21CSC206T – ARTIFICIAL INTELLIGENCE

LAB BASED ASSSIGNMENT – FT2

By: Abishake Priyan K

Reg-No.: RA2311004010058

1. Implementation of a Toy Problem and justify how it can be used for real world problems:

The a_star() function implements the A* pathfinding algorithm to find the shortest route on a 5x5 grid. It starts by initializing a priority queue to explore nodes based on their cost. It uses two dictionaries: g_score to track the cost from the start to each node, and f_score to estimate the total cost to the goal.

The function repeatedly examines the node with the lowest f_score, checking its neighbors. For each neighbor, it calculates a tentative cost and updates the path if this cost is lower than previously recorded.

Once the goal is reached, the function reconstructs the path by backtracking through the came_from dictionary. If no path is found, it returns an empty list. The algorithm efficiently navigates around obstacles, ensuring the robot reaches its destination in the shortest possible way.

PROGRAM:

```
import tkinter as tk

# Grid size
GRID_SIZE = 5
CELL_SIZE = 100
START = (0, 0)
GOAL = (GRID_SIZE - 1, GRID_SIZE - 1)

# Obstacles (more added for complexity)
OBSTACLES = {
    (0, 4),
    (1, 1), (1, 2),
    (2, 3),
    (3, 0), (3, 1), (3, 3),
    (4, 0), (4, 1)
}

class GridWorld:
```

```
    def __init__(self, master):
        self.master = master
        self.canvas = tk.Canvas(master, width=GRID_SIZE * CELL_SIZE,
height=GRID_SIZE * CELL_SIZE, bg="white")
        self.canvas.pack()

        self.text_box = tk.Text(master, width=50, height=10)
        self.text_box.pack(pady=10)
        self.text_box.insert(tk.END, "Step 1: Initial state\n")
        self.text_box.config(state=tk.DISABLED)

        self.robot = START  # Initial robot position
        self.robot_icon = None
        self.goal_reached = False  # Flag to check if the goal is reached
        self.draw_grid()
        self.draw_obstacles()
        self.draw_initial_and_goal()
        self.draw_robot()  # Ensure robot is visible initially

        # Bind the W, A, S, D keys to control the robot
        self.master.bind("<w>", self.move_up)
        self.master.bind("<a>", self.move_left)
        self.master.bind("<s>", self.move_down)
        self.master.bind("<d>", self.move_right)

        # Restart Button
        self.restart_button = tk.Button(master, text="Restart",
command=self.restart, state=tk.DISABLED)
        self.restart_button.pack(pady=10)

    def draw_grid(self):
        """Draw the grid lines."""
        for i in range(GRID_SIZE + 1):
            self.canvas.create_line(i * CELL_SIZE, 0, i * CELL_SIZE,
GRID_SIZE * CELL_SIZE, fill="gray")
            self.canvas.create_line(0, i * CELL_SIZE, GRID_SIZE *
CELL_SIZE, i * CELL_SIZE, fill="gray")

    def draw_obstacles(self):
        """Draw obstacles as rock emojis."""
        for (x, y) in OBSTACLES:
            self.canvas.create_text(
                x * CELL_SIZE + 50, y * CELL_SIZE + 50,
                text="🪨", font=("Helvetica", 40)
            )

    def draw_initial_and_goal(self):
        """Label the initial and goal state."""
        self.canvas.create_text(
            0 * CELL_SIZE + 50, 0 * CELL_SIZE + 50,
            text="Initial State", font=("Helvetica", 10, "bold"),
fill="blue"
        )
        self.canvas.create_text(
```

```
            (GRID_SIZE - 1) * CELL_SIZE + 50, (GRID_SIZE - 1) * CELL_SIZE
