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()