

|  |
| --- |
| **RAJALAKSHMI INSTITUTE OF TECHNOLOGY** |
| (An Autonomous Institution, Affiliated to Anna University, Chennai) |

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | 2117240070006 |
| **NAME** | ABINAYA S |
| **PROJECT TITLE** | Snake and Ladder Solver using BFS |
| **DATE OF SUBMISSION** |  |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**🧠 INTRODUCTION**

Artificial Intelligence (AI) allows computers to perform tasks requiring human-like intelligence such as reasoning, problem-solving, and decision-making.  
Breadth-First Search (BFS) is a fundamental AI algorithm that explores all possible paths level by level to find the shortest route in a graph.

“Snake and Ladder” is a popular board game where players move based on dice rolls. Snakes send players backward and ladders move them forward — making it an ideal scenario for applying BFS to compute the shortest path to victory.

**🧩 PROBLEM STATEMENT**

Design and implement an AI-based system that calculates the minimum number of dice throws required to reach the final cell (100) in a Snake and Ladder game, taking into account all the snakes and ladders on the board.

**🎯 GOALS**

* Compute the shortest path from start (1) to finish (100).
* Demonstrate the efficiency of BFS in pathfinding.
* Visualize and simulate the board using an interactive GUI.
* Provide insights into optimal strategies through AI simulation.

**Expected Result: Minimum dice throws required to reach cell 100, along with the path taken.**

**🧮 THEORETICAL BACKGROUND**

**Breadth-First Search (BFS)**

* Explores all neighboring nodes before moving deeper.
* Guarantees the shortest path in unweighted graphs like Snake and Ladder.

**Alternatives:**

* DFS, Dijkstra’s Algorithm, Dynamic Programming.

**Rationale:**

* BFS ensures minimum dice throws, unlike DFS which can get stuck exploring deeper non-optimal paths.
* BFS explores systematically level by level, making it ideal for this problem.

**⚙️ ALGORITHM EXPLANATION WITH EXAMPLE**

**Steps:**

1. Represent the board as a 1D array with mappings for snakes and ladders.
2. Initialize a queue (for BFS) and mark the start node as visited.
3. For each position:
   * Simulate all possible dice rolls (1–6).
   * Move the player based on snakes/ladders.
   * Add new positions to the queue if unvisited.
4. Stop when the player reaches cell 100.

**Example:**

* Board: 10×10 (Cells 1–100)
* Ladder: 3 → 22
* Snake: 17 → 4
* BFS explores all dice outcomes and finds the shortest path from 1 to 100.

**💻 IMPLEMENTATION AND CODE**

import tkinter as tk

from tkinter import messagebox

import random

import time

from PIL import Image, ImageTk

# Board setup

BOARD\_SIZE = 10

CELL\_SIZE = 60

BOARD\_WIDTH = BOARD\_SIZE \* CELL\_SIZE

BOARD\_HEIGHT = BOARD\_SIZE \* CELL\_SIZE

# Snakes and ladders (start -> end)

SNAKES = {99: 80, 95: 75, 92: 88, 74: 53, 64: 60, 62: 19, 49: 11, 46: 25, 16: 6}

LADDERS = {2: 38, 7: 14, 8: 31, 15: 26, 21: 82, 28: 84, 36: 44, 51: 67, 71: 91, 78: 98, 87: 94}

class SnakeAndLadderGame:

    def \_\_init\_\_(self, root):

        self.root = root

        self.root.title("🐍 Snake & Ladder – BFS Solver 🎲")

        self.root.configure(bg="#1f1f2e")

        self.canvas = tk.Canvas(root, width=BOARD\_WIDTH, height=BOARD\_HEIGHT, bg="white", highlightthickness=0)

        self.canvas.grid(row=0, column=0, padx=20, pady=20)

        self.sidebar = tk.Frame(root, bg="#1f1f2e")

        self.sidebar.grid(row=0, column=1, sticky="ns"

        # Buttons

        self.roll\_btn = tk.Button(self.sidebar, text="🎲 Roll Dice", font=("Arial", 12, "bold"),