+ 50,
            text="Goal State", font=("Helvetica", 10, "bold"),
fill="green"
        )

    def draw_robot(self):
        """Draw the robot as a red circle."""
        x, y = self.robot
        self.robot_icon = self.canvas.create_oval(
            x * CELL_SIZE + 25, y * CELL_SIZE + 25,  # coordinates for the
red circle
            x * CELL_SIZE + 75, y * CELL_SIZE + 75,  # size of the circle
            fill="red", outline="red"
        )

    def display_message(self, message):
        """Display the message in the Tkinter text box and auto-scroll to
the bottom."""
        self.text_box.config(state=tk.NORMAL)
        self.text_box.insert(tk.END, message + "\n")
        self.text_box.config(state=tk.DISABLED)
        self.text_box.yview(tk.END)  # Auto-scroll to the bottom

    def move_up(self, event=None):
        """Move the robot up (decrease row)."""
        if not self.goal_reached:
            self.move_robot(0, -1)

    def move_left(self, event=None):
        """Move the robot left (decrease column)."""
        if not self.goal_reached:
            self.move_robot(-1, 0)

    def move_down(self, event=None):
        """Move the robot down (increase row)."""
        if not self.goal_reached:
            self.move_robot(0, 1)

    def move_right(self, event=None):
        """Move the robot right (increase column)."""
        if not self.goal_reached:
            self.move_robot(1, 0)

    def move_robot(self, dx, dy):
        """Move the robot and handle obstacle detection."""
        new_position = (self.robot[0] + dx, self.robot[1] + dy)

        # Check if the new position is within the grid boundaries
        if 0 <= new_position[0] < GRID_SIZE and 0 <= new_position[1] <
GRID_SIZE:
            # Check if there is an obstacle in the new position
            if new_position in OBSTACLES:
```

```
                self.display_message(f"Obstacle detected at
{new_position}!")
            else:
                # Move robot to the new position
                self.robot = new_position
                self.canvas.coords(
                    self.robot_icon,
                    self.robot[0] * CELL_SIZE + 25, self.robot[1] *
CELL_SIZE + 25,
                    self.robot[0] * CELL_SIZE + 75, self.robot[1] *
CELL_SIZE + 75
                )
                self.display_message(f"Moved to {self.robot}")

                # Check if the robot has reached the goal
                if self.robot == GOAL:
                    self.display_message("Goal State Reached!")
                    self.goal_reached = True
                    self.restart_button.config(state=tk.NORMAL)  # Enable
restart button
        else:
            self.display_message("Out of bounds!")

    def restart(self):
        """Reset the robot to the initial state."""
        self.robot = START
        self.goal_reached = False
        self.canvas.coords(
            self.robot_icon,
            self.robot[0] * CELL_SIZE + 25, self.robot[1] * CELL_SIZE +
25,
            self.robot[0] * CELL_SIZE + 75, self.robot[1] * CELL_SIZE + 75
        )
        self.display_message("Step 1: Initial state")
        self.restart_button.config(state=tk.DISABLED)  # Disable restart
button

if __name__ == "__main__":
    root = tk.Tk()
    root.title("Robot Motion Control - 5x5 Grid with Obstacles")
    app = GridWorld(root)
    root.mainloop()
```

