None:

077 self.show\_path(path)

078 else:

079 print("No path found!")

080

081 def show\_path(self, path):

082 if path:

083 for (x, y) in path:

084 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

085 self.canvas.update()

086 else:

087 print("No path to show.")

088

089 def update\_maze\_type(self, maze\_type):

090 self.maze\_type = maze\_type

091

092 def set\_window\_size(self, width, height):

093 canvas\_width = width \* 10

094 canvas\_height = height \* 10

095 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

096

097 def clear\_root(self):

098 for widget in self.root.winfo\_children():

099 widget.destroy()

100

101

102 def main\_menu(self):

103 self.clear\_root()

104 self.root.geometry("300x200")

105 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

106 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

107 self.add\_quit\_button()

108

109 def make\_maze\_menu(self):

110 self.clear\_root()

111 self.root.geometry("300x400")

112

113 height\_label = tk.Label(self.root, text="Maze Height:")

114 height\_label.pack()

115 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

116 self.height\_slider.pack()

117

118 width\_label = tk.Label(self.root, text="Maze Width:")

119 width\_label.pack()

120 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

121 self.width\_slider.pack()

122

123 maze\_type = tk.StringVar(self.root)

124 maze\_type.set("Perfect")

125 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

126

127 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

128

129 self.add\_go\_back\_button()

130 self.add\_quit\_button()

131

132 def update\_size(self, event=None):

133 height = int(self.height\_slider.get())

134 width = int(self.width\_slider.get())

135 self.set\_window\_size(width, height)

136

137 def my\_mazes(self):

138 self.clear\_root()

139 self.add\_go\_back\_button()

140 self.add\_quit\_button()

141

142 def go\_back(self):

143 self.main\_menu()

144

145 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

146 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

147

148 def quit\_app(self):

149 self.root.quit()

150

151 def add\_go\_back\_button(self):

152 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

153

154 def add\_quit\_button(self):

155 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

156

157 def display\_maze(self, height, width):

158 self.clear\_root()

159 canvas\_width = width \* 10

160 canvas\_height = height \* 10

161

162 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

163 self.canvas = canvas

164

165 canvas.pack(side=tk.TOP)

166

167 self.maze.render(canvas)

168

169 button\_frame = tk.Frame(self.root)

170 button\_frame.pack(after=canvas)

171

172 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

173 quit\_button.pack(side=tk.LEFT)

174

175 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

176 regenerate\_button.pack(side=tk.LEFT)

177

178 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

179 find\_path\_button.pack(side=tk.LEFT)

180

181 self.add\_go\_back\_button\_generate\_maze(button\_frame)

182

183 def generate\_maze(self, height, width, maze\_type):

184 self.current\_height = height

185 self.current\_width = width

186 self.current\_maze\_type = maze\_type

187

188 if not hasattr(self, 'initial\_size\_set') or not self.initial\_size\_set:

189 self.set\_window\_size(width, height)

190 self.initial\_size\_set = True

191

192 valid\_maze = False

193 maze\_generation\_attempts = 0

194 while not valid\_maze:

195 maze\_generation\_attempts += 1

196 self.maze = Maze(height, width)

197 if maze\_type == "Perfect":

198 self.maze.maze = recursive\_backtracker(height, width)

199 else:

200 self.maze.maze = prims\_algorithm(height, width)

201

202 start = (0, 0)

203 end = (height - 1, width - 1)

204 if MazeApplication.a\_star\_search(self.maze.maze, start, end):

205 valid\_maze = True

206 else:

207 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

208

209 if maze\_generation\_attempts > 100:

210 print("Too many attempts to generate a valid maze. Stopping the process.")

211 break

212

213 if valid\_maze:

214 self.display\_maze(height, width)

215 else:

216 print("Could not generate a valid maze. Please try again.")

217

218

219 def recursive\_backtracker(height, width):

220 def carve\_passage\_from(cx, cy, grid):

221 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

222 random.shuffle(directions)

223

224 for (nx, ny) in directions:

225 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

226 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

227 grid[nx][ny] = 1

228 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

229 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

230

231 maze = [[0] \* width for \_ in range(height)]

232 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

233 maze[start\_x][start\_y] = 1

234 carve\_passage\_from(start\_x, start\_y, maze)

235 return maze

236

237 def prims\_algorithm(height, width):

238 local\_maze = [[0] \* width for \_ in range(height)]

239 visited = set()

240 walls = set()

241 start = (0, 0)

242 visited.add(start)

243 local\_maze[start[0]][start[1]] = 1

244 walls.update({(0, 1), (1, 0)})

245

246 while walls:

247 wall = random.choice(list(walls))

248 x, y = wall

249

250 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

251

252 if len(neighbors) == 1:

253 nx, ny = neighbors[0]

254 local\_maze[x][y] = 1

255 visited.add((x, y))

256

257 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

258 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

259 walls.add((dx, dy))

260

261 walls.remove(wall)

262

263 return local\_maze

264

265 if \_\_name\_\_ == "\_\_main\_\_":

266 root = tk.Tk()

267 app = MazeApplication(root)

268 root.mainloop()

Version 7

Changes made:

* Added the ability to “randomise” maze which will give it random height and width when generating

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004 import tkinter.messagebox as msgbox

005

006 class Maze:

007 def \_\_init\_\_(self, height, width):

008 self.height = height

009 self.width = width

010 self.maze = [[0] \* width for \_ in range(height)]

011

012 def generate(self):

013 self.maze = prims\_algorithm(self.height, self.width)

014

015 def render(self, canvas):

016 for i in range(self.height):

017 for j in range(self.width):

018 color = "black" if self.maze[i][j] == 0 else "white"

019 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

020

021 class MazeApplication:

022 def \_\_init\_\_(self, root):

023 self.root = root

024 self.root.title("Maze Program")

025 self.main\_menu()

026 self.maze\_type = "Perfect"

027

028 @staticmethod

029 def a\_star\_search(maze, start, end):

030 def heuristic(a, b):

031 return abs(a[0] - b[0]) + abs(a[1] - b[1])

032

033 def get\_neighbors(pos):

034 neighbors = []

035 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

036 x, y = pos[0] + dx, pos[1] + dy

037 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

038 neighbors.append((x, y))

039 return neighbors

040

041 frontier = PriorityQueue()

042 frontier.put((0, start))

043 came\_from = {}

044 cost\_so\_far = {}

045 came\_from[start] = None

046 cost\_so\_far[start] = 0

047

048 while not frontier.empty():

049 current = frontier.get()[1]

050

051 if current == end:

052 break

053

054 for next in get\_neighbors(current):

055 new\_cost = cost\_so\_far[current] + 1

056 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

057 cost\_so\_far[next] = new\_cost

058 priority = new\_cost + heuristic(end, next)

059 frontier.put((priority, next))

060 came\_from[next] = current

061

062 current = end

063 path = []

064 while current != start:

065 if current not in came\_from:

066 return None

067 path.append(current)

068 current = came\_from[current]

069 path.append(start)

070 path.reverse()

071 return path

072

073 def find\_path(self, height, width):

074 start = (0, 0)

075 end = (height - 1, width - 1)

076 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

077 if path is not None:

078 self.show\_path(path)

079 else:

080 print("No path found!")

081

082 def show\_path(self, path):

083 if path:

084 for (x, y) in path:

085 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

086 self.canvas.update()

087 else:

088 print("No path to show.")

089

090 def update\_maze\_type(self, maze\_type):

091 self.maze\_type = maze\_type

092

093 def set\_window\_size(self, width, height):

094 canvas\_width = width \* 10

095 canvas\_height = height \* 10

096 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

097

098 def clear\_root(self):

099 for widget in self.root.winfo\_children():

100 widget.destroy()

101

102

103 def main\_menu(self):

104 self.clear\_root()

105 self.root.geometry("300x200")

106 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

107 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

108 self.add\_quit\_button()

109

110 def randomize\_and\_generate(self):

111 self.height\_slider.set(random.randint(15, 76))

112 self.width\_slider.set(random.randint(15, 76))

113

114 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

115

116 def make\_maze\_menu(self):

117 self.clear\_root()

118 self.root.geometry("300x400")

119

120 height\_label = tk.Label(self.root, text="Maze Height:")

121 height\_label.pack()

122 self.height\_slider = tk.Scale(self.root, from\_=15, to=100, orient="horizontal", label="15 to 100 cells")

123 self.height\_slider.pack()

124

125 width\_label = tk.Label(self.root, text="Maze Width:")

126 width\_label.pack()

127 self.width\_slider = tk.Scale(self.root, from\_=15, to=100, orient="horizontal", label="15 to 100 cells")

128 self.width\_slider.pack()

129

130 maze\_type = tk.StringVar(self.root)

131 maze\_type.set(self.maze\_type)

132 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

133

134 randomize\_button = tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate)

135 randomize\_button.pack()

136

137 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

138

139 self.add\_go\_back\_button()

140 self.add\_quit\_button()

141

142 def update\_size(self, event=None):

143 height = int(self.height\_slider.get())

144 width = int(self.width\_slider.get())

145 self.set\_window\_size(width, height)

146

147 def my\_mazes(self):

148 self.clear\_root()

149 self.add\_go\_back\_button()

150 self.add\_quit\_button()

151

152 def go\_back(self):

153 self.main\_menu()

154

155 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

156 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

157

158 def quit\_app(self):

159 self.root.quit()

160

161 def add\_go\_back\_button(self):

162 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

163

164 def add\_quit\_button(self):

165 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

166

167 def display\_maze(self, height, width):

168 self.clear\_root()

169 canvas\_width = width \* 10

170 canvas\_height = height \* 10

171

172 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

173 self.canvas = canvas

174

175 canvas.pack(side=tk.TOP)

176

177 self.maze.render(canvas)

178

179 button\_frame = tk.Frame(self.root)

180 button\_frame.pack(after=canvas)

181

182 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

183 quit\_button.pack(side=tk.LEFT)

184

185 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

186 regenerate\_button.pack(side=tk.LEFT)

187

188 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

189 find\_path\_button.pack(side=tk.LEFT)

190

191 self.add\_go\_back\_button\_generate\_maze(button\_frame)

192

193 def generate\_maze(self, height, width, maze\_type):

194 self.current\_height = height

195 self.current\_width = width

196 self.current\_maze\_type = maze\_type

197 valid\_maze = False

198 maze\_generation\_attempts = 0

199

200 while not valid\_maze and maze\_generation\_attempts < 250:

201 maze\_generation\_attempts += 1

202 self.maze = Maze(height, width)

203 if maze\_type == "Perfect":

204 self.maze.maze = recursive\_backtracker(height, width)

205 else:

206 self.maze.maze = prims\_algorithm(height, width)

207

208 start = (0, 0)

209 end = (height - 1, width - 1)

210 if self.a\_star\_search(self.maze.maze, start, end):

211 valid\_maze = True

212 else:

213 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

214

215 if valid\_maze:

216 self.display\_maze(height, width)

217 else:

218 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze with the given dimensions. Please try different dimensions or regenerate.")

219 print("Could not generate a valid maze. Please try again.")

220 self.make\_maze\_menu()

221

222

223 def recursive\_backtracker(height, width):

224 def carve\_passage\_from(cx, cy, grid):

225 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

226 random.shuffle(directions)

227

228 for (nx, ny) in directions:

229 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

230 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

231 grid[nx][ny] = 1

232 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

233 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

234

235 maze = [[0] \* width for \_ in range(height)]

236 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

237 maze[start\_x][start\_y] = 1

238 carve\_passage\_from(start\_x, start\_y, maze)

239 return maze

240

241

242 def prims\_algorithm(height, width):

243 local\_maze = [[0] \* width for \_ in range(height)]

244 visited = set()

245 walls = set()

246 start = (0, 0)

247 visited.add(start)

248 local\_maze[start[0]][start[1]] = 1

249 walls.update({(0, 1), (1, 0)})

250

251 while walls:

252 wall = random.choice(list(walls))

253 x, y = wall

254

255 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

256

257 if len(neighbors) == 1:

258 nx, ny = neighbors[0]

259 local\_maze[x][y] = 1

260 visited.add((x, y))

261

262 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

263 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

264 walls.add((dx, dy))

265

266 walls.remove(wall)

267

268 return local\_maze

269

270 if \_\_name\_\_ == "\_\_main\_\_":

271 root = tk.Tk()

272 app = MazeApplication(root)

273 root.mainloop()

Version 8

Changes made:

* Implemented maze saving functionality, allowing the current maze to be saved
* The maze is saved to the database along with its ID, and timestamp, alongside which algorithm was used to create it and how large the maze was (height and width)
* Information dialogue opens up when the maze has been successfully saved
* To access saved mazes, “my mazes” button now works and opens the saved mazes menu
* Saved mazes can be generated again, which will generate the same identical maze that was saved
* Saved mazes can be deleted one by one
* Saved mazes can be deleted all at once, which will open a confirmation dialog asking the user if they are sure of their decision

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004 import tkinter.messagebox as msgbox

005 import sqlite3

006

007 def create\_database():

008 conn = sqlite3.connect('mazes.db')

009 c = conn.cursor()

010 c.execute('''CREATE TABLE IF NOT EXISTS mazes

011 (id INTEGER PRIMARY KEY, height INTEGER, width INTEGER,

012 maze\_type TEXT, maze\_data TEXT, saved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP)''')

013 conn.commit()

014 conn.close()

015

016 create\_database()

017

018 class Maze:

019 def \_\_init\_\_(self, height, width):

020 self.height = height

021 self.width = width

022 self.maze = [[0] \* width for \_ in range(height)]

023

024 def generate(self):

025 self.maze = prims\_algorithm(self.height, self.width)

026

027 def render(self, canvas):

028 for i in range(self.height):

029 for j in range(self.width):

030 color = "black" if self.maze[i][j] == 0 else "white"

031 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

032

033 class MazeApplication:

034 def \_\_init\_\_(self, root):

035 self.root = root

036 self.root.title("Maze Program")

037 self.main\_menu()

038 self.maze\_type = "Perfect"

039

040 def delete\_maze(self, maze\_id):

041 conn = sqlite3.connect('mazes.db')

042 c = conn.cursor()

043 c.execute("DELETE FROM mazes WHERE id = ?", (maze\_id,))

044 conn.commit()

045 conn.close()

046 self.my\_mazes()

047

048 def delete\_all\_mazes(self):

049 conn = sqlite3.connect('mazes.db')

050 c = conn.cursor()

051 c.execute("SELECT COUNT(\*) FROM mazes")

052 count = c.fetchone()[0]

053 conn.close()

054

055 if count == 0:

056 msgbox.showinfo("Delete All", "There are no mazes to delete.")

057 else:

058 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all mazes?")