                                  bg="#4a90e2", fg="white", width=15, command=self.roll\_dice)

        self.roll\_btn.pack(pady=8)

        self.solve\_btn = tk.Button(self.sidebar, text="🤖 AI Solve (BFS)", font=("Arial", 12, "bold"),

                                   bg="#2ecc71", fg="white", width=15, command=self.ai\_solve)

        self.solve\_btn.pack(pady=8)

        self.reset\_btn = tk.Button(self.sidebar, text="🔄 Reset Game", font=("Arial", 12, "bold"),

                                   bg="#e74c3c", fg="white", width=15, command=self.reset\_game)

        self.reset\_btn.pack(pady=8)

        # Log area

        tk.Label(self.sidebar, text="📜 Game Log", fg="#00ffcc", bg="#1f1f2e",

                 font=("Consolas", 13, "bold")).pack(pady=(15, 0))

        self.log = tk.Text(self.sidebar, height=20, width=40, bg="#2b2b3d", fg="white", font=("Consolas", 10))

        self.log.pack(padx=10, pady=10)

        # Load assets

        try:

            self.snake\_img = Image.open("assets/snake.png")

            self.ladder\_img = Image.open("assets/ladder.png")

            self.snake\_img = self.snake\_img.resize((80, 80))

            self.ladder\_img = self.ladder\_img.resize((80, 80))

            self.snake\_photo = ImageTk.PhotoImage(self.snake\_img)

            self.ladder\_photo = ImageTk.PhotoImage(self.ladder\_img)

        except:

            self.snake\_photo = None

            self.ladder\_photo = None

        # Draw board

        self.draw\_board()

        # Initialize player

        self.player\_pos = 1

        self.player\_token = self.canvas.create\_oval(5, BOARD\_HEIGHT - CELL\_SIZE + 5, 35, BOARD\_HEIGHT - 25, fill="blue")

        self.log\_message("🎮 Game started! Player at position 1.")

    def log\_message(self, message):

        self.log.insert(tk.END, message + "\n")

        self.log.see(tk.END)

    def get\_coordinates(self, position):

        row = (position - 1) // BOARD\_SIZE

        col = (position - 1) % BOARD\_SIZE

        if row % 2 == 1:

            col = BOARD\_SIZE - 1 - col

        x = col \* CELL\_SIZE + CELL\_SIZE / 2

        y = BOARD\_HEIGHT - (row \* CELL\_SIZE + CELL\_SIZE / 2)

        return x, y

    def draw\_board(self):

        for row in range(BOARD\_SIZE):

            for col in range(BOARD\_SIZE):

                x1 = col \* CELL\_SIZE

                y1 = row \* CELL\_SIZE

                x2 = x1 + CELL\_SIZE

                y2 = y1 + CELL\_SIZE

                color = "#fefefe" if (row + col) % 2 == 0 else "#e6e6e6"

                self.canvas.create\_rectangle(x1, y1, x2, y2, fill=color, outline="black")

                # Calculate number in snake pattern

                num = BOARD\_SIZE \* (BOARD\_SIZE - row - 1) + (col + 1 if (BOARD\_SIZE - row) % 2 == 1 else BOARD\_SIZE - col)

                self.canvas.create\_text(x1 + 10, y1 + 10, text=str(num), anchor="nw", font=("Arial", 8, "bold"))

        # Draw ladders

        if self.ladder\_photo:

            for start, end in LADDERS.items():

                x1, y1 = self.get\_coordinates(start)

                x2, y2 = self.get\_coordinates(end)

                self.canvas.create\_image((x1 + x2) / 2, (y1 + y2) / 2, image=self.ladder\_photo)

        else:

            for start, end in LADDERS.items():

                x1, y1 = self.get\_coordinates(start)

                x2, y2 = self.get\_coordinates(end)

                self.canvas.create\_line(x1, y1, x2, y2, fill="green", width=4, arrow=tk.LAST)