OUTPUT:

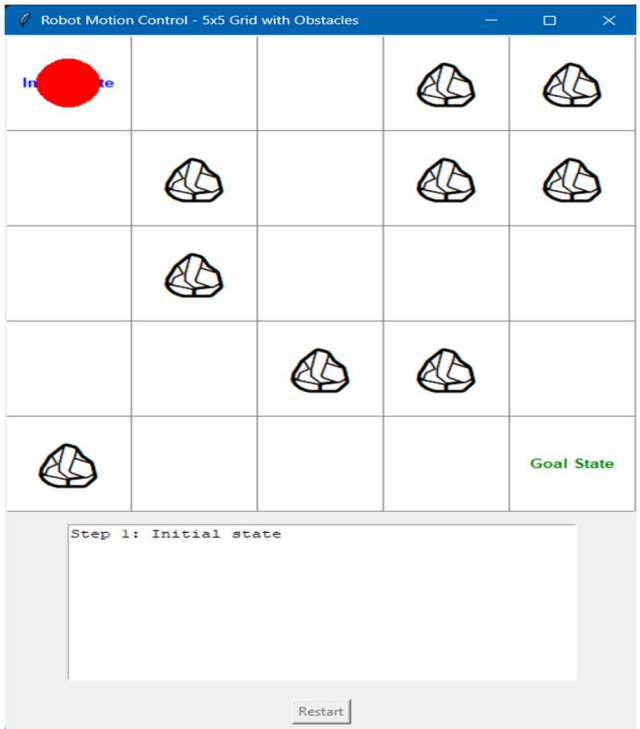

Fig – Robot in initial Position

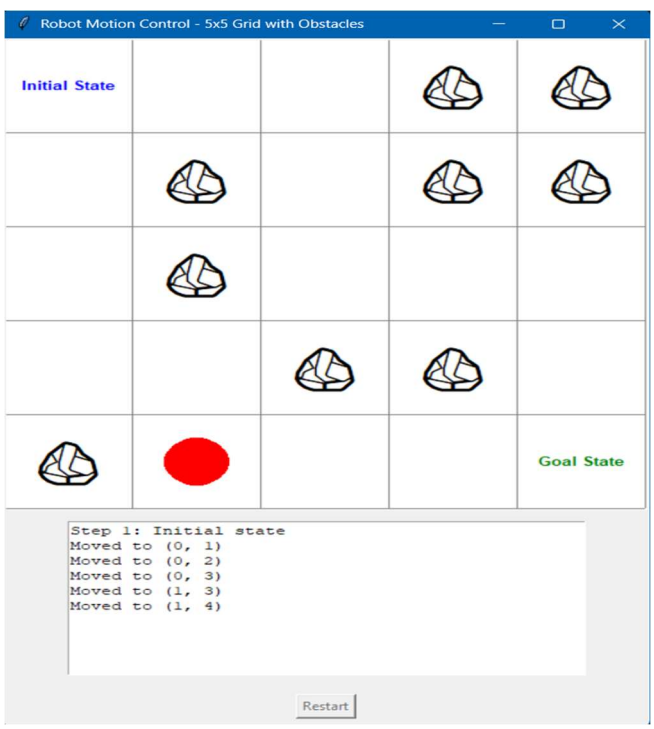

Fig – Robot Moves to reach Goal State

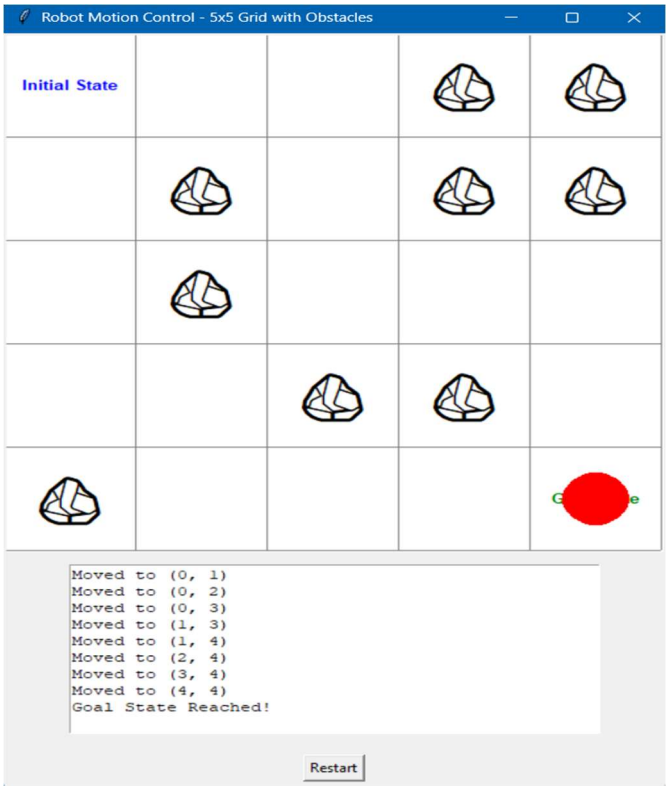

Fig - Goal State Reached

JUSTIFICATION FOR REAL WORLD PROBLEMS:

The **robot motion planning problem** using A* pathfinding is directly applicable to several real-world scenarios involving autonomous navigation, obstacle avoidance, and optimized movement. Here's how it translates into practical applications:

---

1. Warehouse Robotics

- **How it Relates**: Automated warehouse robots navigate large storage areas while avoiding obstacles (shelves, humans, or other robots).
- **Application**: A* helps find the **shortest path** for robots to pick up and deliver items efficiently.
- **Benefit**: Reduces delivery time and optimizes warehouse operations.

---

2. Autonomous Vehicles (Self-Driving Cars)

- **How it Relates**: Self-driving cars need to plan the best routes while avoiding obstacles and traffic.
- **Application**: A* is used for **real-time pathfinding** to determine safe and efficient routes.
- **Benefit**: Improves **safety, navigation efficiency, and fuel economy**.

---

3. Drone Navigation

- **How it Relates**: Drones must navigate through buildings or outdoor environments while avoiding obstacles like trees or power lines.
- **Application**: A* is used to compute the **optimal flight path** with minimal energy consumption.
- **Benefit**: Enables **automated deliveries (e.g., Amazon Prime Air)** and **search-and-rescue missions**.

