CS7050 Artificial Intelligence

Submitted to: Prof Vassil Vassilev
Submitted by
Muhammad Sajjad Hussain

Problem Solving using State Space Search *and Knowledge-based Inference*
December 2, 2024

London Metropolitan University ,
London

Contents

Maze Solving by State Space Search and Knowledge-based Inference

1 Why Greedy Best-First Search

When a heuristic is used to direct the search, the Greedy Best-First Search (GBFS) algorithm is the best choice. The Manhattan distance between the current node and the goal serves as the heuristic function in this job, which uses GBFS to discover the shortest way through a maze. Because it prioritises exploration based on the heuristic value and concentrates on examining the nodes that seem closest to the goal, this approach is favoured above other algorithms like Dijkstra.

2 Pros of Greedy Best-First Search

- **Heuristic-based Exploration**: The algorithm uses a heuristic to prioritize exploration which can lead to faster discovery of the goal.
- **Efficient Search in Unweighted Grids**: In maze navigation where the cost between neighboring nodes is the same, GBFS offers a good trade-off between efficiency and optimality.

3 Program Description

The program implements the Greedy Best-First Search algorithm to navigate through a maze and find the shortest path from the start node to the goal node. It visualizes the steps of the search process and tracks the optimal path and its cost.

3.1 Maze Setup

- The maze is represented as a 5x6 grid.
- Blocked cells are defined by the coordinates: (0, 1), (2, 1), (3, 1), (2, 3), (3, 4), and (4, 4).
- The start node is at (0, 0), and the goal node is at (4, 5).

3.2 Greedy Best-First Search Algorithm

- The algorithm uses a priority queue to explore nodes based on the Manhattan distance heuristic to the goal.
- The heuristic function calculates the Manhattan distance between two nodes.
- The greedy_best_first_search function uses this heuristic to guide the search, exploring neighboring nodes in the order of their heuristic values.
- If a nodes neighbors are valid (within bounds, not blocked, and not already visited) they are added to the priority queue for exploration.

3.3 Visualization

- The maze is visualized at each step, showing the explored nodes, start and goal positions, and the optimal path once it is found.
- Exploration process is animated with a short delay between steps to demonstrate how the algorithm progresses.

3.4 Running the Algorithm

- The start node (0, 0) is placed in the priority queue with a heuristic of 8 (Manhattan distance to the goal).
- Algorithm explores neighboring nodes, updating their heuristic values and predecessors.
- Each iteration the algorithm visualizes the explored nodes and path.
- Process continues until the goal is found or all paths are exhausted.
- Once the goal is reached, the algorithm traces the optimal path back to the start using the **prev** dictionary.

3.5 Optimal Path

Optimal path found by the algorithm is as follows:

$$[(0,0),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),(2,5),(3,5),(4,5)]$$

This path traverses through open cells, avoiding blocked nodes, and minimizes the Manhattan distance at each step.

3.6 Stage-by-Stage Results

- **Step 1:** Cost: 0; Path: $[(0,0)]$
- **Step 2:** Cost: 1; Path: $[(0,0),(1,0)]$
- **Step 3:** Cost: 2; Path: $[(0,0),(1,0),(1,1)]$
- **Step 4:** Cost: 3; Path: $[(0,0),(1,0),(1,1),(1,2)]$
- **Step 5:** Cost: 4; Path: $[(0,0),(1,0),(1,1),(1,2),(1,3)]$
- **Step 6:** Cost: 5; Path: $[(0,0),(1,0),(1,1),(1,2),(1,3),(1,4)]$
- **Step 7:** Cost: 6; Path: $[(0,0),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5)]$
- **Step 8:** Cost: 7; Path: $[(0,0),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),(2,5)]$
- **Step 9:** Cost: 8; Path: $[(0,0),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),(2,5),(3,5)]$
- **Step 10:** Cost: 9; Path: $[(0,0),(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),(2,5),(3,5),(4,5)]$

4 Heuristic Calculation

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

where (x_1, y_1) is the position of the current node and (x_2, y_2) is the goal position.