059 if response:

060 conn = sqlite3.connect('mazes.db')

061 c = conn.cursor()

062 c.execute("DELETE FROM mazes")

063 conn.commit()

064 conn.close()

065 self.my\_mazes()

066

067

068

069 def save\_current\_maze(self):

070 conn = sqlite3.connect('mazes.db')

071 c = conn.cursor()

072 maze\_str = ','.join([' '.join(map(str, row)) for row in self.maze.maze])

073 c.execute("INSERT INTO mazes (height, width, maze\_type, maze\_data) VALUES (?, ?, ?, ?)",

074 (self.current\_height, self.current\_width, self.current\_maze\_type, maze\_str))

075 conn.commit()

076 conn.close()

077 msgbox.showinfo("Success", "Successfully saved the maze")

078

079 @staticmethod

080 def a\_star\_search(maze, start, end):

081 def heuristic(a, b):

082 return abs(a[0] - b[0]) + abs(a[1] - b[1])

083

084 def get\_neighbors(pos):

085 neighbors = []

086 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

087 x, y = pos[0] + dx, pos[1] + dy

088 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

089 neighbors.append((x, y))

090 return neighbors

091

092 frontier = PriorityQueue()

093 frontier.put((0, start))

094 came\_from = {}

095 cost\_so\_far = {}

096 came\_from[start] = None

097 cost\_so\_far[start] = 0

098

099 while not frontier.empty():

100 current = frontier.get()[1]

101

102 if current == end:

103 break

104

105 for next in get\_neighbors(current):

106 new\_cost = cost\_so\_far[current] + 1

107 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

108 cost\_so\_far[next] = new\_cost

109 priority = new\_cost + heuristic(end, next)

110 frontier.put((priority, next))

111 came\_from[next] = current

112

113 current = end

114 path = []

115 while current != start:

116 if current not in came\_from:

117 return None

118 path.append(current)

119 current = came\_from[current]

120 path.append(start)

121 path.reverse()

122 return path

123

124 def find\_path(self, height, width):

125 start = (0, 0)

126 end = (height - 1, width - 1)

127 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

128 if path is not None:

129 self.show\_path(path)

130 else:

131 print("No path found!")

132

133 def show\_path(self, path):

134 if path:

135 for (x, y) in path:

136 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

137 self.canvas.update()

138 else:

139 print("No path to show.")

140

141 def update\_maze\_type(self, maze\_type):

142 self.maze\_type = maze\_type

143

144 def set\_window\_size(self, width, height):

145 canvas\_width = width \* 10

146 canvas\_height = height \* 10

147 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

148

149 def clear\_root(self):

150 for widget in self.root.winfo\_children():

151 widget.destroy()

152

153

154 def main\_menu(self):

155 self.clear\_root()

156 self.root.geometry("300x200")

157 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

158 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

159 self.add\_quit\_button()

160

161 def randomize\_and\_generate(self):

162 self.height\_slider.set(random.randint(15, 76))

163 self.width\_slider.set(random.randint(15, 76))

164

165 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

166

167 def make\_maze\_menu(self):

168 self.clear\_root()

169 self.root.geometry("300x400")

170

171 height\_label = tk.Label(self.root, text="Maze Height:")

172 height\_label.pack()

173 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

174 self.height\_slider.pack()

175

176 width\_label = tk.Label(self.root, text="Maze Width:")

177 width\_label.pack()

178 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

179 self.width\_slider.pack()

180

181 maze\_type = tk.StringVar(self.root)

182 maze\_type.set(self.maze\_type)

183 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

184

185 randomize\_button = tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate)

186 randomize\_button.pack()

187

188 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

189

190 self.add\_go\_back\_button()

191 self.add\_quit\_button()

192

193 def update\_size(self, event=None):

194 height = int(self.height\_slider.get())

195 width = int(self.width\_slider.get())

196 self.set\_window\_size(width, height)

197

198 def regenerate\_saved\_maze(self, maze\_id):

199 conn = sqlite3.connect('mazes.db')

200 c = conn.cursor()

201 c.execute("SELECT height, width, maze\_data FROM mazes WHERE id = ?", (maze\_id,))

202 height, width, maze\_str = c.fetchone()

203 conn.close()

204

205 maze\_array = [list(map(int, row.split())) for row in maze\_str.split(',')]

206 self.maze = Maze(height, width)

207 self.maze.maze = maze\_array

208 self.display\_maze(height, width)

209

210 def my\_mazes(self):

211 self.clear\_root()

212 self.add\_go\_back\_button()

213 self.add\_quit\_button()

214

215 delete\_all\_button = tk.Button(self.root, text="Delete All", command=self.delete\_all\_mazes)

216 delete\_all\_button.pack()

217

218 conn = sqlite3.connect('mazes.db')

219 c = conn.cursor()

220 c.execute("SELECT id, height, width, saved\_at FROM mazes")

221 saved\_mazes = c.fetchall()

222 conn.close()

223

224 scrollable\_frame = tk.Frame(self.root)

225 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

226

227 canvas = tk.Canvas(scrollable\_frame)

228 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

229 canvas.configure(yscrollcommand=scrollbar.set)

230

231 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

232 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

233

234 inner\_frame = tk.Frame(canvas)

235 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

236

237 for maze in saved\_mazes:

238 maze\_id, height, width, saved\_at = maze

239 maze\_frame = tk.Frame(inner\_frame)

240 maze\_label = tk.Label(maze\_frame, text=f"Maze ID: {maze\_id}, Size: {height}x{width}, Saved: {saved\_at}")

241 maze\_label.pack(side=tk.LEFT)

242

243 regenerate\_button = tk.Button(maze\_frame, text="Generate", command=lambda m\_id=maze\_id: self.regenerate\_saved\_maze(m\_id))

244 regenerate\_button.pack(side=tk.LEFT)

245

246 delete\_button = tk.Button(maze\_frame, text="Delete", command=lambda m\_id=maze\_id: self.delete\_maze(m\_id))

247 delete\_button.pack(side=tk.LEFT)

248

249 maze\_frame.pack()

250

251 inner\_frame.update\_idletasks()

252 canvas.config(scrollregion=canvas.bbox("all"))

253

254

255 def go\_back(self):

256 self.main\_menu()

257

258 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

259 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

260

261 def quit\_app(self):

262 self.root.quit()

263

264 def add\_go\_back\_button(self):

265 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

266

267 def add\_quit\_button(self):

268 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

269

270 def display\_maze(self, height, width):

271 self.clear\_root()

272 canvas\_width = width \* 10

273 canvas\_height = height \* 10

274

275 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

276 self.canvas = canvas

277

278 canvas.pack(side=tk.TOP)

279

280 self.maze.render(canvas)

281

282 button\_frame = tk.Frame(self.root)

283 button\_frame.pack(after=canvas)

284

285 save\_button = tk.Button(button\_frame, text="Save", command=self.save\_current\_maze)

286 save\_button.pack(side=tk.LEFT)

287

288 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

289 quit\_button.pack(side=tk.LEFT)

290

291 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

292 regenerate\_button.pack(side=tk.LEFT)

293

294 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

295 find\_path\_button.pack(side=tk.LEFT)

296

297 self.add\_go\_back\_button\_generate\_maze(button\_frame)

298

299 def generate\_maze(self, height, width, maze\_type):

300 self.current\_height = height

301 self.current\_width = width

302 self.current\_maze\_type = maze\_type

303 valid\_maze = False

304 maze\_generation\_attempts = 0

305

306 while not valid\_maze and maze\_generation\_attempts < 250:

307 maze\_generation\_attempts += 1

308 self.maze = Maze(height, width)

309 if maze\_type == "Perfect":

310 self.maze.maze = recursive\_backtracker(height, width)

311 else:

312 self.maze.maze = prims\_algorithm(height, width)

313

314 start = (0, 0)

315 end = (height - 1, width - 1)

316 if self.a\_star\_search(self.maze.maze, start, end):

317 valid\_maze = True

318 else:

319 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

320

321 if valid\_maze:

322 self.display\_maze(height, width)

323 else:

324 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze with the given dimensions. Please try different dimensions or regenerate.")

325 print("Could not generate a valid maze. Please try again.")

326 self.make\_maze\_menu()

327

328

329 def recursive\_backtracker(height, width):

330 def carve\_passage\_from(cx, cy, grid):

331 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

332 random.shuffle(directions)

333

334 for (nx, ny) in directions:

335 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

336 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

337 grid[nx][ny] = 1

338 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

339 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

340

341 maze = [[0] \* width for \_ in range(height)]

342 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

343 maze[start\_x][start\_y] = 1

344 carve\_passage\_from(start\_x, start\_y, maze)

345 return maze

346

347

348 def prims\_algorithm(height, width):

349 local\_maze = [[0] \* width for \_ in range(height)]

350 visited = set()

351 walls = set()

352 start = (0, 0)

353 visited.add(start)

354 local\_maze[start[0]][start[1]] = 1

355 walls.update({(0, 1), (1, 0)})

356

357 while walls:

358 wall = random.choice(list(walls))

359 x, y = wall

360

361 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

362

363 if len(neighbors) == 1:

364 nx, ny = neighbors[0]

365 local\_maze[x][y] = 1

366 visited.add((x, y))

367

368 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

369 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

370 walls.add((dx, dy))

371

372 walls.remove(wall)

373

374 return local\_maze

375

376 if \_\_name\_\_ == "\_\_main\_\_":

377 root = tk.Tk()

378 app = MazeApplication(root)

379 root.mainloop()

Version 9

Changes made:

* Added user authentication system
* The user is now prompted with entering their password and username when the program is first opened
* Added option to sign up
* The password field is obscured with stars (\*\*\*) by default, and has an option to reveal the password
* Added an admin login which opens the admin console, allowing to delete usernames from the database

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004 import tkinter.messagebox as msgbox

005 import sqlite3

006 import bcrypt

007

008 def hash\_password(password):

009 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

010

011 def check\_password(hashed\_password, user\_password):

012 return bcrypt.checkpw(user\_password.encode('utf-8'), hashed\_password)

013

014

015 def create\_user\_database():

016 conn = sqlite3.connect('mazes.db')

017 c = conn.cursor()

018 c.execute('''CREATE TABLE IF NOT EXISTS users

019 (username TEXT PRIMARY KEY, password\_hash TEXT)''')

020 conn.commit()

021 conn.close()

022

023 create\_user\_database()

024

025 def create\_database():

026 conn = sqlite3.connect('mazes.db')

027 c = conn.cursor()

028 c.execute('''CREATE TABLE IF NOT EXISTS mazes

029 (id INTEGER PRIMARY KEY, height INTEGER, width INTEGER,

030 maze\_type TEXT, maze\_data TEXT, saved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP)''')

031 conn.commit()

032 conn.close()

033

034 create\_database()

035

036 class Maze:

037 def \_\_init\_\_(self, height, width):

038 self.height = height

039 self.width = width

040 self.maze = [[0] \* width for \_ in range(height)]

041

042 def generate(self):

043 self.maze = prims\_algorithm(self.height, self.width)

044

045 def render(self, canvas):

046 for i in range(self.height):

047 for j in range(self.width):

048 color = "black" if self.maze[i][j] == 0 else "white"

049 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

050

051 class MazeApplication:

052 def \_\_init\_\_(self, root):

053 self.root = root

054 self.root.title("Maze Program")

055 self.show\_login\_form()

056 self.maze\_type = "Perfect"

057

058 def show\_login\_form(self):

059 self.clear\_root()

060

061 tk.Label(self.root, text="Username").pack()

062 self.username\_entry = tk.Entry(self.root)

063 self.username\_entry.pack()

064

065 tk.Label(self.root, text="Password").pack()

066 self.password\_entry = tk.Entry(self.root, show="\*")

067 self.password\_entry.pack()

068

069 self.show\_password\_button = tk.Button(self.root, text="Show Password", command=self.toggle\_password\_visibility)

070 self.show\_password\_button.pack()

071

072 tk.Button(self.root, text="Log In", command=self.login).pack()

073 tk.Button(self.root, text="Sign Up", command=self.show\_signup\_form).pack()

074

075 def login(self):

076 username = self.username\_entry.get()

077 password = self.password\_entry.get()

078

079 if username == "admin123" and password == "admin123":

080 self.show\_admin\_window()

081 return

082

083 conn = sqlite3.connect('mazes.db')

084 c = conn.cursor()

085 c.execute("SELECT password\_hash FROM users WHERE username = ?", (username,))

086 result = c.fetchone()

087

088 if result and check\_password(result[0], password):

089 self.main\_menu()

090 else:

091 msgbox.showerror("Login Failed", "Invalid username or password")

092 conn.close()

093

094 def toggle\_password\_visibility(self):

095 if self.password\_entry.cget('show') == '\*':

096 self.password\_entry.config(show='')

097 self.show\_password\_button.config(text="Hide Password")

098 else:

099 self.password\_entry.config(show='\*')

100 self.show\_password\_button.config(text="Show Password")

101

102 def toggle\_signup\_password\_visibility(self):

103 if self.new\_password\_entry.cget('show') == '\*':

104 self.new\_password\_entry.config(show='')

105 self.show\_signup\_password\_button.config(text="Hide Password")

106 else:

107 self.new\_password\_entry.config(show='\*')

108 self.show\_signup\_password\_button.config(text="Show Password")

109

110 def show\_signup\_form(self):

111 self.signup\_window = tk.Toplevel(self.root)

112 self.signup\_window.title("Sign Up")

113

114 tk.Label(self.signup\_window, text="Username").pack()

115 self.new\_username\_entry = tk.Entry(self.signup\_window)

116 self.new\_username\_entry.pack()

117

118 tk.Label(self.signup\_window, text="Password").pack()

119 self.new\_password\_entry = tk.Entry(self.signup\_window, show="\*")

120 self.new\_password\_entry.pack()

121

122 self.show\_signup\_password\_button = tk.Button(self.signup\_window, text="Show Password", command=self.toggle\_signup\_password\_visibility)

123 self.show\_signup\_password\_button.pack()

124

125 tk.Button(self.signup\_window, text="Sign Up", command=self.signup).pack()

126

127 def show\_admin\_window(self):

128 if hasattr(self, 'admin\_window') and self.admin\_window.winfo\_exists():

129 for widget in self.admin\_window.winfo\_children():

130 widget.destroy()

131 else:

132 self.admin\_window = tk.Toplevel(self.root)

133 self.admin\_window.title("Admin Panel")

134

135 scrollable\_frame = tk.Frame(self.admin\_window)

136 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

137

138 canvas = tk.Canvas(scrollable\_frame)

139 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

140 canvas.configure(yscrollcommand=scrollbar.set)

141

142 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

143 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

144

145 inner\_frame = tk.Frame(canvas)

146 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

147

148 conn = sqlite3.connect('mazes.db')

149 c = conn.cursor()

150 c.execute("SELECT username FROM users WHERE username != 'admin123'")

151 users = c.fetchall()

152

153 for user in users:

154 username = user[0]

155 user\_frame = tk.Frame(inner\_frame)

156 user\_label = tk.Label(user\_frame, text=username)

157 user\_label.pack(side=tk.LEFT)

158

159 delete\_button = tk.Button(user\_frame, text="Delete", command=lambda u=username: self.delete\_user(u))

160 delete\_button.pack(side=tk.LEFT)

161

162 user\_frame.pack()

163

164 inner\_frame.update\_idletasks()

165 canvas.config(scrollregion=canvas.bbox("all"))

166

167 tk.Button(self.admin\_window, text="Delete All Users", command=self.delete\_all\_users).pack()

168 tk.Button(self.admin\_window, text="Quit", command=self.admin\_window.destroy).pack()

169

170 def delete\_all\_users(self):

171 conn = sqlite3.connect('mazes.db')

172 c = conn.cursor()

173 c.execute("SELECT COUNT(\*) FROM users WHERE username != 'admin123'")

174 count = c.fetchone()[0]

175

176 if count == 0:

177 msgbox.showinfo("Delete All Users", "There are no users to delete.")

178 else:

179 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all users?")

180 if response:

181 c.execute("DELETE FROM users WHERE username != 'admin123'")

182 conn.commit()

183 conn.close()

184 self.show\_admin\_window()

185

186

187 def delete\_user(self, username):

188 conn = sqlite3.connect('mazes.db')

189 c = conn.cursor()

190 c.execute("DELETE FROM users WHERE username = ?", (username,))

191 conn.commit()

192 conn.close()

193 self.show\_admin\_window()

194

195

196 def signup(self):

197 new\_username = self.new\_username\_entry.get()

198 new\_password = self.new\_password\_entry.get()

199 hashed\_password = hash\_password(new\_password)

200

201 if new\_username == "admin123":

202 msgbox.showerror("Signup Failed", "This username is reserved and cannot be used.")