---

4. Mars Rovers (e.g., NASA's Perseverance Rover)

- **How it Relates**: Mars rovers explore unknown terrain, avoiding rocks and cliffs while reaching target locations.
- **Application**: A* is used to plan **collision-free paths** with minimal energy use.
- **Benefit**: Ensures **efficient exploration** of planetary surfaces.

---

5. Game AI (Pathfinding in Video Games)

- **How it Relates**: AI-controlled characters in games navigate complex environments avoiding obstacles.
- **Application**: A* helps game NPCs find **optimal paths** to chase players or patrol areas.

- **Benefit**: Creates **realistic and intelligent AI behavior**.

2. Implementation of Search Algorithms for Real world Problems:

Program:

```
import tkinter as tk
import heapq

# Grid size
GRID_SIZE = 5
CELL_SIZE = 100
START = (0, 0)
GOAL = (GRID_SIZE - 1, GRID_SIZE - 1)

# Obstacles (more added for complexity)
OBSTACLES = {
    (0, 4),
    (1, 1), (1, 2),
    (2, 3),
    (3, 0), (3, 1), (3, 3),
    (4, 0), (4, 1)
}

class GridWorld:
    def __init__(self, master):
        self.master = master
        self.canvas = tk.Canvas(master, width=GRID_SIZE * CELL_SIZE,
height=GRID_SIZE * CELL_SIZE, bg="white")
        self.canvas.pack()

        self.text_box = tk.Text(master, width=50, height=10)
        self.text_box.pack(pady=10)
        self.text_box.insert(tk.END, "Step 1: Initial state\n")
        self.text_box.config(state=tk.DISABLED)

        self.robot = START  # Initial robot position
        self.path = None  # No path initially
        self.robot_icon = None
        self.start_button = tk.Button(master, text="Start",
command=self.start)
        self.start_button.pack(pady=10)

        self.draw_grid()
        self.draw_obstacles()
        self.draw_initial_and_goal()
        self.draw_robot()  # Ensure robot is visible initially

    def draw_grid(self):
        """Draw the grid lines."""
```

```
        for i in range(GRID_SIZE + 1):
            self.canvas.create_line(i * CELL_SIZE, 0, i * CELL_SIZE,
GRID_SIZE * CELL_SIZE, fill="gray")
            self.canvas.create_line(0, i * CELL_SIZE, GRID_SIZE *
CELL_SIZE, i * CELL_SIZE, fill="gray")

    def draw_obstacles(self):
        """Draw obstacles as rock emojis."""
        for (x, y) in OBSTACLES:
            self.canvas.create_text(
                x * CELL_SIZE + 50, y * CELL_SIZE + 50,
                text="🪨", font=("Helvetica", 40)
            )

    def draw_initial_and_goal(self):
        """Label the initial and goal state."""
        self.canvas.create_text(
            0 * CELL_SIZE + 50, 0 * CELL_SIZE + 50,
            text="Initial State", font=("Helvetica", 10, "bold"),
fill="blue"
        )
        self.canvas.create_text(
            (GRID_SIZE - 1) * CELL_SIZE + 50, (GRID_SIZE - 1) * CELL_SIZE
+ 50,
            text="Goal State", font=("Helvetica", 10, "bold"),
fill="green"
        )

    def draw_robot(self):
        """Draw the robot as a red circle."""
        x, y = self.robot
        self.robot_icon = self.canvas.create_oval(
            x * CELL_SIZE + 25, y * CELL_SIZE + 25,  # coordinates for the
red circle
            x * CELL_SIZE + 75, y * CELL_SIZE + 75,  # size of the circle
            fill="red", outline="red"
        )

    def move_robot(self):
        """Move the robot slowly along the computed path and display step
messages."""
        if self.path:
            current = self.robot
            next_step = self.path.pop(0)

            # Update the robot's position
            self.robot = next_step
            self.canvas.coords(
                self.robot_icon,
