

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

**DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | 2117240030044 |
| **NAME** | HARSHAVARDHAN |
| **PROJECT TITLE** | Parking Slot Finder Application Using BFS Algorithm |
| **DATE OF SUBMISSION** |  |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**1. INTRODUCTION**  
Artificial Intelligence (AI) enables machines to perform tasks that require human-like reasoning, learning, and decision-making. In smart city applications, AI is widely used for traffic control, parking management, and route optimization.

This project implements a **Parking Slot Finder Application** using the **Breadth-First Search (BFS)** algorithm. The aim is to simulate an AI agent that explores a parking lot grid and finds the **nearest available parking slot** from the entry point marked as 'I'.

**2. PROBLEM STATEMENT**  
This project models a parking lot as a 2D grid, where each cell represents a parking space. The AI agent must navigate from the **entry point 'I'** to the **nearest empty slot 'E'** using BFS.

**Expected Results:**

* BFS identifies the shortest and most efficient path to an empty slot.
* The system marks the slot as occupied once parked.

**Future Possibilities:**

* Add multiple entry points and dynamic slot updates.
* Extend with A\* Search or integrate a graphical interface using Tkinter or Pygame.
* Include real-time sensors and IoT integration for live parking data.

**3. THEORETICAL BACKGROUND**  
**Depth-First Search (DFS):**

* Explores deep paths first.
* May not find the shortest path in a grid.

**Breadth-First Search (BFS):**

* Explores level by level.
* Guarantees the shortest path in unweighted grids.

*Dijkstra / A Search:*\*

* Designed for weighted graphs.
* Not required for this project since all moves are equal.

**Justification for Choosing BFS:**

* The parking lot is modeled as an **unweighted grid**.
* BFS ensures the **shortest path** to the nearest empty slot.
* DFS may explore unnecessary paths or miss optimal solutions.
* Weighted algorithms are unnecessary for equal-cost movements.

**4. ALGORITHM EXPLANATION**  
**Breadth-First Search (BFS)** is a grid-based traversal algorithm that explores cells in layers.

**Steps:**

1. Start at the entry point 'I'.
2. Add it to a queue and mark it as visited.
3. While the queue is not empty:
   * Remove the first cell.
   * If it is an empty slot 'E', stop and mark it as parked 'X'.
   * Otherwise, add all unvisited neighboring cells to the queue.
4. Repeat until an empty slot is found or all reachable cells are explored.

**5. BFS EXAMPLE GRID**

Initial Grid:

I X X E

X X E X

X E X X

X X X E

**Execution Steps:**

* Queue: [(0, 0)] → Explore right and down
* Queue: [(0, 3)] → Found empty slot
* Mark (0, 3) as parked 'X'

**Shortest Path Found:**

Start → (0, 3) → Parked

Updated Grid:

I X X X

X X E X

X E X X

X X X E

**6. QUEUE ACCESS ORDER IN BFS**

In the Parking Slot Finder Application, the **Breadth-First Search (BFS)** algorithm uses a **queue** to explore the parking lot grid level by level. The queue ensures that the nearest positions are checked first, guaranteeing the shortest path to an empty slot.

**Step-by-Step Access Order:**

1. **Initialize the Queue**
   * Add the starting position 'I' to the queue.
   * Example: Queue = [(0, 0)]
2. **Explore the First Position**
   * Remove the first position from the queue using popleft().
   * Check if it is an empty slot 'E'.
   * If yes, mark it as parked 'X' and stop.
3. **Add Valid Neighbors**
   * If the current cell is not 'E', check its neighbors:
     + Up (r-1, c)
     + Down (r+1, c)
     + Left (r, c-1)
     + Right (r, c+1)
   * For each neighbor:
     + If it is not blocked 'X' and not visited, add it to the queue.
     + Example: Queue = [(0, 3), (1, 0)]
4. **Repeat Until Found**
   * Continue removing and exploring positions from the queue.
   * BFS always explores the **oldest added position first**, ensuring the shortest path.
5. **Mark the Slot**
   * Once an empty slot 'E' is found, change it to 'X' to show it is now parked.