203 return

204

205 try:

206 conn = sqlite3.connect('mazes.db')

207 c = conn.cursor()

208 c.execute("INSERT INTO users (username, password\_hash) VALUES (?, ?)", (new\_username, hashed\_password))

209 conn.commit()

210 msgbox.showinfo("Signup Successful", "Account created successfully")

211 self.signup\_window.destroy()

212 except sqlite3.IntegrityError:

213 msgbox.showerror("Signup Failed", "Username already exists")

214 finally:

215 conn.close()

216

217 def delete\_maze(self, maze\_id):

218 conn = sqlite3.connect('mazes.db')

219 c = conn.cursor()

220 c.execute("DELETE FROM mazes WHERE id = ?", (maze\_id,))

221 conn.commit()

222 conn.close()

223 self.my\_mazes()

224

225 def delete\_all\_mazes(self):

226 conn = sqlite3.connect('mazes.db')

227 c = conn.cursor()

228 c.execute("SELECT COUNT(\*) FROM mazes")

229 count = c.fetchone()[0]

230 conn.close()

231

232 if count == 0:

233 msgbox.showinfo("Delete All", "There are no mazes to delete.")

234 else:

235 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all mazes?")

236 if response:

237 conn = sqlite3.connect('mazes.db')

238 c = conn.cursor()

239 c.execute("DELETE FROM mazes")

240 conn.commit()

241 conn.close()

242 self.my\_mazes()

243

244

245

246 def save\_current\_maze(self):

247 conn = sqlite3.connect('mazes.db')

248 c = conn.cursor()

249 maze\_str = ','.join([' '.join(map(str, row)) for row in self.maze.maze])

250 c.execute("INSERT INTO mazes (height, width, maze\_type, maze\_data) VALUES (?, ?, ?, ?)",

251 (self.current\_height, self.current\_width, self.current\_maze\_type, maze\_str))

252 conn.commit()

253 conn.close()

254 msgbox.showinfo("Success", "Successfully saved the maze")

255

256 @staticmethod

257 def a\_star\_search(maze, start, end):

258 def heuristic(a, b):

259 return abs(a[0] - b[0]) + abs(a[1] - b[1])

260

261 def get\_neighbors(pos):

262 neighbors = []

263 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

264 x, y = pos[0] + dx, pos[1] + dy

265 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

266 neighbors.append((x, y))

267 return neighbors

268

269 frontier = PriorityQueue()

270 frontier.put((0, start))

271 came\_from = {}

272 cost\_so\_far = {}

273 came\_from[start] = None

274 cost\_so\_far[start] = 0

275

276 while not frontier.empty():

277 current = frontier.get()[1]

278

279 if current == end:

280 break

281

282 for next in get\_neighbors(current):

283 new\_cost = cost\_so\_far[current] + 1

284 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

285 cost\_so\_far[next] = new\_cost

286 priority = new\_cost + heuristic(end, next)

287 frontier.put((priority, next))

288 came\_from[next] = current

289

290 current = end

291 path = []

292 while current != start:

293 if current not in came\_from:

294 return None

295 path.append(current)

296 current = came\_from[current]

297 path.append(start)

298 path.reverse()

299 return path

300

301 def find\_path(self, height, width):

302 start = (0, 0)

303 end = (height - 1, width - 1)

304 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

305 if path is not None:

306 self.show\_path(path)

307 else:

308 print("No path found!")

309

310 def show\_path(self, path):

311 if path:

312 for (x, y) in path:

313 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

314 self.canvas.update()

315 else:

316 print("No path to show.")

317

318 def update\_maze\_type(self, maze\_type):

319 self.maze\_type = maze\_type

320

321 def set\_window\_size(self, width, height):

322 canvas\_width = width \* 10

323 canvas\_height = height \* 10

324 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

325

326 def clear\_root(self):

327 for widget in self.root.winfo\_children():

328 widget.destroy()

329

330

331 def main\_menu(self):

332 self.clear\_root()

333 self.root.geometry("300x200")

334 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

335 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

336 self.add\_quit\_button()

337

338 def randomize\_and\_generate(self):

339 self.height\_slider.set(random.randint(15, 76))

340 self.width\_slider.set(random.randint(15, 76))

341

342 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

343

344 def make\_maze\_menu(self):

345 self.clear\_root()

346 self.root.geometry("300x400")

347

348 height\_label = tk.Label(self.root, text="Maze Height:")

349 height\_label.pack()

350 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

351 self.height\_slider.pack()

352

353 width\_label = tk.Label(self.root, text="Maze Width:")

354 width\_label.pack()

355 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

356 self.width\_slider.pack()

357

358 maze\_type = tk.StringVar(self.root)

359 maze\_type.set(self.maze\_type)

360 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

361

362 randomize\_button = tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate)

363 randomize\_button.pack()

364

365 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

366

367 self.add\_go\_back\_button()

368 self.add\_quit\_button()

369

370 def update\_size(self, event=None):

371 height = int(self.height\_slider.get())

372 width = int(self.width\_slider.get())

373 self.set\_window\_size(width, height)

374

375 def regenerate\_saved\_maze(self, maze\_id):

376 conn = sqlite3.connect('mazes.db')

377 c = conn.cursor()

378 c.execute("SELECT height, width, maze\_data FROM mazes WHERE id = ?", (maze\_id,))

379 height, width, maze\_str = c.fetchone()

380 conn.close()

381

382 maze\_array = [list(map(int, row.split())) for row in maze\_str.split(',')]

383 self.maze = Maze(height, width)

384 self.maze.maze = maze\_array

385 self.display\_maze(height, width)

386

387 def my\_mazes(self):

388 self.clear\_root()

389 self.add\_go\_back\_button()

390 self.add\_quit\_button()

391

392 delete\_all\_button = tk.Button(self.root, text="Delete All", command=self.delete\_all\_mazes)

393 delete\_all\_button.pack()

394

395 conn = sqlite3.connect('mazes.db')

396 c = conn.cursor()

397 c.execute("SELECT id, height, width, saved\_at FROM mazes")

398 saved\_mazes = c.fetchall()

399 conn.close()

400

401 scrollable\_frame = tk.Frame(self.root)

402 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

403

404 canvas = tk.Canvas(scrollable\_frame)

405 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

406 canvas.configure(yscrollcommand=scrollbar.set)

407

408 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

409 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

410

411 inner\_frame = tk.Frame(canvas)

412 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

413

414 for maze in saved\_mazes:

415 maze\_id, height, width, saved\_at = maze

416 maze\_frame = tk.Frame(inner\_frame)

417 maze\_label = tk.Label(maze\_frame, text=f"Maze ID: {maze\_id}, Size: {height}x{width}, Saved: {saved\_at}")

418 maze\_label.pack(side=tk.LEFT)

419

420 regenerate\_button = tk.Button(maze\_frame, text="Generate", command=lambda m\_id=maze\_id: self.regenerate\_saved\_maze(m\_id))

421 regenerate\_button.pack(side=tk.LEFT)

422

423 delete\_button = tk.Button(maze\_frame, text="Delete", command=lambda m\_id=maze\_id: self.delete\_maze(m\_id))

424 delete\_button.pack(side=tk.LEFT)

425

426 maze\_frame.pack()

427

428 inner\_frame.update\_idletasks()

429 canvas.config(scrollregion=canvas.bbox("all"))

430

431

432 def go\_back(self):

433 self.main\_menu()

434

435 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

436 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

437

438 def quit\_app(self):

439 self.root.quit()

440

441 def add\_go\_back\_button(self):

442 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

443

444 def add\_quit\_button(self):

445 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

446

447 def display\_maze(self, height, width):

448 self.clear\_root()

449 canvas\_width = width \* 10

450 canvas\_height = height \* 10

451

452 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

453 self.canvas = canvas

454

455 canvas.pack(side=tk.TOP)

456

457 self.maze.render(canvas)

458

459 button\_frame = tk.Frame(self.root)

460 button\_frame.pack(after=canvas)

461

462 save\_button = tk.Button(button\_frame, text="Save", command=self.save\_current\_maze)

463 save\_button.pack(side=tk.LEFT)

464

465 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

466 quit\_button.pack(side=tk.LEFT)

467

468 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

469 regenerate\_button.pack(side=tk.LEFT)

470

471 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

472 find\_path\_button.pack(side=tk.LEFT)

473

474 self.add\_go\_back\_button\_generate\_maze(button\_frame)

475

476 def generate\_maze(self, height, width, maze\_type):

477 self.current\_height = height

478 self.current\_width = width

479 self.current\_maze\_type = maze\_type

480 valid\_maze = False

481 maze\_generation\_attempts = 0

482

483 while not valid\_maze and maze\_generation\_attempts < 250:

484 maze\_generation\_attempts += 1

485 self.maze = Maze(height, width)

486 if maze\_type == "Perfect":

487 self.maze.maze = recursive\_backtracker(height, width)

488 else:

489 self.maze.maze = prims\_algorithm(height, width)

490

491 start = (0, 0)

492 end = (height - 1, width - 1)

493 if self.a\_star\_search(self.maze.maze, start, end):

494 valid\_maze = True

495 else:

496 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

497

498 if valid\_maze:

499 self.display\_maze(height, width)

500 else:

501 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze with the given dimensions. Please try different dimensions or regenerate.")

502 print("Could not generate a valid maze. Please try again.")

503 self.make\_maze\_menu()

504

505

506 def recursive\_backtracker(height, width):

507 def carve\_passage\_from(cx, cy, grid):

508 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

509 random.shuffle(directions)

510

511 for (nx, ny) in directions:

512 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

513 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

514 grid[nx][ny] = 1

515 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

516 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

517

518 maze = [[0] \* width for \_ in range(height)]

519 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

520 maze[start\_x][start\_y] = 1

521 carve\_passage\_from(start\_x, start\_y, maze)

522 return maze

523

524

525 def prims\_algorithm(height, width):

526 local\_maze = [[0] \* width for \_ in range(height)]

527 visited = set()

528 walls = set()

529 start = (0, 0)

530 visited.add(start)

531 local\_maze[start[0]][start[1]] = 1

532 walls.update({(0, 1), (1, 0)})

533

534 while walls:

535 wall = random.choice(list(walls))

536 x, y = wall

537

538 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

539

540 if len(neighbors) == 1:

541 nx, ny = neighbors[0]

542 local\_maze[x][y] = 1

543 visited.add((x, y))

544

545 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

546 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

547 walls.add((dx, dy))

548

549 walls.remove(wall)

550

551 return local\_maze

552

553 if \_\_name\_\_ == "\_\_main\_\_":

554 root = tk.Tk()

555 app = MazeApplication(root)

556 root.mainloop()

Version 10

Changes made:

* There is now an error when I try to sign up, saying that hash\_password is not defined. I will attempt to fix this error in the next version.

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004 import tkinter.messagebox as msgbox

005 import sqlite3

006 import bcrypt

007

008 class Maze:

009 def \_\_init\_\_(self, height, width):

010 self.height = height

011 self.width = width

012 self.maze = [[0] \* width for \_ in range(height)]

013

014 def generate(self):

015 self.maze = prims\_algorithm(self.height, self.width)

016

017 def render(self, canvas):

018 for i in range(self.height):

019 for j in range(self.width):

020 color = "black" if self.maze[i][j] == 0 else "white"

021 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

022

023 class MazeApplication:

024 def \_\_init\_\_(self, root):

025 self.root = root

026 self.root.title("Maze Program")

027 self.show\_login\_form()

028 self.maze\_type = "Perfect"

029

030 def create\_user\_database():

031 conn = sqlite3.connect('mazes.db')

032 c = conn.cursor()

033 c.execute('''CREATE TABLE IF NOT EXISTS users

034 (username TEXT PRIMARY KEY, password\_hash TEXT)''')

035 conn.commit()

036 conn.close()

037

038 def create\_database():

039 conn = sqlite3.connect('mazes.db')

040 c = conn.cursor()

041 c.execute('''CREATE TABLE IF NOT EXISTS mazes

042 (id INTEGER PRIMARY KEY, height INTEGER, width INTEGER,

043 maze\_type TEXT, maze\_data TEXT, saved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP)''')

044 conn.commit()

045 conn.close()

046

047 create\_user\_database()

048 create\_database()

049

050 def hash\_password(password):

051 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

052

053 def check\_password(self, hashed\_password, user\_password):

054 return bcrypt.checkpw(user\_password.encode('utf-8'), hashed\_password)

055

056 def show\_login\_form(self):

057 self.clear\_root()

058

059 tk.Label(self.root, text="Username").pack()

060 self.username\_entry = tk.Entry(self.root)

061 self.username\_entry.pack()

062

063 tk.Label(self.root, text="Password").pack()

064 self.password\_entry = tk.Entry(self.root, show="\*")

065 self.password\_entry.pack()

066

067 self.show\_password\_button = tk.Button(self.root, text="Show Password", command=self.toggle\_password\_visibility)

068 self.show\_password\_button.pack()

069

070 tk.Button(self.root, text="Log In", command=self.login).pack()

071 tk.Button(self.root, text="Sign Up", command=self.show\_signup\_form).pack()

072

073 def login(self):

074 username = self.username\_entry.get()

075 password = self.password\_entry.get()

076

077 if username == "admin123" and password == "admin123":

078 self.show\_admin\_window()

079 return

080

081 conn = sqlite3.connect('mazes.db')

082 c = conn.cursor()

083 c.execute("SELECT password\_hash FROM users WHERE username = ?", (username,))

084 result = c.fetchone()

085

086 if result and self.check\_password(result[0], password):

087 self.main\_menu()

088 else:

089 msgbox.showerror("Login Failed", "Invalid username or password")

090 conn.close()

091

092 def toggle\_password\_visibility(self):

093 if self.password\_entry.cget('show') == '\*':

094 self.password\_entry.config(show='')

095 self.show\_password\_button.config(text="Hide Password")

096 else:

097 self.password\_entry.config(show='\*')

098 self.show\_password\_button.config(text="Show Password")

099

100 def toggle\_signup\_password\_visibility(self):

101 if self.new\_password\_entry.cget('show') == '\*':

102 self.new\_password\_entry.config(show='')

103 self.show\_signup\_password\_button.config(text="Hide Password")

104 else:

105 self.new\_password\_entry.config(show='\*')

106 self.show\_signup\_password\_button.config(text="Show Password")

107

108 def show\_signup\_form(self):

109 self.signup\_window = tk.Toplevel(self.root)

110 self.signup\_window.title("Sign Up")

111

112 tk.Label(self.signup\_window, text="Username").pack()

113 self.new\_username\_entry = tk.Entry(self.signup\_window)

114 self.new\_username\_entry.pack()

115

116 tk.Label(self.signup\_window, text="Password").pack()

117 self.new\_password\_entry = tk.Entry(self.signup\_window, show="\*")

118 self.new\_password\_entry.pack()

119

120 self.show\_signup\_password\_button = tk.Button(self.signup\_window, text="Show Password", command=self.toggle\_signup\_password\_visibility)

121 self.show\_signup\_password\_button.pack()

122

123 tk.Button(self.signup\_window, text="Sign Up", command=self.signup).pack()

124

125 def show\_admin\_window(self):

126 if hasattr(self, 'admin\_window') and self.admin\_window.winfo\_exists():

127 for widget in self.admin\_window.winfo\_children():

128 widget.destroy()

129 else:

130 self.admin\_window = tk.Toplevel(self.root)

131 self.admin\_window.title("Admin Panel")

132

133 scrollable\_frame = tk.Frame(self.admin\_window)

134 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

135

136 canvas = tk.Canvas(scrollable\_frame)

137 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

138 canvas.configure(yscrollcommand=scrollbar.set)

139

140 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

141 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

142

143 inner\_frame = tk.Frame(canvas)

144 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

145

146 conn = sqlite3.connect('mazes.db')

147 c = conn.cursor()

148 c.execute("SELECT username FROM users WHERE username != 'admin123'")

149 users = c.fetchall()

150

151 for user in users:

152 username = user[0]

153 user\_frame = tk.Frame(inner\_frame)

154 user\_label = tk.Label(user\_frame, text=username)

155 user\_label.pack(side=tk.LEFT)

156

157 delete\_button = tk.Button(user\_frame, text="Delete", command=lambda u=username: self.delete\_user(u))

158 delete\_button.pack(side=tk.LEFT)

159

160 user\_frame.pack()

161

162 inner\_frame.update\_idletasks()

163 canvas.config(scrollregion=canvas.bbox("all"))

164

165 tk.Button(self.admin\_window, text="Delete All Users", command=self.delete\_all\_users).pack()

166 tk.Button(self.admin\_window, text="Quit", command=self.admin\_window.destroy).pack()

167

168 def delete\_all\_users(self):

169 conn = sqlite3.connect('mazes.db')

170 c = conn.cursor()

171 c.execute("SELECT COUNT(\*) FROM users WHERE username != 'admin123'")

172 count = c.fetchone()[0]

173

174 if count == 0:

175 msgbox.showinfo("Delete All Users", "There are no users to delete.")