                self.robot[0] * CELL_SIZE + 25, self.robot[1] * CELL_SIZE
+ 25,
                self.robot[0] * CELL_SIZE + 75, self.robot[1] * CELL_SIZE
+ 75
            )
```

```
            # Message about the movement
            step_message = f"Moving from {current} to {self.robot}"

            # Check if there is an obstacle
            if self.robot in OBSTACLES:
                step_message += " (Obstacle detected, changing path!)"

            self.display_message(step_message)

            # Wait and continue moving robot in 500 ms
            self.master.after(1500, self.move_robot)
        else:
            self.display_message("Final Step: Goal state reached")

    def display_message(self, message):
        """Display the message in the Tkinter text box."""
        self.text_box.config(state=tk.NORMAL)
        self.text_box.insert(tk.END, message + "\n")
        self.text_box.config(state=tk.DISABLED)
        self.text_box.yview(tk.END)

    def heuristic(self, a, b):
        """Manhattan distance heuristic."""
        return abs(a[0] - b[0]) + abs(a[1] - b[1])

    def a_star(self, start, goal):
        """A* pathfinding algorithm."""
        open_list = []
        heapq.heappush(open_list, (0, start))
        came_from = {start: None}
        g_score = {start: 0}
        f_score = {start: self.heuristic(start, goal)}

        while open_list:
            _, current = heapq.heappop(open_list)

            if current == goal:
                path = []
                while current:
                    path.append(current)
                    current = came_from[current]
                return path[::-1]  # Return the correct path order

            neighbors = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down,
Left, Up
            for dx, dy in neighbors:
                neighbor = (current[0] + dx, current[1] + dy)

                if (0 <= neighbor[0] < GRID_SIZE and 0 <= neighbor[1] <
GRID_SIZE
                    and neighbor not in OBSTACLES):

                    temp_g_score = g_score[current] + 1
```

```
                    if neighbor not in g_score or temp_g_score <
g_score[neighbor]:
                        g_score[neighbor] = temp_g_score
                        f_score[neighbor] = temp_g_score +
self.heuristic(neighbor, goal)
                        heapq.heappush(open_list, (f_score[neighbor],
neighbor))
                        came_from[neighbor] = current

        return None  # No path found

    def start(self):
        """Start or restart the robot motion."""
        # Clear the text box and reset everything
        self.text_box.config(state=tk.NORMAL)
        self.text_box.delete(1.0, tk.END)
        self.text_box.insert(tk.END, "Step 1: Initial state\n")
        self.text_box.config(state=tk.DISABLED)

        # If the robot has already completed the path, reset everything
        if self.robot_icon is not None:
            self.canvas.delete(self.robot_icon)
            self.robot = START  # Reset robot to the start
            self.path = None
            self.draw_robot()  # Redraw robot at the initial position

        # Compute path and move the robot
        self.display_message("Step 2: Finding the shortest path")
        self.path = self.a_star(START, GOAL)  # Compute path
        if self.path:
            self.move_robot()  # Start moving the robot
        else:
            self.display_message("No path found!")

if __name__ == "__main__":
    root = tk.Tk()
    root.title("Robot Motion Planning (A* Algorithm) - 5x5 Grid")
    app = GridWorld(root)
    root.mainloop()
```