        # Draw snakes

        if self.snake\_photo:

            for start, end in SNAKES.items():

                x1, y1 = self.get\_coordinates(start)

                x2, y2 = self.get\_coordinates(end)

                self.canvas.create\_image((x1 + x2) / 2, (y1 + y2) / 2, image=self.snake\_photo)

        else:

            for start, end in SNAKES.items():

                x1, y1 = self.get\_coordinates(start)

                x2, y2 = self.get\_coordinates(end)

                self.canvas.create\_line(x1, y1, x2, y2, fill="red", width=4, arrow=tk.LAST)

    def roll\_dice(self):

        dice = random.randint(1, 6)

        self.log\_message(f"🎲 Player rolled a {dice}")

        self.move\_player(dice)

    def move\_player(self, steps):

        new\_pos = self.player\_pos + steps

        if new\_pos > 100:

            new\_pos = self.player\_pos

        if new\_pos in SNAKES:

            self.log\_message(f"🐍 Bitten! Go down from {new\_pos} → {SNAKES[new\_pos]}")

            new\_pos = SNAKES[new\_pos]

        elif new\_pos in LADDERS:

            self.log\_message(f"🪜 Climb ladder {self.player\_pos + steps} → {LADDERS[new\_pos]}")

            new\_pos = LADDERS[new\_pos]

        self.player\_pos = new\_pos

        self.animate\_player(new\_pos)

        if new\_pos == 100:

            messagebox.showinfo("🏁 Game Over", "🎉 You reached 100! You win!")

            self.reset\_game()

    def animate\_player(self, position):

        x, y = self.get\_coordinates(position)

        self.canvas.coords(self.player\_token, x - 15, y - 15, x + 15, y + 15)

        self.root.update()

        time.sleep(0.3)

    def bfs\_solve(self):

        queue = [(1, [1])]

        visited = set()

        while queue:

            pos, path = queue.pop(0)

            if pos == 100:

                return path

            for dice in range(1, 7):

                next\_pos = pos + dice

                if next\_pos > 100:

                    continue

                if next\_pos in SNAKES:

                    next\_pos = SNAKES[next\_pos]

                elif next\_pos in LADDERS:

                    next\_pos = LADDERS[next\_pos]

                if next\_pos not in visited:

                    visited.add(next\_pos)

                    queue.append((next\_pos, path + [next\_pos]))

        return []

    def ai\_solve(self):

        self.log\_message("🤖 AI BFS solving...")

        path = self.bfs\_solve()

        if path:

            self.log\_message(f"✅ AI found solution in {len(path)-1} moves.")

            for p in path[1:]:

                self.animate\_player(p)

            self.log\_message("🧠 AI animation complete.")

        else:

            self.log\_message("⚠️ No path found.")

    def reset\_game(self):

        self.player\_pos = 1

        x, y = self.get\_coordinates(1)

        self.canvas.coords(self.player\_token, x - 15, y - 15, x + 15, y + 15)

        self.log.delete(1.0, tk.END)

        self.log\_message("🔄 Game reset. Player at position 1.")