4.1 Initial State:

The starting node is $(0, 0)$, and the goal node is $(4, 5)$.Manhattan distance from $(0, 0)$ to $(4, 5)$ is:

$$h(0, 0) = |0 - 4| + |0 - 5| = 4 + 5 = 9$$

Algorithm begins with the start node in the priority queue.

4.1.1 Step 1 (Explore $(0, 0)$):

The current node is $(0, 0)$, and its heuristic value is 9 (Manhattan distance to the goal).
The possible neighbors of $(0, 0)$ are:

- $(1, 0)$: $h(1, 0) = |1 - 4| + |0 - 5| = 3 + 5 = 8$
- $(0, 1)$: Blocked
- $(0, -1)$: Out of bounds

Algorithm selects $(1, 0)$ as the next node to explore because it has the smallest heuristic value.

4.1.2 Step 2 (Explore $(1, 0)$):

The current node is $(1, 0)$, and its heuristic value is 8.
Possible neighbors of $(1, 0)$:

- $(2, 0)$: $h(2, 0) = 7$
- $(0, 0)$: Already visited
- $(1, 1)$: $h(1, 1) = 7$

Select $(1, 1)$ for exploration since it has the lowest heuristic value and is reachable.and this one comes first in queue so priority queue the algorithm select (1,1).if (2,0) comes first in the queue then the algorithm select this one and the optimal path will change.

4.1.3 Step 3 (Explore $(1, 1)$):

The current node is $(1, 1)$, and its heuristic value is 7.
Possible neighbors of $(1, 1)$:

- $(2, 1)$: Blocked
- $(1, 0)$: Already visited
- $(1, 2)$: $h(1, 2) = 6$
- $(0, 1)$: Blocked

Choose $(1, 2)$.

4.1.4 Step 4 (Explore $(1, 2)$):

The current node is $(1, 2)$, and its heuristic value is 6.
Possible neighbors of $(1, 2)$:

- $(2, 2)$: $h(2, 2) = 5$
- $(1, 3)$: $h(1, 3) = 5$

Choose $(1, 3)$ Because its comes first in Queue.

4.1.5 Step 5 (Explore $(1,3)$):

The current node is $(1,3)$, and its heuristic value is 5.
Possible neighbors of $(1,3)$:

- $(1,4): \quad h(1,4) = 4$
- $(1,2): \quad$ Already visited
- $(2,3): \quad$ Blocked
- $(0,3): \quad h(0,3) = 6$

4.1.6 Step 6 (Explore $(1,4)$):

The current node is $(1,4)$, and its heuristic value is 4.
Possible neighbors:

- $(1,5): \quad h(1,5) = 3$

We select $(1,5)$.

4.1.7 Step 7 (Explore $(1,5)$):

The current node is $(1,5)$, and its heuristic value is 3.
Possible neighbors:

- $(2,5): \quad h(2,5) = 2$

Move to $(2,5)$.

4.1.8 Step 8 (Explore $(2,5)$):

The current node is $(2,5)$, and its heuristic value is 2.
Possible neighbors:

- $(3,5): \quad h(3,5) = 1$

Move to $(3,5)$.

4.1.9 Step 9 (Explore $(3,5)$):

The current node is $(3,5)$, and its heuristic value is 1.
Possible neighbors:

- $(4,5): \quad$ Goal node

The algorithm is called **greedy** because it always chooses the **best** option available based on the current knowledge (the heuristic) without considering the broader search space.

```python
import matplotlib.pyplot as plt
import numpy as np
import heapq
import time

# Maze dimensions
rows, cols = 5, 6  # Maze has 5 rows and 6 columns

# Blocked nodes (cells that are obstacles) and open cells
blocked_nodes = [(0, 1), (2, 1), (3, 1), (2, 3), (3, 4), (4, 4)]  # Blocked
    cells
```

```
open_nodes = [
    (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (1, 1), (4, 1),
    (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (0, 3), (1, 3),
    (3, 3), (4, 3), (0, 4), (1, 4), (2, 4), (0, 5), (1, 5),
    (2, 5), (3, 5), (4, 5)
]

# Maze grid with zeros (open cells) and ones (blocked cells)
maze = np.zeros((rows, cols))  # Create a 5x6 grid initialized to 0 (open cells)
for r, c in blocked_nodes:  # Set blocked cells to -1
    maze[r][c] = -1

# start and goal positions
start = (0, 0)  # Start node is at (0, 0)
goal = (4, 5)  # Goal node is at (4, 5)

# Manhattan distance heuristic function
def heuristic(a, b):
    """Returns the Manhattan distance between two points."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

# Greedy Best-First Search algorithm
def greedy_best_first_search(maze, start, goal):
    """Performs Greedy Best-First Search on the maze to find the shortest path."""
    pq = [(heuristic(start, goal), start)]  # Priority queue initialized with start node which node goes firstfor exploration
    visited = set()  # Set to keep track of visited nodes
    prev = {start: None}  # Dictionary to store the predecessor of each node for path reconstruction for backtracking
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Directions for movement (up, down, left, right)
    explored = []  # List to store explored nodes for visualization

    while pq:
        _, current = heapq.heappop(pq)  # Get the node with the lowest heuristic value

        # If the goal is reached, reconstruct the path
        if current == goal:
            path = []  # List to store the path from start to goal
            while current:
                path.append(current)  # Add the current node to the path
                current = prev[current]  # Move to the predecessor of the current node
            return path[::-1], explored  # Reverse the path to start from the start node

        if current in visited:
            continue  # Skip nodes that have already been visited
        visited.add(current)  # Mark the current node as visited
        explored.append(current)  # Add the current node to the explored list

        # Explore the neighboring nodes
        for d in directions:
            neighbor = (current[0] + d[0], current[1] + d[1])  # Get the neighbor coordinates
            # Check if the neighbor is within bounds, not blocked, and not visited
            if (0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols
                    and maze[neighbor[0]][neighbor[1]] != -1
                    and neighbor not in visited):
```

```python
                prev[neighbor] = current  # Set the predecessor of the neighbor
                heapq.heappush(pq, (heuristic(neighbor, goal), neighbor))  # Add neighbor to priority queue

                # Visualize the maze after each step
                visualize_maze(maze, explored, start, goal)
                time.sleep(0.3)  # Pause for a short time to show the process
                plt.clf()  # Clear the figure for the next step

    return None, explored  # Return None if no path is found

# Visualization function for the maze
def visualize_maze(maze, explored, start, goal, path=None, show_path=True):
    """Visualizes the maze with explored nodes and the current path."""
    fig, ax = plt.subplots(figsize=(8, 6))

    # Custom color map: light blue for explored node and dark green for path of the maze an din the last  gray for blocked cells
    cmap = plt.cm.get_cmap("Blues", 2)  # Blues color map for explored and path
    cmap.set_under('gray')  # Blocked cells in gray
    ax.imshow(maze, cmap=cmap, origin='upper')

    # Annotate start and goal points
    ax.text(start[1], start[0], 'Start', color='green', ha='center', va='top', fontsize=15, fontweight='bold')
    ax.text(goal[1], goal[0], 'Goal', color='red', ha='center', va='top', fontsize=15, fontweight='bold')

    # Highlight the explored nodes with light blue
    for (r, c) in explored:
        ax.add_patch(plt.Rectangle((c - 0.5, r - 0.5), 1, 1, fill=True, color='lightblue', alpha=0.7))

    # Highlight the final path with dark green if the path is to be shown
    if show_path and path:
        for (r, c) in path:
            ax.add_patch(plt.Rectangle((c - 0.5, r - 0.5), 1, 1, fill=True, color='darkgreen', alpha=0.8))

    # Add gridlines for better readability
    ax.set_xticks(np.arange(-0.5, cols, 1), minor=True)
    ax.set_yticks(np.arange(-0.5, rows, 1), minor=True)
    ax.grid(which="minor", color="black", linestyle='-', linewidth=2)

    # Title and legend for further understanding
    ax.set_title("Greedy Best-First Search Maze Visualization")
    from matplotlib.lines import Line2D
    legend_elements = [
        Line2D([0], [0], color='green', lw=4, label='Start'),
        Line2D([0], [0], color='red', lw=4, label='Goal'),
        Line2D([0], [0], color='lightblue', lw=4, label='Explored Nodes'),
        Line2D([0], [0], color='darkgreen', lw=4, label='Path')
    ]
    ax.legend(handles=legend_elements, loc='upper left')

    plt.pause(0.1)  # Pause to allow the visualization to update

# Execute the Greedy Best-First Search algorithm
path, explored = greedy_best_first_search(maze, start, goal)

# Final visualization with the optimal path
if path:
    visualize_maze(maze, explored, start, goal, path=path, show_path=True)
```

```
    print(f"Optimal Path: {path}")
    print(f"Cost of Path: {len(path) - 1}")
else:
    print("No path found.")
```