176 else:

177 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all users?")

178 if response:

179 c.execute("DELETE FROM users WHERE username != 'admin123'")

180 conn.commit()

181 conn.close()

182 self.show\_admin\_window()

183

184 def delete\_user(self, username):

185 conn = sqlite3.connect('mazes.db')

186 c = conn.cursor()

187 c.execute("DELETE FROM users WHERE username = ?", (username,))

188 conn.commit()

189 conn.close()

190 self.show\_admin\_window()

191

192 def signup(self):

193 new\_username = self.new\_username\_entry.get()

194 new\_password = self.new\_password\_entry.get()

195 hashed\_password = hash\_password(new\_password)

196

197 if new\_username == "admin123":

198 msgbox.showerror("Signup Failed", "This username is reserved and cannot be used.")

199 return

200

201 try:

202 conn = sqlite3.connect('mazes.db')

203 c = conn.cursor()

204 c.execute("INSERT INTO users (username, password\_hash) VALUES (?, ?)", (new\_username, hashed\_password))

205 conn.commit()

206 msgbox.showinfo("Signup Successful", "Account created successfully")

207 self.signup\_window.destroy()

208 except sqlite3.IntegrityError:

209 msgbox.showerror("Signup Failed", "Username already exists")

210 finally:

211 conn.close()

212

213 def delete\_maze(self, maze\_id):

214 conn = sqlite3.connect('mazes.db')

215 c = conn.cursor()

216 c.execute("DELETE FROM mazes WHERE id = ?", (maze\_id,))

217 conn.commit()

218 conn.close()

219 self.my\_mazes()

220

221 def delete\_all\_mazes(self):

222 conn = sqlite3.connect('mazes.db')

223 c = conn.cursor()

224 c.execute("SELECT COUNT(\*) FROM mazes")

225 count = c.fetchone()[0]

226 conn.close()

227

228 if count == 0:

229 msgbox.showinfo("Delete All", "There are no mazes to delete.")

230 else:

231 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all mazes?")

232 if response:

233 conn = sqlite3.connect('mazes.db')

234 c = conn.cursor()

235 c.execute("DELETE FROM mazes")

236 conn.commit()

237 conn.close()

238 self.my\_mazes()

239

240 def save\_current\_maze(self):

241 conn = sqlite3.connect('mazes.db')

242 c = conn.cursor()

243 maze\_str = ','.join([' '.join(map(str, row)) for row in self.maze.maze])

244 c.execute("INSERT INTO mazes (height, width, maze\_type, maze\_data) VALUES (?, ?, ?, ?)",

245 (self.current\_height, self.current\_width, self.current\_maze\_type, maze\_str))

246 conn.commit()

247 conn.close()

248 msgbox.showinfo("Success", "Successfully saved the maze")

249

250 @staticmethod

251 def a\_star\_search(maze, start, end):

252 def heuristic(a, b):

253 return abs(a[0] - b[0]) + abs(a[1] - b[1])

254

255 def get\_neighbors(pos):

256 neighbors = []

257 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

258 x, y = pos[0] + dx, pos[1] + dy

259 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

260 neighbors.append((x, y))

261 return neighbors

262

263 frontier = PriorityQueue()

264 frontier.put((0, start))

265 came\_from = {}

266 cost\_so\_far = {}

267 came\_from[start] = None

268 cost\_so\_far[start] = 0

269

270 while not frontier.empty():

271 current = frontier.get()[1]

272

273 if current == end:

274 break

275

276 for next in get\_neighbors(current):

277 new\_cost = cost\_so\_far[current] + 1

278 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

279 cost\_so\_far[next] = new\_cost

280 priority = new\_cost + heuristic(end, next)

281 frontier.put((priority, next))

282 came\_from[next] = current

283

284 current = end

285 path = []

286 while current != start:

287 if current not in came\_from:

288 return None

289 path.append(current)

290 current = came\_from[current]

291 path.append(start)

292 path.reverse()

293 return path

294

295 def find\_path(self, height, width):

296 start = (0, 0)

297 end = (height - 1, width - 1)

298 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

299 if path is not None:

300 self.show\_path(path)

301 else:

302 print("No path found!")

303

304 def show\_path(self, path):

305 if path:

306 for (x, y) in path:

307 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

308 self.canvas.update()

309 else:

310 print("No path to show.")

311

312 def update\_maze\_type(self, maze\_type):

313 self.maze\_type = maze\_type

314

315 def set\_window\_size(self, width, height):

316 canvas\_width = width \* 10

317 canvas\_height = height \* 10

318 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

319

320 def clear\_root(self):

321 for widget in self.root.winfo\_children():

322 widget.destroy()

323

324 def main\_menu(self):

325 self.clear\_root()

326 self.root.geometry("300x200")

327 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

328 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

329 self.add\_quit\_button()

330

331 def randomize\_and\_generate(self):

332 self.height\_slider.set(random.randint(15, 76))

333 self.width\_slider.set(random.randint(15, 76))

334

335 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

336

337 def make\_maze\_menu(self):

338 self.clear\_root()

339 self.root.geometry("300x400")

340

341 height\_label = tk.Label(self.root, text="Maze Height:")

342 height\_label.pack()

343 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

344 self.height\_slider.pack()

345

346 width\_label = tk.Label(self.root, text="Maze Width:")

347 width\_label.pack()

348 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

349 self.width\_slider.pack()

350

351 maze\_type = tk.StringVar(self.root)

352 maze\_type.set(self.maze\_type)

353 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

354

355 randomize\_button = tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate)

356 randomize\_button.pack()

357

358 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

359

360 self.add\_go\_back\_button()

361 self.add\_quit\_button()

362

363 def update\_size(self, event=None):

364 height = int(self.height\_slider.get())

365 width = int(self.width\_slider.get())

366 self.set\_window\_size(width, height)

367

368 def regenerate\_saved\_maze(self, maze\_id):

369 conn = sqlite3.connect('mazes.db')

370 c = conn.cursor()

371 c.execute("SELECT height, width, maze\_data FROM mazes WHERE id = ?", (maze\_id,))

372 height, width, maze\_str = c.fetchone()

373 conn.close()

374

375 maze\_array = [list(map(int, row.split())) for row in maze\_str.split(',')]

376 self.maze = Maze(height, width)

377 self.maze.maze = maze\_array

378 self.display\_maze(height, width)

379

380 def my\_mazes(self):

381 self.clear\_root()

382 self.add\_go\_back\_button()

383 self.add\_quit\_button()

384

385 delete\_all\_button = tk.Button(self.root, text="Delete All", command=self.delete\_all\_mazes)

386 delete\_all\_button.pack()

387

388 conn = sqlite3.connect('mazes.db')

389 c = conn.cursor()

390 c.execute("SELECT id, height, width, saved\_at FROM mazes")

391 saved\_mazes = c.fetchall()

392 conn.close()

393

394 scrollable\_frame = tk.Frame(self.root)

395 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

396

397 canvas = tk.Canvas(scrollable\_frame)

398 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

399 canvas.configure(yscrollcommand=scrollbar.set)

400

401 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

402 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

403

404 inner\_frame = tk.Frame(canvas)

405 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

406

407 for maze in saved\_mazes:

408 maze\_id, height, width, saved\_at = maze

409 maze\_frame = tk.Frame(inner\_frame)

410 maze\_label = tk.Label(maze\_frame, text=f"Maze ID: {maze\_id}, Size: {height}x{width}, Saved: {saved\_at}")

411 maze\_label.pack(side=tk.LEFT)

412

413 regenerate\_button = tk.Button(maze\_frame, text="Generate", command=lambda m\_id=maze\_id: self.regenerate\_saved\_maze(m\_id))

414 regenerate\_button.pack(side=tk.LEFT)

415

416 delete\_button = tk.Button(maze\_frame, text="Delete", command=lambda m\_id=maze\_id: self.delete\_maze(m\_id))

417 delete\_button.pack(side=tk.LEFT)

418

419 maze\_frame.pack()

420

421 inner\_frame.update\_idletasks()

422 canvas.config(scrollregion=canvas.bbox("all"))

423

424 def go\_back(self):

425 self.main\_menu()

426

427 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

428 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

429

430 def quit\_app(self):

431 self.root.quit()

432

433 def add\_go\_back\_button(self):

434 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

435

436 def add\_quit\_button(self):

437 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

438

439 def display\_maze(self, height, width):

440 self.clear\_root()

441 canvas\_width = width \* 10

442 canvas\_height = height \* 10

443

444 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

445 self.canvas = canvas

446

447 canvas.pack(side=tk.TOP)

448

449 self.maze.render(canvas)

450

451 button\_frame = tk.Frame(self.root)

452 button\_frame.pack(after=canvas)

453

454 save\_button = tk.Button(button\_frame, text="Save", command=self.save\_current\_maze)

455 save\_button.pack(side=tk.LEFT)

456

457 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

458 quit\_button.pack(side=tk.LEFT)

459

460 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

461 regenerate\_button.pack(side=tk.LEFT)

462

463 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

464 find\_path\_button.pack(side=tk.LEFT)

465

466 self.add\_go\_back\_button\_generate\_maze(button\_frame)

467

468 def generate\_maze(self, height, width, maze\_type):

469 self.current\_height = height

470 self.current\_width = width

471 self.current\_maze\_type = maze\_type

472 valid\_maze = False

473 maze\_generation\_attempts = 0

474

475 while not valid\_maze and maze\_generation\_attempts < 250:

476 maze\_generation\_attempts += 1

477 self.maze = Maze(height, width)

478 if maze\_type == "Perfect":

479 self.maze.maze = recursive\_backtracker(height, width)

480 else:

481 self.maze.maze = prims\_algorithm(height, width)

482

483 start = (0, 0)

484 end = (height - 1, width - 1)

485 if self.a\_star\_search(self.maze.maze, start, end):

486 valid\_maze = True

487 else:

488 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

489

490 if valid\_maze:

491 self.display\_maze(height, width)

492 else:

493 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze with the given dimensions. Please try different dimensions or regenerate.")

494 print("Could not generate a valid maze. Please try again.")

495 self.make\_maze\_menu()

496

497 def recursive\_backtracker(height, width):

498 def carve\_passage\_from(cx, cy, grid):

499 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

500 random.shuffle(directions)

501

502 for (nx, ny) in directions:

503 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

504 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

505 grid[nx][ny] = 1

506 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

507 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

508

509 maze = [[0] \* width for \_ in range(height)]

510 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

511 maze[start\_x][start\_y] = 1

512 carve\_passage\_from(start\_x, start\_y, maze)

513 return maze

514

515 def prims\_algorithm(height, width):

516 local\_maze = [[0] \* width for \_ in range(height)]

517 visited = set()

518 walls = set()

519 start = (0, 0)

520 visited.add(start)

521 local\_maze[start[0]][start[1]] = 1

522 walls.update({(0, 1), (1, 0)})

523

524 while walls:

525 wall = random.choice(list(walls))

526 x, y = wall

527

528 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

529

530 if len(neighbors) == 1:

531 nx, ny = neighbors[0]

532 local\_maze[x][y] = 1

533 visited.add((x, y))

534

535 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

536 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

537 walls.add((dx, dy))

538

539 walls.remove(wall)

540

541 return local\_maze

542

543 if \_\_name\_\_ == "\_\_main\_\_":

544 root = tk.Tk()

545 app = MazeApplication(root)

546 root.mainloop()

Version 11

Changes made:

* Restructured the program completely, now the elements related to user authentication (such as signup, login) are now part of the UserAuthentication class
* UserAuthentication is instantiated in MazeApplication and a callback method on\_login\_success is passed to handle successful login actions

001 import tkinter as tk

002 import random

003 from queue import PriorityQueue

004 import tkinter.messagebox as msgbox

005 import sqlite3

006 import bcrypt

007

008 class Maze:

009 def \_\_init\_\_(self, height, width):

010 self.height = height

011 self.width = width

012 self.maze = [[0] \* width for \_ in range(height)]

013

014 def generate(self):

015 self.maze = prims\_algorithm(self.height, self.width)

016

017 def render(self, canvas):

018 for i in range(self.height):

019 for j in range(self.width):

020 color = "black" if self.maze[i][j] == 0 else "white"

021 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

022

023 class UserAuthentication:

024 def \_\_init\_\_(self, root, on\_login\_success):

025 self.root = root

026 self.on\_login\_success = on\_login\_success

027 self.create\_user\_database()

028 self.create\_database()

029 self.show\_login\_form()

030

031 @staticmethod

032 def create\_user\_database():

033 conn = sqlite3.connect('mazes.db')

034 c = conn.cursor()

035 c.execute('''CREATE TABLE IF NOT EXISTS users

036 (username TEXT PRIMARY KEY, password\_hash TEXT)''')

037 conn.commit()

038 conn.close()

039

040 @staticmethod

041 def create\_database():

042 conn = sqlite3.connect('mazes.db')

043 c = conn.cursor()

044 c.execute('''CREATE TABLE IF NOT EXISTS mazes

045 (id INTEGER PRIMARY KEY, height INTEGER, width INTEGER,

046 maze\_type TEXT, maze\_data TEXT, saved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP)''')

047 conn.commit()

048 conn.close()

049

050 @staticmethod

051 def hash\_password(password):

052 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

053

054 def clear\_root(self):

055 for widget in self.root.winfo\_children():

056 widget.destroy()

057

058 def check\_password(self, hashed\_password, user\_password):

059 return bcrypt.checkpw(user\_password.encode('utf-8'), hashed\_password)

060

061 def show\_login\_form(self):

062 self.clear\_root()

063

064 tk.Label(self.root, text="Username").pack()

065 self.username\_entry = tk.Entry(self.root)

066 self.username\_entry.pack()

067

068 tk.Label(self.root, text="Password").pack()

069 self.password\_entry = tk.Entry(self.root, show="\*")

070 self.password\_entry.pack()

071

072 self.show\_password\_button = tk.Button(self.root, text="Show Password", command=self.toggle\_password\_visibility)

073 self.show\_password\_button.pack()

074

075 tk.Button(self.root, text="Log In", command=self.login).pack()

076 tk.Button(self.root, text="Sign Up", command=self.show\_signup\_form).pack()