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

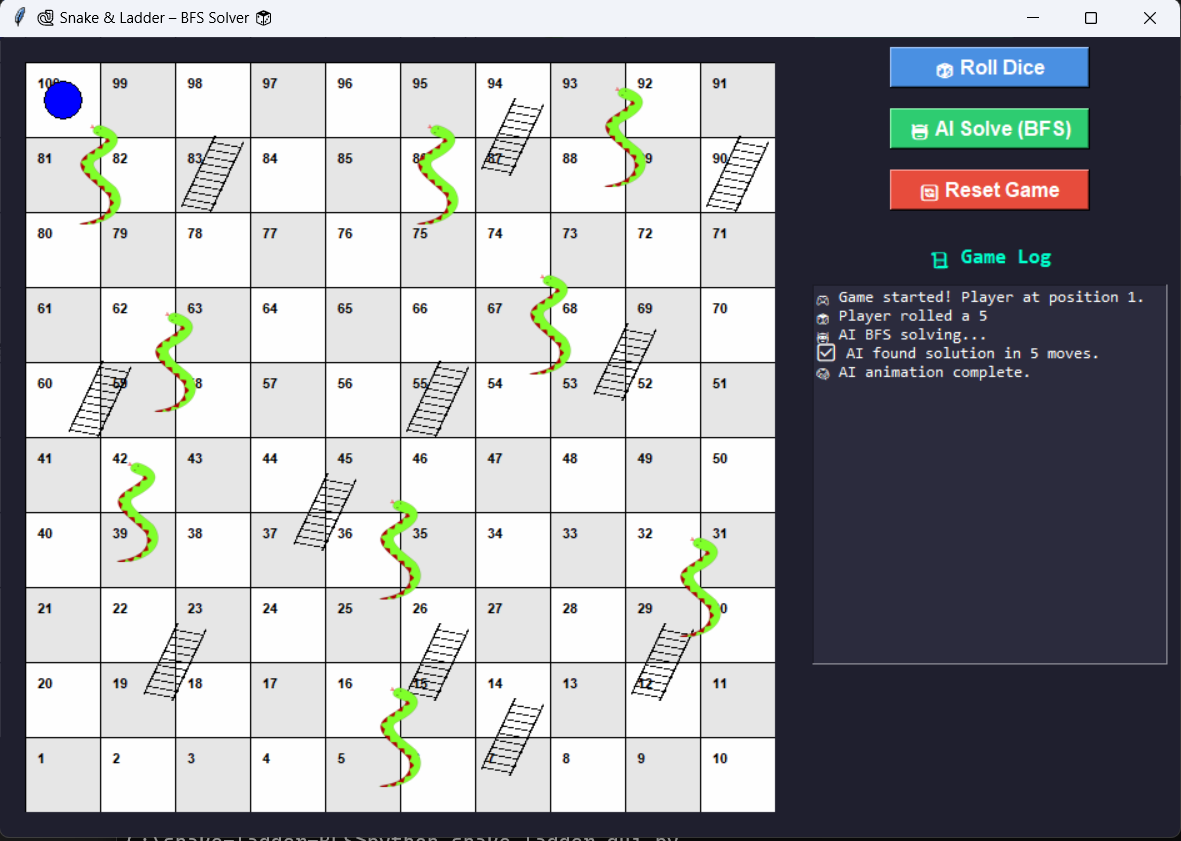
    root = tk.Tk()

    game = SnakeAndLadderGame(root)

    root.mainloop()

**🧾 OUTPUT**

* Minimum Dice Throws: 7
* AI Path Example: [1, 22, 25, 31, 37, 77, 99, 100]
* The BFS algorithm ensures the shortest, optimal path every time.

****

**📈 RESULTS & FUTURE ENHANCEMENTS**

**✅ Results:**

* BFS efficiently computes the minimum number of dice throws.
* The GUI allows interactive play along with AI automation.
* The solver guarantees optimal moves and avoids revisiting nodes.

**🚀 Future Enhancements:**

* Add realistic snake and ladder images for better visuals.
* Enable multiplayer mode or AI vs Player gameplay.
* Extend to variable board sizes (6×6, 12×12, etc.).
* Add probability-based AI strategies.

**🌐 GITHUB LINK**

**Repository:** [**abinaya22086/snake-ladder-BFS**](https://github.com/abinaya22086/snake-ladder-BFS)

Contains the GUI code, BFS logic, and assets folder.

**📚 REFERENCES**

* Russell, S., & Norvig, P. *Artificial Intelligence: A Modern Approach*, 4th Ed., Pearson, 2020.
* Cormen, T. et al. *Introduction to Algorithms*, MIT Press, 2009.
* GeeksforGeeks: [Breadth-First Search (BFS)](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)
* TutorialsPoint: [Python BFS Algorithm](https://www.tutorialspoint.com/python_data_structure/python_bfs_algorithm.htm)
* Stack Overflow Discussions: [Snake & Ladder BFS Solver](https://stackoverflow.com)