Listing 1: Greedy Best First Search Code

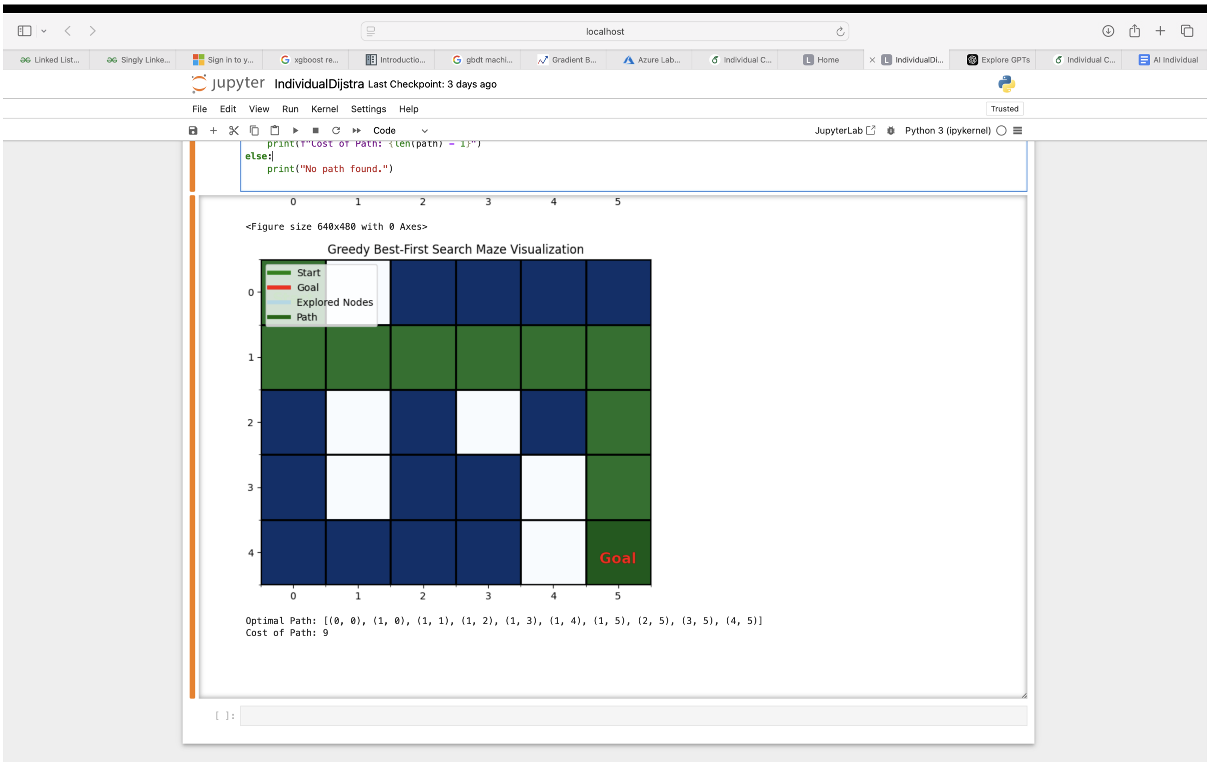

Figure 1: Visualization

5 Backward chaining and Resolution

Backward Chaining is a logical reasoning technique that works in the reverse direction of Forward Chaining. Instead of starting from known facts , Backward Chaining begins with a goal and work backward to determine if the goal can be achieved using a series of rules and known facts.

5.1 Maze Pathfinding with Backward Chaining

- Start at the goal state (4,5) and trace back through the maze rules to see if it connects to the start state (0,0).
- Avoid unnecessary exploration of unrelated paths, focusing only on those contributing to the goal.
- Validate feasibility by confirming whether a continuous path exists from the start to the goal while avoiding obstacles.

Backward Chaining starts at the goal state (4,5) and applies rules to determine possible prior states that could lead to the current state. This continues recursively until the start node (0,0) is reached or all possible paths are exhausted.

5.1.1 Components:

- **State Representation:** Each position in the maze is represented as (row, column).
- **Rules:** Define movement between states, such as If (x', y') leads to (x, y), then (x', y') must be valid (open and unvisited).
- **Visited Set:** Tracks explored nodes to prevent revisiting and redundant computations.

5.2 Execution Tracing Table Using Backward Chaining

If $P(x, y)$ is true (goal position), then $P(x', y')$ must also be true if (x', y') can move to (x, y).

Step	Goal	Inference Rule Applied	New Goal	Path So Far