IMPLEMENTATION AND CODE:

from collections import deque

# Step 1: Create a 10x10 parking lot grid with open paths

def create\_parking\_lot():

    lot = [

        ['I', 'O', 'O', 'E', 'X', 'X', 'X', 'X', 'X', 'X'],

        ['X', 'O', 'X', 'O', 'O', 'X', 'X', 'X', 'X', 'X'],

        ['X', 'O', 'X', 'X', 'O', 'O', 'O', 'X', 'X', 'X'],

        ['X', 'O', 'O', 'O', 'E', 'X', 'O', 'X', 'X', 'X'],

        ['X', 'X', 'X', 'X', 'O', 'O', 'O', 'X', 'X', 'X'],

        ['X', 'X', 'X', 'X', 'X', 'X', 'O', 'O', 'O', 'E'],

        ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'O', 'X'],

        ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'O', 'X'],

        ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'O', 'X'],

        ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'O', 'E']

    ]

    return lot, (0, 0)

# Step 2: Print the parking lot grid

def print\_grid(lot, title):

    print(f"\n🅿️ {title}:")

    for row in lot:

        print(" ".join(row))

    print()

# Step 3: BFS to find and mark the nearest empty slot

def find\_and\_park(lot, start):

    rows, cols = len(lot), len(lot[0])

    visited = [[False]\*cols for \_ in range(rows)]

    queue = deque([start])

    visited[start[0]][start[1]] = True

    directions = [(-1,0), (1,0), (0,-1), (0,1)]  # Up, Down, Left, Right

    while queue:

        r, c = queue.popleft()

        # Check if current cell is an empty slot

        if lot[r][c] == 'E':

            lot[r][c] = 'X'  # Mark as occupied

            print(f"✅ Nearest free parking slot found at: ({r}, {c})")

            print("🚗 Vehicle parked successfully!")

            return True

        # Explore neighbors

        for dr, dc in directions:

            nr, nc = r + dr, c + dc

            if 0 <= nr < rows and 0 <= nc < cols:

                if not visited[nr][nc] and lot[nr][nc] in ('O', 'E'):

                    visited[nr][nc] = True

                    queue.append((nr, nc))

    print("❌ No empty parking slot available.")

    return False

# Step 4: Remove (unpark) a vehicle

def remove\_car(lot):

    print\_grid(lot, "Current Parking Lot")

    try:

        r = int(input("Enter the row number of the car to remove (0–9): "))

        c = int(input("Enter the column number of the car to remove (0–9): "))

        if lot[r][c] == 'X':

            lot[r][c] = 'E'

            print(f"🚙 Car removed successfully from location ({r}, {c}).")

        else:

            print("❌ That spot doesn’t have a car parked.")

    except (ValueError, IndexError):

        print("⚠️ Invalid input! Please enter row and column between 0–9.")

# Step 5: Menu system

def main\_menu():

    lot, start\_pos = create\_parking\_lot()

    while True:

        print("\n=== 🚘 PARKING SLOT FINDER ===")

        print("1. Find Nearest Parking Slot")

        print("2. Remove a Parked Car")

        print("3. Exit")

        choice = input("Enter your choice (1–3): ").strip()

        if choice == '1':

            print\_grid(lot, "Current Parking Lot")

            find\_and\_park(lot, start\_pos)

            print\_grid(lot, "Updated Parking Lot After Parking")

        elif choice == '2':

            remove\_car(lot)

            print\_grid(lot, "Updated Parking Lot After Car Removal")

        elif choice == '3':

            print("👋 Exiting the program. Have a nice day!")

            break

        else:

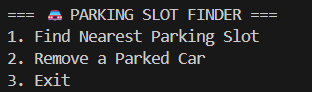
            print("❌ Invalid choice. Please enter 1, 2, or 3.")