077

078 def login(self):

079 username = self.username\_entry.get()

080 password = self.password\_entry.get()

081

082 if username == "admin123" and password == "admin123":

083 self.show\_admin\_window()

084 return

085

086 conn = sqlite3.connect('mazes.db')

087 c = conn.cursor()

088 c.execute("SELECT password\_hash FROM users WHERE username = ?", (username,))

089 result = c.fetchone()

090

091 if result and self.check\_password(result[0], password):

092 self.on\_login\_success()

093 else:

094 msgbox.showerror("Login Failed", "Invalid username or password")

095 conn.close()

096

097 def toggle\_password\_visibility(self):

098 if self.password\_entry.cget('show') == '\*':

099 self.password\_entry.config(show='')

100 self.show\_password\_button.config(text="Hide Password")

101 else:

102 self.password\_entry.config(show='\*')

103 self.show\_password\_button.config(text="Show Password")

104

105 def toggle\_signup\_password\_visibility(self):

106 if self.new\_password\_entry.cget('show') == '\*':

107 self.new\_password\_entry.config(show='')

108 self.show\_signup\_password\_button.config(text="Hide Password")

109 else:

110 self.new\_password\_entry.config(show='\*')

111 self.show\_signup\_password\_button.config(text="Show Password")

112

113 def show\_signup\_form(self):

114 self.signup\_window = tk.Toplevel(self.root)

115 self.signup\_window.title("Sign Up")

116

117 tk.Label(self.signup\_window, text="Username").pack()

118 self.new\_username\_entry = tk.Entry(self.signup\_window)

119 self.new\_username\_entry.pack()

120

121 tk.Label(self.signup\_window, text="Password").pack()

122 self.new\_password\_entry = tk.Entry(self.signup\_window, show="\*")

123 self.new\_password\_entry.pack()

124

125 self.show\_signup\_password\_button = tk.Button(self.signup\_window, text="Show Password", command=self.toggle\_signup\_password\_visibility)

126 self.show\_signup\_password\_button.pack()

127

128 tk.Button(self.signup\_window, text="Sign Up", command=self.signup).pack()

129

130 def show\_admin\_window(self):

131 if hasattr(self, 'admin\_window') and self.admin\_window.winfo\_exists():

132 for widget in self.admin\_window.winfo\_children():

133 widget.destroy()

134 else:

135 self.admin\_window = tk.Toplevel(self.root)

136 self.admin\_window.title("Admin Panel")

137

138 scrollable\_frame = tk.Frame(self.admin\_window)

139 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

140

141 canvas = tk.Canvas(scrollable\_frame)

142 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

143 canvas.configure(yscrollcommand=scrollbar.set)

144

145 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

146 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

147

148 inner\_frame = tk.Frame(canvas)

149 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

150

151 conn = sqlite3.connect('mazes.db')

152 c = conn.cursor()

153 c.execute("SELECT username FROM users WHERE username != 'admin123'")

154 users = c.fetchall()

155

156 for user in users:

157 username = user[0]

158 user\_frame = tk.Frame(inner\_frame)

159 user\_label = tk.Label(user\_frame, text=username)

160 user\_label.pack(side=tk.LEFT)

161

162 delete\_button = tk.Button(user\_frame, text="Delete", command=lambda u=username: self.delete\_user(u))

163 delete\_button.pack(side=tk.LEFT)

164

165 user\_frame.pack()

166

167 inner\_frame.update\_idletasks()

168 canvas.config(scrollregion=canvas.bbox("all"))

169

170 tk.Button(self.admin\_window, text="Delete All Users", command=self.delete\_all\_users).pack()

171 tk.Button(self.admin\_window, text="Quit", command=self.admin\_window.destroy).pack()

172

173 def delete\_all\_users(self):

174 conn = sqlite3.connect('mazes.db')

175 c = conn.cursor()

176 c.execute("SELECT COUNT(\*) FROM users WHERE username != 'admin123'")

177 count = c.fetchone()[0]

178

179 if count == 0:

180 msgbox.showinfo("Delete All Users", "There are no users to delete.")

181 else:

182 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all users?")

183 if response:

184 c.execute("DELETE FROM users WHERE username != 'admin123'")

185 conn.commit()

186 conn.close()

187 self.show\_admin\_window()

188

189 def delete\_user(self, username):

190 conn = sqlite3.connect('mazes.db')

191 c = conn.cursor()

192 c.execute("DELETE FROM users WHERE username = ?", (username,))

193 conn.commit()

194 conn.close()

195 self.show\_admin\_window()

196

197 def signup(self):

198 new\_username = self.new\_username\_entry.get()

199 new\_password = self.new\_password\_entry.get()

200 hashed\_password = UserAuthentication.hash\_password(new\_password)

201

202 if new\_username == "admin123":

203 msgbox.showerror("Signup Failed", "This username is reserved and cannot be used.")

204 return

205

206 try:

207 conn = sqlite3.connect('mazes.db')

208 c = conn.cursor()

209 c.execute("INSERT INTO users (username, password\_hash) VALUES (?, ?)", (new\_username, hashed\_password))

210 conn.commit()

211 msgbox.showinfo("Signup Successful", "Account created successfully")

212 self.signup\_window.destroy()

213 except sqlite3.IntegrityError:

214 msgbox.showerror("Signup Failed", "Username already exists")

215 finally:

216 conn.close()

217

218 class MazeApplication:

219 def \_\_init\_\_(self, root):

220 self.root = root

221 self.root.title("Maze Program")

222 self.user\_auth = UserAuthentication(root, self.on\_login\_success)

223 self.maze\_type = "Perfect"

224

225 def on\_login\_success(self):

226 self.main\_menu()

227

228 def delete\_maze(self, maze\_id):

229 conn = sqlite3.connect('mazes.db')

230 c = conn.cursor()

231 c.execute("DELETE FROM mazes WHERE id = ?", (maze\_id,))

232 conn.commit()

233 conn.close()

234 self.my\_mazes()

235

236 def delete\_all\_mazes(self):

237 conn = sqlite3.connect('mazes.db')

238 c = conn.cursor()

239 c.execute("SELECT COUNT(\*) FROM mazes")

240 count = c.fetchone()[0]

241 conn.close()

242

243 if count == 0:

244 msgbox.showinfo("Delete All", "There are no mazes to delete.")

245 else:

246 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all mazes?")

247 if response:

248 conn = sqlite3.connect('mazes.db')

249 c = conn.cursor()

250 c.execute("DELETE FROM mazes")

251 conn.commit()

252 conn.close()

253 self.my\_mazes()

254

255 def save\_current\_maze(self):

256 conn = sqlite3.connect('mazes.db')

257 c = conn.cursor()

258 maze\_str = ','.join([' '.join(map(str, row)) for row in self.maze.maze])

259 c.execute("INSERT INTO mazes (height, width, maze\_type, maze\_data) VALUES (?, ?, ?, ?)",

260 (self.current\_height, self.current\_width, self.current\_maze\_type, maze\_str))

261 conn.commit()

262 conn.close()

263 msgbox.showinfo("Success", "Successfully saved the maze")

264

265 @staticmethod

266 def a\_star\_search(maze, start, end):

267 def heuristic(a, b):

268 return abs(a[0] - b[0]) + abs(a[1] - b[1])

269

270 def get\_neighbors(pos):

271 neighbors = []

272 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

273 x, y = pos[0] + dx, pos[1] + dy

274 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

275 neighbors.append((x, y))

276 return neighbors

277

278 frontier = PriorityQueue()

279 frontier.put((0, start))

280 came\_from = {}

281 cost\_so\_far = {}

282 came\_from[start] = None

283 cost\_so\_far[start] = 0

284

285 while not frontier.empty():

286 current = frontier.get()[1]

287

288 if current == end:

289 break

290

291 for next in get\_neighbors(current):

292 new\_cost = cost\_so\_far[current] + 1

293 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

294 cost\_so\_far[next] = new\_cost

295 priority = new\_cost + heuristic(end, next)

296 frontier.put((priority, next))

297 came\_from[next] = current

298

299 current = end

300 path = []

301 while current != start:

302 if current not in came\_from:

303 return None

304 path.append(current)

305 current = came\_from[current]

306 path.append(start)

307 path.reverse()

308 return path

309

310 def find\_path(self, height, width):

311 start = (0, 0)

312 end = (height - 1, width - 1)

313 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

314 if path is not None:

315 self.show\_path(path)

316 else:

317 print("No path found!")

318

319 def show\_path(self, path):

320 if path:

321 for (x, y) in path:

322 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

323 self.canvas.update()

324 else:

325 print("No path to show.")

326

327 def update\_maze\_type(self, maze\_type):

328 self.maze\_type = maze\_type

329

330 def set\_window\_size(self, width, height):

331 canvas\_width = width \* 10

332 canvas\_height = height \* 10

333 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

334

335 def clear\_root(self):

336 for widget in self.root.winfo\_children():

337 widget.destroy()

338

339 def main\_menu(self):

340 self.clear\_root()

341 self.root.geometry("300x200")

342 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

343 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

344 self.add\_quit\_button()

345

346 def randomize\_and\_generate(self):

347 self.height\_slider.set(random.randint(15, 76))

348 self.width\_slider.set(random.randint(15, 76))

349

350 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

351

352 def make\_maze\_menu(self):

353 self.clear\_root()

354 self.root.geometry("300x400")

355

356 height\_label = tk.Label(self.root, text="Maze Height:")

357 height\_label.pack()

358 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

359 self.height\_slider.pack()

360

361 width\_label = tk.Label(self.root, text="Maze Width:")

362 width\_label.pack()

363 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

364 self.width\_slider.pack()

365

366 maze\_type = tk.StringVar(self.root)

367 maze\_type.set(self.maze\_type)

368 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

369

370 randomize\_button = tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate)

371 randomize\_button.pack()

372

373 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

374

375 self.add\_go\_back\_button()

376 self.add\_quit\_button()

377

378 def update\_size(self, event=None):

379 height = int(self.height\_slider.get())

380 width = int(self.width\_slider.get())

381 self.set\_window\_size(width, height)

382

383 def regenerate\_saved\_maze(self, maze\_id):

384 conn = sqlite3.connect('mazes.db')

385 c = conn.cursor()

386 c.execute("SELECT height, width, maze\_data FROM mazes WHERE id = ?", (maze\_id,))

387 height, width, maze\_str = c.fetchone()

388 conn.close()

389

390 maze\_array = [list(map(int, row.split())) for row in maze\_str.split(',')]

391 self.maze = Maze(height, width)

392 self.maze.maze = maze\_array

393 self.display\_maze(height, width)

394

395 def my\_mazes(self):

396 self.clear\_root()

397 self.add\_go\_back\_button()

398 self.add\_quit\_button()

399

400 delete\_all\_button = tk.Button(self.root, text="Delete All", command=self.delete\_all\_mazes)

401 delete\_all\_button.pack()

402

403 conn = sqlite3.connect('mazes.db')

404 c = conn.cursor()

405 c.execute("SELECT id, height, width, saved\_at FROM mazes")

406 saved\_mazes = c.fetchall()

407 conn.close()

408

409 scrollable\_frame = tk.Frame(self.root)

410 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

411

412 canvas = tk.Canvas(scrollable\_frame)

413 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

414 canvas.configure(yscrollcommand=scrollbar.set)

415

416 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

417 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

418

419 inner\_frame = tk.Frame(canvas)

420 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

421

422 for maze in saved\_mazes:

423 maze\_id, height, width, saved\_at = maze

424 maze\_frame = tk.Frame(inner\_frame)

425 maze\_label = tk.Label(maze\_frame, text=f"Maze ID: {maze\_id}, Size: {height}x{width}, Saved: {saved\_at}")

426 maze\_label.pack(side=tk.LEFT)

427

428 regenerate\_button = tk.Button(maze\_frame, text="Generate", command=lambda m\_id=maze\_id: self.regenerate\_saved\_maze(m\_id))

429 regenerate\_button.pack(side=tk.LEFT)

430

431 delete\_button = tk.Button(maze\_frame, text="Delete", command=lambda m\_id=maze\_id: self.delete\_maze(m\_id))

432 delete\_button.pack(side=tk.LEFT)

433

434 maze\_frame.pack()

435

436 inner\_frame.update\_idletasks()

437 canvas.config(scrollregion=canvas.bbox("all"))

438

439 def go\_back(self):

440 self.main\_menu()

441

442 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

443 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

444

445 def quit\_app(self):

446 self.root.quit()

447

448 def add\_go\_back\_button(self):

449 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

450

451 def add\_quit\_button(self):

452 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

453

454 def display\_maze(self, height, width):

455 self.clear\_root()

456 canvas\_width = width \* 10

457 canvas\_height = height \* 10

458

459 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

460 self.canvas = canvas

461

462 canvas.pack(side=tk.TOP)

463

464 self.maze.render(canvas)

465

466 button\_frame = tk.Frame(self.root)

467 button\_frame.pack(after=canvas)

468

469 save\_button = tk.Button(button\_frame, text="Save", command=self.save\_current\_maze)

470 save\_button.pack(side=tk.LEFT)

471

472 quit\_button = tk.Button(button\_frame, text="Quit", command=self.quit\_app)

473 quit\_button.pack(side=tk.LEFT)

474

475 regenerate\_button = tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type))

476 regenerate\_button.pack(side=tk.LEFT)

477

478 find\_path\_button = tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width))

479 find\_path\_button.pack(side=tk.LEFT)

480

481 self.add\_go\_back\_button\_generate\_maze(button\_frame)

482

483 def generate\_maze(self, height, width, maze\_type):

484 self.current\_height = height

485 self.current\_width = width

486 self.current\_maze\_type = maze\_type

487 valid\_maze = False

488 maze\_generation\_attempts = 0

489

490 while not valid\_maze and maze\_generation\_attempts < 250:

491 maze\_generation\_attempts += 1

492 self.maze = Maze(height, width)

493 if maze\_type == "Perfect":

494 self.maze.maze = recursive\_backtracker(height, width)

495 else:

496 self.maze.maze = prims\_algorithm(height, width)

497

498 start = (0, 0)

499 end = (height - 1, width - 1)

500 if self.a\_star\_search(self.maze.maze, start, end):

501 valid\_maze = True

502 else:

503 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

504

505 if valid\_maze:

506 self.display\_maze(height, width)

507 else:

508 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze with the given dimensions. Please try different dimensions or regenerate.")

509 print("Could not generate a valid maze. Please try again.")