1	(4,5)	From (4,5), move back to (3,5)	(3,5)	(4,5)
2	(3,5)	From (3,5), move back to (2,5), (4,5)	(2,5)	(3,5) → (4,5)
3	(2,5)	From (2,5), move back to (2,4), (1,5), (3,5)	(2,4), (1,5)	(2,5) → (3,5) → (4,5)
4	(2,4)	From (2,4), move back to (2,5), (1,4)	(1,4)	(2,4) → (2,5) → (3,5) → (4,5)
5	(1,5)	From (1,5), move back to (1,4), (0,5)	(0,5)	(1,5) → (2,4) → (2,5) → (3,5) → (4,5)
6	(1,4)	From (1,4), move back to (1,5), (2,4), (1,3), (0,4)	(0,4), (1,3)	(1,4) → (1,5) → (2,4) → (2,5) → (3,5) → (4,5)
7	(0,5)	From (0,5), move back to (0,4), (1,5)	(0,4)	(0,5) → (1,5) → (2,5) → (3,5) → (4,5)
8	(0,4)	From (0,4), move back to (0,3), (1,4), (0,5)	(0,3)	(0,4) → (0,5) → (1,5) → (2,5) → (3,5) → (4,5)
9	(1,3)	From (1,3), move back to (0,3), (1,2)	(1,2)	(1,3) → (1,4) → (2,4) → (2,5) → (3,5) → (4,5)
10	(0,3)	From (0,3), move back to (0,2), (0,4), (1,2)	(0,2)	(0,3) → (1,3) → (1,4) → (2,4) → (2,5) → (3,5) → (4,5)
11	(1,2)	From (1,2), move back to (0,2), (1,1), (2,2)	(1,1)	(1,2) → (1,3) → (1,4) → (2,4) → (2,5) → (3,5) → (4,5)
12	(1,1)	From (1,1), move back to (1,0), (1,2)	(1,0)	(1,1) → (1,2) → (1,3) → (1,4) → (2,4) → (2,5) → (3,5) → (4,5)
13	(1,0)	From (1,0), move back to (0,0), (2,0)	(0,0)	(0,0) → (1,0) → (1,1) → (1,2) → (1,3) → (1,4) → (2,4) → (2,5) → (3,5) → (4,5)

Final Path:

$$(0,0) \rightarrow (1,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,4) \rightarrow (1,5) \rightarrow (2,5) \rightarrow (3,5) \rightarrow (4,5)$$

5.3 Advantages of Backward Chaining for Maze-Solving

- Focuses on paths leading to the goal rather than exploring all possibilities.