OUTPUT:

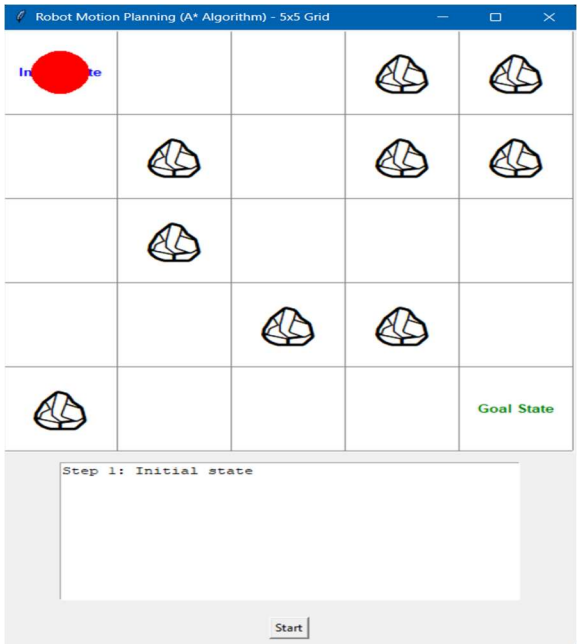

Fig – Robot Starts From Initial State

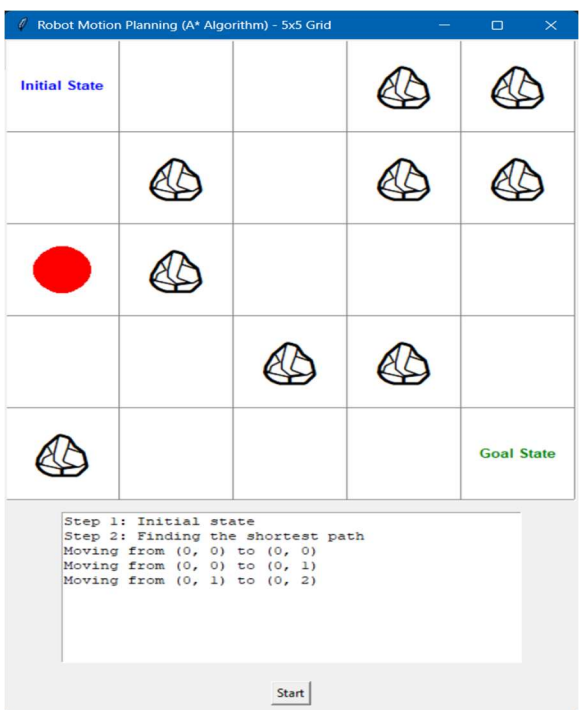

Fig – Robot Detects the Shortest Path
And Avoid Obstacles

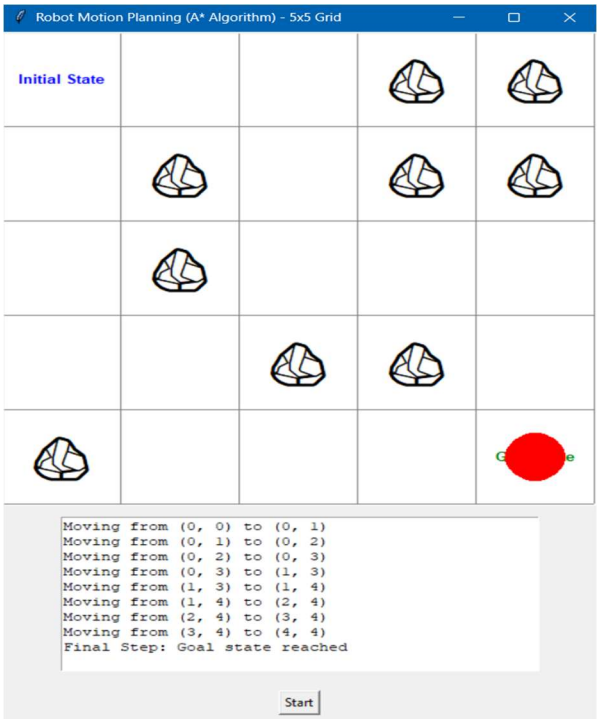

Fig – Reached Final Goal State
By Shortest Path

What is the A* Algorithm?

A* (A-Star) is a **graph-based pathfinding algorithm** used to find the shortest path between a start and a goal while considering both **cost efficiency** and **heuristics**. It is widely used in robotics, AI, and navigation systems because it efficiently finds optimal paths while avoiding obstacles.

How it Works

A* evaluates paths using the function:

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

Where:

- **g(n):** The actual cost from the start node to the current node.
- **h(n):** The estimated cost from the current node to the goal (heuristic).

- **f(n):** The total estimated cost of the path.

Algorithm Steps

1. **Initialize**:
    - Add the **start node** to a priority queue (open set).
    - Set g(start) = 0 and f(start) = heuristic(start, goal).
2. **Process Nodes**:
    - Extract the node with the lowest f(n).
    - If it's the **goal**, reconstruct and return the path.
    - Otherwise, expand to valid neighboring nodes.
3. **Update Costs**:
    - Compute g(new) = g(current) + movement_cost.
    - If g(new) < g(previous), update came_from and f(new).
    - Add neighbors to the open set if not already explored.
4. **Repeat Until Goal is Reached**

Applications of A Algorithm*

1. Robotics and Autonomous Navigation

- Used for **robot motion planning** in self-driving cars, warehouse robots, and drones.
- Helps **avoid obstacles** and **navigate efficiently**.

2. GPS Navigation Systems

- Used in **Google Maps, Apple Maps**, and vehicle GPS to compute the **fastest route**.
- Considers **road distance (g)** and **estimated travel time (h)**.

3. Video Games (AI Pathfinding)

- Used in **game NPCs** for efficient movement in large maps.
- Examples: Pathfinding in **Pac-Man, Age of Empires, Call of Duty**.

4. Network Routing

- Used in **computer networks** to find the shortest **data packet routes**.
- Helps optimize **internet traffic and network efficiency**.

5. Space Exploration

- Used by **Mars rovers (e.g., Perseverance)** to navigate **unknown terrain**.
- Ensures safe movement while avoiding obstacles.

Conclusion

The A* algorithm is a **versatile, efficient, and widely-used pathfinding technique**. It balances speed and accuracy, making it ideal for applications in robotics, AI, navigation, and beyond.