510 self.make\_maze\_menu()

511

512 def recursive\_backtracker(height, width):

513 def carve\_passage\_from(cx, cy, grid):

514 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

515 random.shuffle(directions)

516

517 for (nx, ny) in directions:

518 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

519 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

520 grid[nx][ny] = 1

521 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

522 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

523

524 maze = [[0] \* width for \_ in range(height)]

525 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

526 maze[start\_x][start\_y] = 1

527 carve\_passage\_from(start\_x, start\_y, maze)

528 return maze

529

530 def prims\_algorithm(height, width):

531 local\_maze = [[0] \* width for \_ in range(height)]

532 visited = set()

533 walls = set()

534 start = (0, 0)

535 visited.add(start)

536 local\_maze[start[0]][start[1]] = 1

537 walls.update({(0, 1), (1, 0)})

538

539 while walls:

540 wall = random.choice(list(walls))

541 x, y = wall

542

543 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

544

545 if len(neighbors) == 1:

546 nx, ny = neighbors[0]

547 local\_maze[x][y] = 1

548 visited.add((x, y))

549

550 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

551 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

552 walls.add((dx, dy))

553

554 walls.remove(wall)

555

556 return local\_maze

557

558 if \_\_name\_\_ == "\_\_main\_\_":

559 root = tk.Tk()

560 app = MazeApplication(root)

561 root.mainloop()

Version 12

Changes Made:

* Improved code structure and organisation
* Added improved comments to increase readability of the code
* Improved admin features
* Increased robustness and error handling

001 *# Imports, tkinter: for GUI, random: for shuffling, PriorityQueue: for A\* search, msgbox: for displaying dialog boxes, sqlite3: for databases, bcrypt: for hashing passwords*

002 import tkinter as tk

003 import random

004 from queue import PriorityQueue

005 import tkinter.messagebox as msgbox

006 import sqlite3

007 import bcrypt

008

009 *### Maze class instantiated: Lines 21, 397, 473*

010 *### UserAuthentication class instantiated: Lines 40, 230*

011 *### MazeApplication class instantiated: Line 529*

012 *### Maze generation algorithms (Recursive Backtracker and Prim's): Lines 487, 504*

013 *### Pathfinding algorithm: Lines 277-317*

014 *### Database creation: Lines 49-66*

015 *### Maze generation and rendering: Lines 19-35*

016 *### Main execution block: Line 527*

017

018 *# Maze class for maze generation and rendering*

019 class Maze:

020 *# Initialises maze dimensions and creates a grid with default wall/unvisited cells*

021 def \_\_init\_\_(self, height, width):

022 self.height = height

023 self.width = width

024 self.maze = [[0] \* width for \_ in range(height)]

025

026 *# Generates maze using Prim's algorithm*

027 def generate(self):

028 self.maze = prims\_algorithm(self.height, self.width)

029

030 *# Renders maze on canvas, black for walls, white for paths*

031 def render(self, canvas):

032 for i in range(self.height):

033 for j in range(self.width):

034 color = "black" if self.maze[i][j] == 0 else "white"

035 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

036

037 *# User authentication class for login and database management*

038 class UserAuthentication:

039 *# Sets up GUI elements and database for user authentication*

040 def \_\_init\_\_(self, root, on\_login\_success):

041 self.root = root

042 self.on\_login\_success = on\_login\_success

043 self.create\_user\_database()

044 self.create\_database()

045 self.show\_login\_form()

046

047 *# Creates user table in SQLite database*

048 @staticmethod

049 def create\_user\_database():

050 conn = sqlite3.connect('mazes.db')

051 c = conn.cursor()

052 c.execute('''CREATE TABLE IF NOT EXISTS users

053 (username TEXT PRIMARY KEY, password\_hash TEXT)''')

054 conn.commit()

055 conn.close()

056

057 *# Creates mazes table in SQLite database*

058 @staticmethod

059 def create\_database():

060 conn = sqlite3.connect('mazes.db')

061 c = conn.cursor()

062 c.execute('''CREATE TABLE IF NOT EXISTS mazes

063 (id INTEGER PRIMARY KEY, height INTEGER, width INTEGER,

064 maze\_type TEXT, maze\_data TEXT, saved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP)''')

065 conn.commit()

066 conn.close()

067

068 *# Hashes password using bcrypt*

069 @staticmethod

070 def hash\_password(password):

071 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

072

073 *# Clears all widgets from root*

074 def clear\_root(self):

075 for widget in self.root.winfo\_children():

076 widget.destroy()

077

078 *# Compares user password with hashed password*

079 def check\_password(self, hashed\_password, user\_password):

080 return bcrypt.checkpw(user\_password.encode('utf-8'), hashed\_password)

081

082 *# Displays login form in root widget*

083 def show\_login\_form(self):

084 self.clear\_root()

085 tk.Label(self.root, text="Username").pack()

086 self.username\_entry = tk.Entry(self.root)

087 self.username\_entry.pack()

088 tk.Label(self.root, text="Password").pack()

089 self.password\_entry = tk.Entry(self.root, show="\*")

090 self.password\_entry.pack()

091 self.show\_password\_button = tk.Button(self.root, text="Show Password", command=self.toggle\_password\_visibility)

092 self.show\_password\_button.pack()

093 tk.Button(self.root, text="Log In", command=self.login).pack()

094 tk.Button(self.root, text="Sign Up", command=self.show\_signup\_form).pack()

095

096 *# Handles user login with admin check, password validation, and database interaction*

097 def login(self):

098 username = self.username\_entry.get()

099 password = self.password\_entry.get()

100 if username == "admin123" and password == "admin123":

101 self.show\_admin\_window()

102 return

103 conn = sqlite3.connect('mazes.db')

104 c = conn.cursor()

105 c.execute("SELECT password\_hash FROM users WHERE username = ?", (username,))

106 result = c.fetchone()

107 if result and self.check\_password(result[0], password):

108 self.on\_login\_success()

109 else:

110 msgbox.showerror("Login Failed", "Invalid username or password")

111 conn.close()

112

113 *# Toggles the visibility of the password in the login form*

114 def toggle\_password\_visibility(self):

115 if self.password\_entry.cget('show') == '\*':

116 self.password\_entry.config(show='')

117 self.show\_password\_button.config(text="Hide Password")

118 else:

119 self.password\_entry.config(show='\*')

120 self.show\_password\_button.config(text="Show Password")

121

122 *# Toggles the visibility of the password in the signup form*

123 def toggle\_signup\_password\_visibility(self):

124 if self.new\_password\_entry.cget('show') == '\*':

125 self.new\_password\_entry.config(show='')

126 self.show\_signup\_password\_button.config(text="Hide Password")

127 else:

128 self.new\_password\_entry.config(show='\*')

129 self.show\_signup\_password\_button.config(text="Show Password")

130

131 *# Sets up and displays the signup form with username and password fields*

132 def show\_signup\_form(self):

133 self.signup\_window = tk.Toplevel(self.root)

134 self.signup\_window.title("Sign Up")

135 tk.Label(self.signup\_window, text="Username").pack()

136 self.new\_username\_entry = tk.Entry(self.signup\_window)

137 self.new\_username\_entry.pack()

138 tk.Label(self.signup\_window, text="Password").pack()

139 self.new\_password\_entry = tk.Entry(self.signup\_window, show="\*")

140 self.new\_password\_entry.pack()

141 self.show\_signup\_password\_button = tk.Button(self.signup\_window, text="Show Password", command=self.toggle\_signup\_password\_visibility)

142 self.show\_signup\_password\_button.pack()

143 tk.Button(self.signup\_window, text="Sign Up", command=self.signup).pack()

144

145 *# Manages admin window creation, user display, and admin actions*

146 def show\_admin\_window(self):

147 if hasattr(self, 'admin\_window') and self.admin\_window.winfo\_exists():

148 for widget in self.admin\_window.winfo\_children():

149 widget.destroy()

150 else:

151 self.admin\_window = tk.Toplevel(self.root)

152 self.admin\_window.title("Admin Panel")

153 scrollable\_frame = tk.Frame(self.admin\_window)

154 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

155 canvas = tk.Canvas(scrollable\_frame)

156 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

157 canvas.configure(yscrollcommand=scrollbar.set)

158 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

159 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

160 inner\_frame = tk.Frame(canvas)

161 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

162 conn = sqlite3.connect('mazes.db')

163 c = conn.cursor()

164 c.execute("SELECT username FROM users WHERE username != 'admin123'")

165 users = c.fetchall()

166 for user in users:

167 username = user[0]

168 user\_frame = tk.Frame(inner\_frame)

169 user\_label = tk.Label(user\_frame, text=username)

170 user\_label.pack(side=tk.LEFT)

171 delete\_button = tk.Button(user\_frame, text="Delete", command=lambda u=username: self.delete\_user(u))

172 delete\_button.pack(side=tk.LEFT)

173 user\_frame.pack()

174 inner\_frame.update\_idletasks()

175 canvas.config(scrollregion=canvas.bbox("all"))

176 tk.Button(self.admin\_window, text="Delete All Users", command=self.delete\_all\_users).pack()

177 tk.Button(self.admin\_window, text="Quit", command=self.admin\_window.destroy).pack()

178

179 *# Deletes all non-admin users after confirmation, updates admin window*

180 def delete\_all\_users(self):

181 conn = sqlite3.connect('mazes.db')

182 c = conn.cursor()

183 c.execute("SELECT COUNT(\*) FROM users WHERE username != 'admin123'")

184 count = c.fetchone()[0]

185 if count == 0:

186 msgbox.showinfo("Delete All Users", "There are no users to delete.")

187 else:

188 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all users?")

189 if response:

190 c.execute("DELETE FROM users WHERE username != 'admin123'")

191 conn.commit()

192 conn.close()

193 self.show\_admin\_window()

194

195 *# Deletes a specific user and refreshes admin window*

196 def delete\_user(self, username):

197 conn = sqlite3.connect('mazes.db')

198 c = conn.cursor()

199 c.execute("DELETE FROM users WHERE username = ?", (username,))

200 conn.commit()

201 conn.close()

202 self.show\_admin\_window()

203

204 *# Handles new user registration with username uniqueness and reserved username check*

205 def signup(self):

206 new\_username = self.new\_username\_entry.get()

207 new\_password = self.new\_password\_entry.get()

208 hashed\_password = UserAuthentication.hash\_password(new\_password)

209 if new\_username == "admin123":

210 msgbox.showerror("Signup Failed", "This username is reserved and cannot be used.")

211 return

212 try:

213 conn = sqlite3.connect('mazes.db')

214 c = conn.cursor()

215 c.execute("INSERT INTO users (username, password\_hash) VALUES (?, ?)", (new\_username, hashed\_password))

216 conn.commit()

217 msgbox.showinfo("Signup Successful", "Account created successfully")

218 self.signup\_window.destroy()

219 except sqlite3.IntegrityError:

220 msgbox.showerror("Signup Failed", "Username already exists")

221 finally:

222 conn.close()

223

224 *# MazeApplication: Manages maze creation, user interactions, pathfinding, and GUI for the maze program.*

225 class MazeApplication:

226 *# Sets up the main application window and user authentication*

227 def \_\_init\_\_(self, root):

228 self.root = root

229 self.root.title("Maze Program")

230 self.user\_auth = UserAuthentication(root, self.on\_login\_success)

231 self.maze\_type = "Perfect"

232

233 *# Displays main menu upon successful login*

234 def on\_login\_success(self):

235 self.main\_menu()

236

237 *# Deletes a specific maze by ID and refreshes maze list*

238 def delete\_maze(self, maze\_id):

239 conn = sqlite3.connect('mazes.db')

240 c = conn.cursor()

241 c.execute("DELETE FROM mazes WHERE id = ?", (maze\_id,))

242 conn.commit()

243 conn.close()

244 self.my\_mazes()

245

246 *# Deletes all mazes after user confirmation, updates maze list*

247 def delete\_all\_mazes(self):

248 conn = sqlite3.connect('mazes.db')

249 c = conn.cursor()

250 c.execute("SELECT COUNT(\*) FROM mazes")

251 count = c.fetchone()[0]

252 if count == 0:

253 msgbox.showinfo("Delete All", "There are no mazes to delete.")

254 else:

255 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all mazes?")

256 if response:

257 conn = sqlite3.connect('mazes.db')

258 c = conn.cursor()

259 c.execute("DELETE FROM mazes")

260 conn.commit()

261 conn.close()

262 self.my\_mazes()

263

264 *# Saves current maze state to the database*

265 def save\_current\_maze(self):

266 conn = sqlite3.connect('mazes.db')

267 c = conn.cursor()

268 maze\_str = ','.join([' '.join(map(str, row)) for row in self.maze.maze])

269 c.execute("INSERT INTO mazes (height, width, maze\_type, maze\_data) VALUES (?, ?, ?, ?)",

270 (self.current\_height, self.current\_width, self.current\_maze\_type, maze\_str))

271 conn.commit()

272 conn.close()

273 msgbox.showinfo("Success", "Successfully saved the maze")

274

275 *# A\* search algorithm for pathfinding in maze*

276 @staticmethod

277 def a\_star\_search(maze, start, end):

278 def heuristic(a, b):

279 return abs(a[0] - b[0]) + abs(a[1] - b[1])

280

281 def get\_neighbors(pos):

282 neighbors = []

283 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

284 x, y = pos[0] + dx, pos[1] + dy

285 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

286 neighbors.append((x, y))

287 return neighbors

288

289 frontier = PriorityQueue()

290 frontier.put((0, start))

291 came\_from = {}

292 cost\_so\_far = {}

293 came\_from[start] = None

294 cost\_so\_far[start] = 0

295

296 while not frontier.empty():

297 current = frontier.get()[1]

298 if current == end:

299 break

300 for next in get\_neighbors(current):

301 new\_cost = cost\_so\_far[current] + 1

302 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

303 cost\_so\_far[next] = new\_cost

304 priority = new\_cost + heuristic(end, next)

305 frontier.put((priority, next))

306 came\_from[next] = current

307

308 current = end

309 path = []

310 while current != start:

311 if current not in came\_from:

312 return None

313 path.append(current)

314 current = came\_from[current]

315 path.append(start)

316 path.reverse()

317 return path

318

319 *# Finds and displays a path in the maze using A\* search*

320 def find\_path(self, height, width):

321 start = (0, 0)

322 end = (height - 1, width - 1)

323 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

324 if path is not None:

325 self.show\_path(path)

326 else:

327 print("No path found!")

328

329 *# Visually represents a found path on the maze*

330 def show\_path(self, path):

331 if path:

332 for (x, y) in path:

333 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

334 self.canvas.update()

335 else:

336 print("No path to show.")