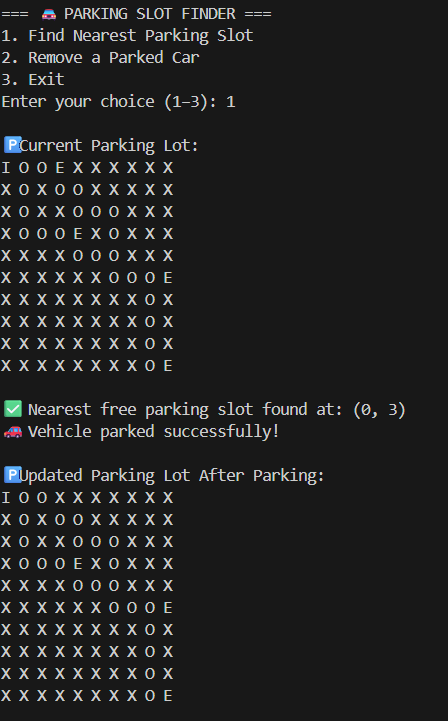
# Run the program

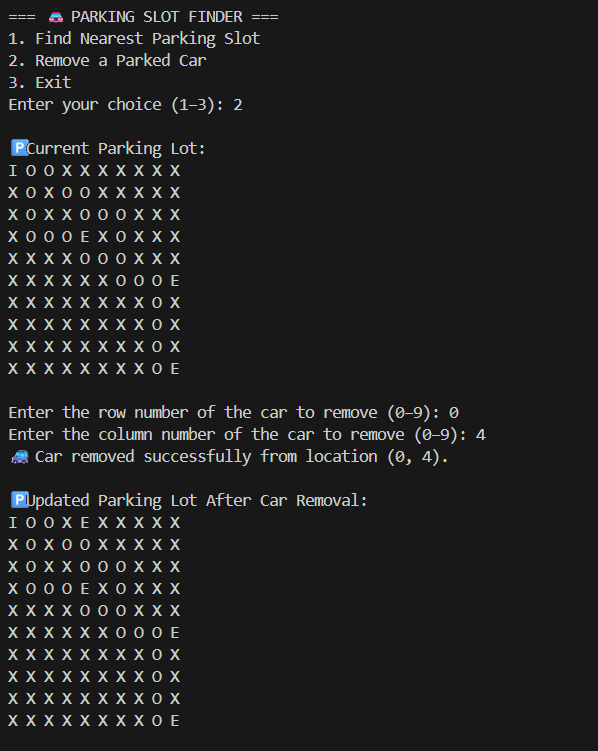
if \_\_name\_\_ == "\_\_main\_\_":

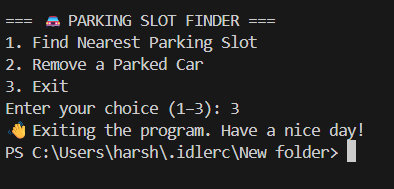
    main\_menu()

OUTPUT:









**Explanation:**

* The project simulates a **smart parking management system** that helps find the **nearest available parking slot** from an entrance point using the **Breadth-First Search (BFS)** algorithm.
* The parking area is represented as a **10×10 grid**, where:
  + 'I' denotes the **entry point**,
  + 'E' denotes an **empty parking slot**,
  + 'X' denotes an **occupied or blocked area**, and
  + 'O' denotes an **open path**.
* The BFS algorithm explores all reachable cells **level by level** from the entry point to locate the **nearest free slot**.
* Once found, the slot is marked as **occupied**, simulating that a vehicle has been parked successfully.
* A **car removal feature** allows users to free an occupied slot and make it available again.
* The system provides a **menu-based interface**, enabling users to:
  + Find the nearest parking slot.
  + Remove a parked car.
  + Exit the program.
* BFS ensures an **efficient and shortest route** to available slots while avoiding blocked cells.

**RESULTS AND FUTURE ENHANCEMENTS**

**Result**

* BFS successfully identifies the **nearest available parking slot** in the grid.
* The algorithm handles **multiple branches** and ensures that no cell is revisited.
* The **menu-driven interface** allows users to park and unpark vehicles easily.
* The system simulates **real-time parking management**, showing grid updates after every operation.
* It demonstrates the efficiency and real-world applicability of BFS in **resource allocation and navigation systems**.

**Future Enhancements**

**1. Multiple Vehicle Management**

* Extend the system to manage **multiple vehicles simultaneously**, each tracked with unique IDs.
* Add a **queue system** to handle parking requests when the lot is full.

**2. Smart Slot Reservation**

* Introduce a **reservation feature** that allows users to pre-book parking slots.
* Use **priority queues** to assign slots based on vehicle type or user status.

**3. Obstacles and Dynamic Movement**

* Add **temporary obstacles** like maintenance zones or reserved spaces.
* Modify BFS to dynamically recalculate paths if obstacles appear.

**4. Weighted Pathfinding**

* Assign weights to certain cells (like narrow lanes or high-traffic areas).
* Upgrade the algorithm from BFS to **Dijkstra’s** or **A**\* to find optimal paths based on weights.

**5. Real-Time Simulation and GUI**

* Implement a **graphical interface** using **Tkinter** or **Pygame**.
* Visually display the parking grid and animate the car’s movement to the allocated slot.

**6. Database Integration**

* Connect to a **MySQL or SQLite database** to store and retrieve slot information persistently.
* Keep track of total slots, parked vehicles, and available spaces across sessions.

**GitHub Link of the Project and Report**

🔗 https://github.com/HARSHA2006ind/AIMINIPROJECT

**REFERENCES**

* **GeeksforGeeks** — Breadth First Search (BFS) Algorithm in Python  
  🔗 <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
* **Python Software Foundation** — Python 3 Official Documentation  
  🔗 <https://docs.python.org/3/>
* **Real Python** — Python BFS and Queue Implementations  
  🔗 <https://realpython.com/>
* **TutorialsPoint** — Python Functions and Control Structures  
  🔗 <https://www.tutorialspoint.com/python/>
* **Medium (Towards Data Science)** — Understanding Breadth-First Search in Graph Theory  
  🔗 <https://towardsdatascience.com/>