337

338 *# Updates the maze type (Perfect/Non-Perfect)*

339 def update\_maze\_type(self, maze\_type):

340 self.maze\_type = maze\_type

341

342 *# Sets application window size based on maze dimensions*

343 def set\_window\_size(self, width, height):

344 canvas\_width = width \* 10

345 canvas\_height = height \* 10

346 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

347

348 *# Clears all widgets from the root window*

349 def clear\_root(self):

350 for widget in self.root.winfo\_children():

351 widget.destroy()

352

353 *# Displays the main menu with options for maze creation and viewing*

354 def main\_menu(self):

355 self.clear\_root()

356 self.root.geometry("300x200")

357 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

358 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

359 self.add\_quit\_button()

360

361 *# Randomizes maze dimensions and generates the maze*

362 def randomize\_and\_generate(self):

363 self.height\_slider.set(random.randint(15, 76))

364 self.width\_slider.set(random.randint(15, 76))

365 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

366

367 *# Displays the maze generation menu with dimension controls and maze type selection*

368 def make\_maze\_menu(self):

369 self.clear\_root()

370 self.root.geometry("300x400")

371 tk.Label(self.root, text="Maze Height:").pack()

372 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

373 self.height\_slider.pack()

374 tk.Label(self.root, text="Maze Width:").pack()

375 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

376 self.width\_slider.pack()

377 maze\_type = tk.StringVar(self.root)

378 maze\_type.set(self.maze\_type)

379 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

380 tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate).pack()

381 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

382 self.add\_go\_back\_button()

383 self.add\_quit\_button()

384

385 *# Updates window size based on the maze dimensions*

386 def update\_size(self, event=None):

387 self.set\_window\_size(int(self.width\_slider.get()), int(self.height\_slider.get()))

388

389 *# Regenerates and displays a saved maze from the database*

390 def regenerate\_saved\_maze(self, maze\_id):

391 conn = sqlite3.connect('mazes.db')

392 c = conn.cursor()

393 c.execute("SELECT height, width, maze\_data FROM mazes WHERE id = ?", (maze\_id,))

394 height, width, maze\_str = c.fetchone()

395 conn.close()

396 maze\_array = [list(map(int, row.split())) for row in maze\_str.split(',')]

397 self.maze = Maze(height, width)

398 self.maze.maze = maze\_array

399 self.display\_maze(height, width)

400

401 *# Displays saved mazes with options for regeneration and deletion*

402 def my\_mazes(self):

403 self.clear\_root()

404 self.add\_go\_back\_button()

405 self.add\_quit\_button()

406 tk.Button(self.root, text="Delete All", command=self.delete\_all\_mazes).pack()

407 conn = sqlite3.connect('mazes.db')

408 c = conn.cursor()

409 c.execute("SELECT id, height, width, saved\_at FROM mazes")

410 saved\_mazes = c.fetchall()

411 conn.close()

412 scrollable\_frame = tk.Frame(self.root)

413 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

414 canvas = tk.Canvas(scrollable\_frame)

415 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

416 canvas.configure(yscrollcommand=scrollbar.set)

417 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

418 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

419 inner\_frame = tk.Frame(canvas)

420 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

421 for maze in saved\_mazes:

422 maze\_id, height, width, saved\_at = maze

423 maze\_frame = tk.Frame(inner\_frame)

424 tk.Label(maze\_frame, text=f"Maze ID: {maze\_id}, Size: {height}x{width}, Saved: {saved\_at}").pack(side=tk.LEFT)

425 tk.Button(maze\_frame, text="Generate", command=lambda m\_id=maze\_id: self.regenerate\_saved\_maze(m\_id)).pack(side=tk.LEFT)

426 tk.Button(maze\_frame, text="Delete", command=lambda m\_id=maze\_id: self.delete\_maze(m\_id)).pack(side=tk.LEFT)

427 maze\_frame.pack()

428 inner\_frame.update\_idletasks()

429 canvas.config(scrollregion=canvas.bbox("all"))

430

431 *# Returns to the main menu*

432 def go\_back(self):

433 self.main\_menu()

434

435 *# Adds a 'Go Back' button specific to the maze generation interface*

436 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

437 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

438

439 *# Quits the application*

440 def quit\_app(self):

441 self.root.quit()

442

443 *# Adds a generic 'Go Back' button*

444 def add\_go\_back\_button(self):

445 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

446

447 *# Adds a 'Quit' button to the interface*

448 def add\_quit\_button(self):

449 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

450

451 *# Displays the maze with interaction options*

452 def display\_maze(self, height, width):

453 self.clear\_root()

454 canvas\_width, canvas\_height = width \* 10, height \* 10

455 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

456 self.canvas = canvas

457 canvas.pack(side=tk.TOP)

458 self.maze.render(canvas)

459 button\_frame = tk.Frame(self.root)

460 button\_frame.pack(after=canvas)

461 tk.Button(button\_frame, text="Save", command=self.save\_current\_maze).pack(side=tk.LEFT)

462 tk.Button(button\_frame, text="Quit", command=self.quit\_app).pack(side=tk.LEFT)

463 tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type)).pack(side=tk.LEFT)

464 tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width)).pack(side=tk.LEFT)

465 self.add\_go\_back\_button\_generate\_maze(button\_frame)

466

467 *# Generates a new maze based on dimensions and type*

468 def generate\_maze(self, height, width, maze\_type):

469 self.current\_height, self.current\_width, self.current\_maze\_type = height, width, maze\_type

470 valid\_maze, maze\_generation\_attempts = False, 0

471 while not valid\_maze and maze\_generation\_attempts < 250:

472 maze\_generation\_attempts += 1

473 self.maze = Maze(height, width)

474 self.maze.maze = recursive\_backtracker(height, width) if maze\_type == "Perfect" else prims\_algorithm(height, width)

475 if self.a\_star\_search(self.maze.maze, (0, 0), (height - 1, width - 1)):

476 valid\_maze = True

477 else:

478 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

479 if valid\_maze:

480 self.display\_maze(height, width)

481 else:

482 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze. Please try different dimensions or regenerate.")

483 print("Could not generate a valid maze. Please try again.")

484 self.make\_maze\_menu()

485

486 *# Recursive Backtracker algorithm for maze generation*

487 def recursive\_backtracker(height, width):

488 def carve\_passage\_from(cx, cy, grid):

489 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

490 random.shuffle(directions)

491 for (nx, ny) in directions:

492 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

493 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

494 grid[nx][ny] = 1

495 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

496 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

497 maze = [[0] \* width for \_ in range(height)]

498 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

499 maze[start\_x][start\_y] = 1

500 carve\_passage\_from(start\_x, start\_y, maze)

501 return maze

502

503 *# Prim's algorithm for maze generation*

504 def prims\_algorithm(height, width):

505 local\_maze = [[0] \* width for \_ in range(height)]

506 visited = set()

507 walls = set()

508 start = (0, 0)

509 visited.add(start)

510 local\_maze[start[0]][start[1]] = 1

511 walls.update({(0, 1), (1, 0)})

512 while walls:

513 wall = random.choice(list(walls))

514 x, y = wall

515 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

516 if len(neighbors) == 1:

517 nx, ny = neighbors[0]

518 local\_maze[x][y] = 1

519 visited.add((x, y))

520 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

521 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

522 walls.add((dx, dy))

523 walls.remove(wall)

524 return local\_maze

525

526 *# Main execution block*

527 if \_\_name\_\_ == "\_\_main\_\_":

528 root = tk.Tk()

529 app = MazeApplication(root)

530 root.mainloop()

Version 13 (Final code)

Changes made:

* Added restrictions to not allow more than 1 signup window to be created and more than 1 error dialogue to be created
* Customised error dialogue to not have it use windows error dialogue, but rather created my own
* Added restrictions to how the username and password are allowed to be input, you won’t be able to input more than 16 characters or less than 3 characters in either, also all characters have to be alphanumeric , underscore and dash are also allowed. Example : a-z, A-Z, 0-9, -, \_

001 *# Imports, tkinter: for GUI, random: for shuffling, PriorityQueue: for A\* search, msgbox: for displaying dialog boxes, sqlite3: for databases, bcrypt: for hashing passwords*

002 import tkinter as tk

003 import random

004 from queue import PriorityQueue

005 import tkinter.messagebox as msgbox

006 import sqlite3

007 import bcrypt

008

009 *### Maze class instantiated: Lines 19, 518, 442*

010 *### UserAuthentication class instantiated: Lines 38, 275*

011 *### MazeApplication class instantiated: Line 574*

012 *### Maze generation algorithms (Recursive Backtracker and Prim's): Lines 532, 549*

013 *### Pathfinding algorithm: Lines 322-362*

014 *### Database creation: Lines 49-68*

015 *### Maze generation and rendering: Lines 18-35*

016 *### Main execution block: Line 572*

017

018 *# Maze class for maze generation and rendering*

019 class Maze:

020 *# Initialises maze dimensions and creates a grid with default wall/unvisited cells*

021 def \_\_init\_\_(self, height, width):

022 self.height = height

023 self.width = width

024 self.maze = [[0] \* width for \_ in range(height)]

025

026 *# Generates maze using Prim's algorithm*

027 def generate(self):

028 self.maze = prims\_algorithm(self.height, self.width)

029

030 *# Renders maze on canvas, black for walls, white for paths*

031 def render(self, canvas):

032 for i in range(self.height):

033 for j in range(self.width):

034 color = "black" if self.maze[i][j] == 0 else "white"

035 canvas.create\_rectangle(j\*10, i\*10, (j+1)\*10, (i+1)\*10, fill=color)

036

037 *# User authentication class for login and database management*

038 class UserAuthentication:

039 *# Sets up GUI elements and database for user authentication*

040 def \_\_init\_\_(self, root, on\_login\_success):

041 self.root = root

042 self.on\_login\_success = on\_login\_success

043 self.signup\_window = None

044 self.error\_window = None

045 self.create\_user\_database()

046 self.create\_database()

047 self.show\_login\_form()

048

049 *# Creates user table in SQLite database*

050 @staticmethod

051 def create\_user\_database():

052 conn = sqlite3.connect('mazes.db')

053 c = conn.cursor()

054 c.execute('''CREATE TABLE IF NOT EXISTS users

055 (username TEXT PRIMARY KEY, password\_hash TEXT)''')

056 conn.commit()

057 conn.close()

058

059 *# Creates mazes table in SQLite database*

060 @staticmethod

061 def create\_database():

062 conn = sqlite3.connect('mazes.db')

063 c = conn.cursor()

064 c.execute('''CREATE TABLE IF NOT EXISTS mazes

065 (id INTEGER PRIMARY KEY, height INTEGER, width INTEGER,

066 maze\_type TEXT, maze\_data TEXT, saved\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP)''')

067 conn.commit()

068 conn.close()

069

070 *# Hashes password using bcrypt*

071 @staticmethod

072 def hash\_password(password):

073 return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

074

075 *# Clears all widgets from root*

076 def clear\_root(self):

077 for widget in self.root.winfo\_children():

078 widget.destroy()

079

080 *# Compares user password with hashed password*

081 def check\_password(self, hashed\_password, user\_password):

082 return bcrypt.checkpw(user\_password.encode('utf-8'), hashed\_password)

083

084 *# Displays login form in root widget*

085 def show\_login\_form(self):

086 self.clear\_root()

087 tk.Label(self.root, text="Username").pack()

088 self.username\_entry = tk.Entry(self.root)

089 self.username\_entry.pack()

090 tk.Label(self.root, text="Password").pack()

091 self.password\_entry = tk.Entry(self.root, show="\*")

092 self.password\_entry.pack()

093 self.show\_password\_button = tk.Button(self.root, text="Show Password", command=self.toggle\_password\_visibility)

094 self.show\_password\_button.pack()

095 tk.Button(self.root, text="Log In", command=self.login).pack()

096 tk.Button(self.root, text="Sign Up", command=self.show\_signup\_form).pack()

097

098 *# Handles user login with admin check, password validation, and database interaction*

099 def login(self):

100 username = self.username\_entry.get()

101 password = self.password\_entry.get()

102 if username == "admin123" and password == "admin123":

103 self.show\_admin\_window()

104 return

105 conn = sqlite3.connect('mazes.db')

106 c = conn.cursor()

107 c.execute("SELECT password\_hash FROM users WHERE username = ?", (username,))

108 result = c.fetchone()

109 if result and self.check\_password(result[0], password):

110 self.on\_login\_success()

111 else:

112 msgbox.showerror("Login Failed", "Invalid username or password")

113 conn.close()

114

115 *# Toggles the visibility of the password in the login form*

116 def toggle\_password\_visibility(self):

117 if self.password\_entry.cget('show') == '\*':

118 self.password\_entry.config(show='')

119 self.show\_password\_button.config(text="Hide Password")

120 else:

121 self.password\_entry.config(show='\*')

122 self.show\_password\_button.config(text="Show Password")

123

124 *# Toggles the visibility of the password in the signup form*

125 def toggle\_signup\_password\_visibility(self):

126 if self.new\_password\_entry.cget('show') == '\*':

127 self.new\_password\_entry.config(show='')

128 self.show\_signup\_password\_button.config(text="Hide Password")

129 else:

130 self.new\_password\_entry.config(show='\*')

131 self.show\_signup\_password\_button.config(text="Show Password")

132

133 *# Sets up and displays the signup form with username and password fields*

134 def show\_signup\_form(self):

135 if self.signup\_window and self.signup\_window.winfo\_exists():

136 return

137 self.signup\_window = tk.Toplevel(self.root)

138 self.signup\_window.title("Sign Up")

139 tk.Label(self.signup\_window, text="Username").pack()

140 self.new\_username\_entry = tk.Entry(self.signup\_window)

141 self.new\_username\_entry.pack()

142 tk.Label(self.signup\_window, text="Password").pack()

143 self.new\_password\_entry = tk.Entry(self.signup\_window, show="\*")

144 self.new\_password\_entry.pack()

145 self.show\_signup\_password\_button = tk.Button(self.signup\_window, text="Show Password", command=self.toggle\_signup\_password\_visibility)

146 self.show\_signup\_password\_button.pack()

147 tk.Button(self.signup\_window, text="Sign Up", command=self.signup).pack()

148 self.signup\_window.protocol("WM\_DELETE\_WINDOW", self.on\_signup\_window\_close)

149

150 *# Closes the signup window*

151 def on\_signup\_window\_close(self):

152 self.signup\_window.destroy()

153 self.signup\_window = None

154

155 *# Manages admin window creation, user display, and admin actions*

156 def show\_admin\_window(self):

157 if hasattr(self, 'admin\_window') and self.admin\_window.winfo\_exists():

158 for widget in self.admin\_window.winfo\_children():

159 widget.destroy()

160 else:

161 self.admin\_window = tk.Toplevel(self.root)

162 self.admin\_window.title("Admin Panel")

163 scrollable\_frame = tk.Frame(self.admin\_window)

164 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

165 canvas = tk.Canvas(scrollable\_frame)

166 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

167 canvas.configure(yscrollcommand=scrollbar.set)

168 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

169 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

170 inner\_frame = tk.Frame(canvas)

171 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

172 conn = sqlite3.connect('mazes.db')

173 c = conn.cursor()

174 c.execute("SELECT username FROM users WHERE username != 'admin123'")

175 users = c.fetchall()

176 for user in users:

177 username = user[0]

178 user\_frame = tk.Frame(inner\_frame)

179 user\_label = tk.Label(user\_frame, text=username)

180 user\_label.pack(side=tk.LEFT)

181 delete\_button = tk.Button(user\_frame, text="Delete", command=lambda u=username: self.delete\_user(u))

182 delete\_button.pack(side=tk.LEFT)

183 user\_frame.pack()

184 inner\_frame.update\_idletasks()

185 canvas.config(scrollregion=canvas.bbox("all"))

186 tk.Button(self.admin\_window, text="Delete All Users", command=self.delete\_all\_users).pack()

187 tk.Button(self.admin\_window, text="Quit", command=self.admin\_window.destroy).pack()

188

189 *# Deletes all non-admin users after confirmation, updates admin window*

190 def delete\_all\_users(self):

191 conn = sqlite3.connect('mazes.db')

192 c = conn.cursor()

193 c.execute("SELECT COUNT(\*) FROM users WHERE username != 'admin123'")

194 count = c.fetchone()[0]

195 if count == 0:

196 msgbox.showinfo("Delete All Users", "There are no users to delete.")

197 else:

198 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all users?")

199 if response:

200 c.execute("DELETE FROM users WHERE username != 'admin123'")

201 conn.commit()

202 conn.close()

203 self.show\_admin\_window()

204

205 *# Deletes a specific user and refreshes admin window*

206 def delete\_user(self, username):

207 conn = sqlite3.connect('mazes.db')

208 c = conn.cursor()

209 c.execute("DELETE FROM users WHERE username = ?", (username,))

210 conn.commit()

211 conn.close()

212 self.show\_admin\_window()

213

214 *# Handles new user registration with username uniqueness and reserved username check*

215 def signup(self):

216 new\_username = self.new\_username\_entry.get()

217 new\_password = self.new\_password\_entry.get()

218 if not self.validate\_credentials(new\_username, new\_password):

219 return

220 hashed\_password = UserAuthentication.hash\_password(new\_password)

221 if new\_username == "admin123":

222 msgbox.showerror("Signup Failed", "This username is reserved and cannot be used.")

223 return

224 try:

225 conn = sqlite3.connect('mazes.db')

226 c = conn.cursor()

227 c.execute("INSERT INTO users (username, password\_hash) VALUES (?, ?)", (new\_username, hashed\_password))

228 conn.commit()

229 msgbox.showinfo("Signup Successful", "Account created successfully")

230 self.signup\_window.destroy()

231 except sqlite3.IntegrityError:

232 msgbox.showerror("Signup Failed", "Username already exists")

233 finally:

234 conn.close()

235

236 *# Validates the input string based on length and allowed characters*

237 def is\_valid\_input(self, input\_string):

238 if not (3 <= len(input\_string) <= 16):

239 return False

240 return all(char.isalnum() or char in '-\_' for char in input\_string)

241

242 *# Shows a custom error dialog with the given message*

243 def show\_error\_dialog(self, message):

244 if self.error\_window and self.error\_window.winfo\_exists():

245 self.error\_window.destroy()

246 self.error\_window = tk.Toplevel(self.root)

247 self.error\_window.title("Error")

248 tk.Label(self.error\_window, text=message).pack(padx=10, pady=10)

249 tk.Button(self.error\_window, text="OK", command=self.error\_window.destroy).pack(pady=(0, 10))

250

251 *# Validates both username and password*

252 def validate\_credentials(self, username, password):

253 valid\_username = self.is\_valid\_input(username)

254 valid\_password = self.is\_valid\_input(password)

255 error\_message = "Invalid input:\n"

256 if not valid\_username or not valid\_password:

257 if not all(char.isalnum() or char in '-\_' for char in username):

258 error\_message += "- Username can only contain a-z, A-Z, 0-9, -, and \_.\n"

259 if not all(char.isalnum() or char in '-\_' for char in password):

260 error\_message += "- Password can only contain a-z, A-Z, 0-9, -, and \_.\n"

261 if not (3 <= len(username) <= 16):

262 error\_message += "- Username must be 3 to 16 characters long.\n"

263 if not (3 <= len(password) <= 16):

264 error\_message += "- Password must be 3 to 16 characters long."

265 self.show\_error\_dialog(error\_message)

266 return False

267 return True

268

269 *# Manages maze creation, user interactions, pathfinding, and GUI for the maze program*

270 class MazeApplication:

271 *# Sets up the main application window and user authentication*

272 def \_\_init\_\_(self, root):

273 self.root = root

274 self.root.title("Maze Program")

275 self.user\_auth = UserAuthentication(root, self.on\_login\_success)

276 self.maze\_type = "Perfect"

277

278 *# Displays main menu upon successful login*

279 def on\_login\_success(self):

280 self.main\_menu()

281

282 *# Deletes a specific maze by ID and refreshes maze list*

283 def delete\_maze(self, maze\_id):

284 conn = sqlite3.connect('mazes.db')

285 c = conn.cursor()

286 c.execute("DELETE FROM mazes WHERE id = ?", (maze\_id,))

287 conn.commit()

288 conn.close()

289 self.my\_mazes()

290

291 *# Deletes all mazes after user confirmation, updates maze list*

292 def delete\_all\_mazes(self):

293 conn = sqlite3.connect('mazes.db')

294 c = conn.cursor()

295 c.execute("SELECT COUNT(\*) FROM mazes")

296 count = c.fetchone()[0]

297 if count == 0:

298 msgbox.showinfo("Delete All", "There are no mazes to delete.")

299 else:

300 response = msgbox.askyesno("Confirm", "Are you sure you want to delete all mazes?")

301 if response:

302 conn = sqlite3.connect('mazes.db')

303 c = conn.cursor()

304 c.execute("DELETE FROM mazes")

305 conn.commit()

306 conn.close()

307 self.my\_mazes()

308

309 *# Saves current maze state to the database*

310 def save\_current\_maze(self):

311 conn = sqlite3.connect('mazes.db')

312 c = conn.cursor()

313 maze\_str = ','.join([' '.join(map(str, row)) for row in self.maze.maze])

314 c.execute("INSERT INTO mazes (height, width, maze\_type, maze\_data) VALUES (?, ?, ?, ?)",

315 (self.current\_height, self.current\_width, self.current\_maze\_type, maze\_str))

316 conn.commit()

317 conn.close()

318 msgbox.showinfo("Success", "Successfully saved the maze")

319

320 *# A\* search algorithm for pathfinding in maze*

321 @staticmethod

322 def a\_star\_search(maze, start, end):

323 def heuristic(a, b):

324 return abs(a[0] - b[0]) + abs(a[1] - b[1])

325

326 def get\_neighbors(pos):

327 neighbors = []

328 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

329 x, y = pos[0] + dx, pos[1] + dy

330 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] == 1:

331 neighbors.append((x, y))

332 return neighbors

333

334 frontier = PriorityQueue()

335 frontier.put((0, start))

336 came\_from = {}

337 cost\_so\_far = {}

338 came\_from[start] = None

339 cost\_so\_far[start] = 0

340

341 while not frontier.empty():

342 current = frontier.get()[1]

343 if current == end:

344 break

345 for next in get\_neighbors(current):

346 new\_cost = cost\_so\_far[current] + 1

347 if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

348 cost\_so\_far[next] = new\_cost

349 priority = new\_cost + heuristic(end, next)

350 frontier.put((priority, next))

351 came\_from[next] = current

352

353 current = end

354 path = []

355 while current != start:

356 if current not in came\_from:

357 return None

358 path.append(current)

359 current = came\_from[current]

360 path.append(start)

361 path.reverse()

362 return path

363

364 *# Finds and displays a path in the maze using A\* search*

365 def find\_path(self, height, width):

366 start = (0, 0)

367 end = (height - 1, width - 1)

368 path = MazeApplication.a\_star\_search(self.maze.maze, start, end)

369 if path is not None:

370 self.show\_path(path)

371 else:

372 print("No path found!")

373

374 *# Visually represents a found path on the maze*

375 def show\_path(self, path):

376 if path:

377 for (x, y) in path:

378 self.canvas.create\_rectangle(y\*10, x\*10, (y+1)\*10, (x+1)\*10, fill="blue")

379 self.canvas.update()

380 else:

381 print("No path to show.")

382

383 *# Updates the maze type (Perfect/Non-Perfect)*

384 def update\_maze\_type(self, maze\_type):

385 self.maze\_type = maze\_type

386

387 *# Sets application window size based on maze dimensions*

388 def set\_window\_size(self, width, height):

389 canvas\_width = width \* 10

390 canvas\_height = height \* 10

391 self.root.geometry(f"{canvas\_width}x{canvas\_height + 50}")

392

393 *# Clears all widgets from the root window*

394 def clear\_root(self):

395 for widget in self.root.winfo\_children():

396 widget.destroy()

397

398 *# Displays the main menu with options for maze creation and viewing*

399 def main\_menu(self):

400 self.clear\_root()

401 self.root.geometry("300x200")

402 tk.Button(self.root, text="Make Maze", command=self.make\_maze\_menu).pack()

403 tk.Button(self.root, text="My Mazes", command=self.my\_mazes).pack()

404 self.add\_quit\_button()

405

406 *# Randomizes maze dimensions and generates the maze*

407 def randomize\_and\_generate(self):

408 self.height\_slider.set(random.randint(15, 76))

409 self.width\_slider.set(random.randint(15, 76))

410 self.generate\_maze(self.height\_slider.get(), self.width\_slider.get(), self.maze\_type)

411

412 *# Displays the maze generation menu with dimension controls and maze type selection*

413 def make\_maze\_menu(self):

414 self.clear\_root()

415 self.root.geometry("300x400")

416 tk.Label(self.root, text="Maze Height:").pack()

417 self.height\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

418 self.height\_slider.pack()

419 tk.Label(self.root, text="Maze Width:").pack()

420 self.width\_slider = tk.Scale(self.root, from\_=15, to=75, orient="horizontal", label="15 to 75 cells")

421 self.width\_slider.pack()

422 maze\_type = tk.StringVar(self.root)

423 maze\_type.set(self.maze\_type)

424 tk.OptionMenu(self.root, maze\_type, "Perfect", "Non-Perfect", command=self.update\_maze\_type).pack()

425 tk.Button(self.root, text="Randomize", command=self.randomize\_and\_generate).pack()

426 tk.Button(self.root, text="Generate Maze", command=lambda: self.generate\_maze(int(self.height\_slider.get()), int(self.width\_slider.get()), maze\_type.get())).pack()

427 self.add\_go\_back\_button()

428 self.add\_quit\_button()

429

430 *# Updates window size based on the maze dimensions*

431 def update\_size(self, event=None):

432 self.set\_window\_size(int(self.width\_slider.get()), int(self.height\_slider.get()))

433

434 *# Regenerates and displays a saved maze from the database*

435 def regenerate\_saved\_maze(self, maze\_id):

436 conn = sqlite3.connect('mazes.db')

437 c = conn.cursor()

438 c.execute("SELECT height, width, maze\_data FROM mazes WHERE id = ?", (maze\_id,))

439 height, width, maze\_str = c.fetchone()

440 conn.close()

441 maze\_array = [list(map(int, row.split())) for row in maze\_str.split(',')]

442 self.maze = Maze(height, width)

443 self.maze.maze = maze\_array

444 self.display\_maze(height, width)

445

446 *# Displays saved mazes with options for regeneration and deletion*

447 def my\_mazes(self):

448 self.clear\_root()

449 self.add\_go\_back\_button()

450 self.add\_quit\_button()

451 tk.Button(self.root, text="Delete All", command=self.delete\_all\_mazes).pack()

452 conn = sqlite3.connect('mazes.db')

453 c = conn.cursor()

454 c.execute("SELECT id, height, width, saved\_at FROM mazes")

455 saved\_mazes = c.fetchall()

456 conn.close()

457 scrollable\_frame = tk.Frame(self.root)

458 scrollable\_frame.pack(fill=tk.BOTH, expand=True)

459 canvas = tk.Canvas(scrollable\_frame)

460 scrollbar = tk.Scrollbar(scrollable\_frame, orient="vertical", command=canvas.yview)

461 canvas.configure(yscrollcommand=scrollbar.set)

462 scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

463 canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

464 inner\_frame = tk.Frame(canvas)

465 canvas.create\_window((0, 0), window=inner\_frame, anchor='nw')

466 for maze in saved\_mazes:

467 maze\_id, height, width, saved\_at = maze

468 maze\_frame = tk.Frame(inner\_frame)

469 tk.Label(maze\_frame, text=f"Maze ID: {maze\_id}, Size: {height}x{width}, Saved: {saved\_at}").pack(side=tk.LEFT)

470 tk.Button(maze\_frame, text="Generate", command=lambda m\_id=maze\_id: self.regenerate\_saved\_maze(m\_id)).pack(side=tk.LEFT)

471 tk.Button(maze\_frame, text="Delete", command=lambda m\_id=maze\_id: self.delete\_maze(m\_id)).pack(side=tk.LEFT)

472 maze\_frame.pack()

473 inner\_frame.update\_idletasks()

474 canvas.config(scrollregion=canvas.bbox("all"))

475

476 *# Returns to the main menu*

477 def go\_back(self):

478 self.main\_menu()

479

480 *# Adds a 'Go Back' button specific to the maze generation interface*

481 def add\_go\_back\_button\_generate\_maze(self, button\_frame):

482 tk.Button(button\_frame, text="Go Back", command=self.make\_maze\_menu).pack(side=tk.BOTTOM)

483

484 *# Quits the application*

485 def quit\_app(self):

486 self.root.quit()

487

488 *# Adds a 'Go Back' button*

489 def add\_go\_back\_button(self):

490 tk.Button(self.root, text="Go Back", command=self.go\_back).pack()

491

492 *# Adds a 'Quit' button*

493 def add\_quit\_button(self):

494 tk.Button(self.root, text="Quit", command=self.quit\_app).pack()

495

496 *# Displays the maze with interaction options*

497 def display\_maze(self, height, width):

498 self.clear\_root()

499 canvas\_width, canvas\_height = width \* 10, height \* 10

500 canvas = tk.Canvas(self.root, width=canvas\_width, height=canvas\_height)

501 self.canvas = canvas

502 canvas.pack(side=tk.TOP)

503 self.maze.render(canvas)

504 button\_frame = tk.Frame(self.root)

505 button\_frame.pack(after=canvas)

506 tk.Button(button\_frame, text="Save", command=self.save\_current\_maze).pack(side=tk.LEFT)

507 tk.Button(button\_frame, text="Quit", command=self.quit\_app).pack(side=tk.LEFT)

508 tk.Button(button\_frame, text="Regenerate Maze", command=lambda: self.generate\_maze(self.current\_height, self.current\_width, self.current\_maze\_type)).pack(side=tk.LEFT)

509 tk.Button(button\_frame, text="Find Path", command=lambda: self.find\_path(height, width)).pack(side=tk.LEFT)

510 self.add\_go\_back\_button\_generate\_maze(button\_frame)

511

512 *# Generates a new maze based on dimensions and type*

513 def generate\_maze(self, height, width, maze\_type):

514 self.current\_height, self.current\_width, self.current\_maze\_type = height, width, maze\_type

515 valid\_maze, maze\_generation\_attempts = False, 0

516 while not valid\_maze and maze\_generation\_attempts < 250:

517 maze\_generation\_attempts += 1

518 self.maze = Maze(height, width)

519 self.maze.maze = recursive\_backtracker(height, width) if maze\_type == "Perfect" else prims\_algorithm(height, width)

520 if self.a\_star\_search(self.maze.maze, (0, 0), (height - 1, width - 1)):

521 valid\_maze = True

522 else:

523 print(f"Maze generation attempt {maze\_generation\_attempts} failed. No path found.")

524 if valid\_maze:

525 self.display\_maze(height, width)

526 else:

527 msgbox.showwarning("Maze Generation Failed", "Unable to generate a solvable maze. Please try different dimensions or regenerate.")

528 print("Could not generate a valid maze. Please try again.")

529 self.make\_maze\_menu()

530

531 *# Recursive backtracker algorithm for maze generation*

532 def recursive\_backtracker(height, width):

533 def carve\_passage\_from(cx, cy, grid):

534 directions = [(cx - 1, cy), (cx + 1, cy), (cx, cy - 1), (cx, cy + 1)]

535 random.shuffle(directions)

536 for (nx, ny) in directions:

537 if 0 <= nx < height and 0 <= ny < width and grid[nx][ny] == 0:

538 if 0 <= nx + (nx - cx) < height and 0 <= ny + (ny - cy) < width and grid[nx + (nx - cx)][ny + (ny - cy)] == 0:

539 grid[nx][ny] = 1

540 grid[nx + (nx - cx)][ny + (ny - cy)] = 1

541 carve\_passage\_from(nx + (nx - cx), ny + (ny - cy), grid)

542 maze = [[0] \* width for \_ in range(height)]

543 start\_x, start\_y = random.randint(0, height - 1), random.randint(0, width - 1)

544 maze[start\_x][start\_y] = 1

545 carve\_passage\_from(start\_x, start\_y, maze)

546 return maze

547

548 *# Prim's algorithm for maze generation*

549 def prims\_algorithm(height, width):

550 local\_maze = [[0] \* width for \_ in range(height)]

551 visited = set()

552 walls = set()

553 start = (0, 0)

554 visited.add(start)

555 local\_maze[start[0]][start[1]] = 1

556 walls.update({(0, 1), (1, 0)})

557 while walls:

558 wall = random.choice(list(walls))

559 x, y = wall

560 neighbors = [(nx, ny) for nx, ny in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] if (nx, ny) in visited]

561 if len(neighbors) == 1:

562 nx, ny = neighbors[0]

563 local\_maze[x][y] = 1

564 visited.add((x, y))

565 for dx, dy in [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]:

566 if 0 <= dx < height and 0 <= dy < width and (dx, dy) not in visited:

567 walls.add((dx, dy))

568 walls.remove(wall)

569 return local\_maze

570

571 *# Main execution block*

572 if \_\_name\_\_ == "\_\_main\_\_":

573 root = tk.Tk()

574 app = MazeApplication(root)

575 root.mainloop()

aw