- Avoids redundant exploration by only considering paths contributing to the goal.
- Provides clear reasoning for why the goal is achievable based on logical deductions.

5.4 GMP Definition & Inference Rule

GMP applies the logical rule:

$$\text{If } A \to B \text{ and } A \text{ is true, then } B \text{ is true.}$$

For maze-solving:

$$A \text{ can represent the "current position"}$$

$$B \text{ can represent the "next valid position."}$$

5.5 Backtracking with GMP

- Start at the initial position (e.g., $P(0,0)$).
- Use GMP to deduce the next possible valid moves.
- Explore one move at a time.
- If a move leads to a dead end, backtrack to the previous position and try another valid move.

5.6 How GMP Works in Backtracking

Initial Setup:

- Known facts (starting point):
$$F = \{P(0,0)\}.$$
- Goal: Reach $P(4,5)$.
- Rule:
$$P(x,y) \to P(x',y'),$$
where $P(x',y')$ is:
  - A valid neighboring cell.
  - Open (no obstacle).
  - Not already visited in the current path.

Recursive Exploration:

- At each step, GMP deduces possible valid moves $P(x',y')$ from the current position $P(x,y)$.
- Choose one move and recursively continue the process.
- If the goal is reached, stop.
- If no valid moves are left, backtrack to the previous step and explore a different move.

```python
import matplotlib.pyplot as plt
import numpy as np
import time

# Define the maze dimensions
rows, cols = 5, 6  # 5 rows and 6 columns

# Define blocked and open nodes
blocked_nodes = {(0, 1), (2, 1), (3, 1), (2, 3), (3, 4), (4, 4)}  # Predefined blocked nodes
open_nodes = {
    (r, c) for r in range(rows) for c in range(cols)
} - blocked_nodes  # Open nodes are all grid cells minus the blocked ones

# Define the start and goal positions
start = (0, 0)  # Starting position
goal = (4, 5)  # Goal position

# Define the possible moves (4 directions: up, down, left, right)
moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Each move represented as a change in row and column

# Helper function to get valid neighbors of a node
def get_neighbors(node):
    neighbors = []  # List to store valid neighbors
    for dr, dc in moves:  # Iterate over possible moves
        new_r, new_c = node[0] + dr, node[1] + dc  # Calculate new position
        if (new_r, new_c) in open_nodes:  # Check if the position is open
            neighbors.append((new_r, new_c))  # Add valid neighbor to the list
    return neighbors  # Return all valid neighbors



def visualize_maze(facts, new_facts, visited, step, path=None, final=False):
    plt.figure(figsize=(8, 6))
    plt.title(f"Step {step}" if not final else "Final Path", fontsize=16)

    # Set the background color to gray
    plt.gcf().set_facecolor('gray')  # Set the entire figure's background to gray

    # Draw grid lines
    for r in range(rows + 1):
        plt.axhline(r - 0.5, color="black", linewidth=0.5)
    for c in range(cols + 1):
        plt.axvline(c - 0.5, color="black", linewidth=0.5)

    # Mark blocked cells in red (e.g., obstacles)
    for r, c in blocked_nodes:
        plt.text(c, r, "X", color="red", fontsize=20, ha="center", va="center")

    # Mark visited cells with a dark color (e.g., dark orange)
    for r, c in visited:
        plt.text(c, r, "    ", color="darkorange", fontsize=20, ha="center", va="center")

    # Highlight newly inferred facts with a dark color (e.g., dark blue)
    for r, c in new_facts:
        plt.text(c, r, "*", color="darkblue", fontsize=20, ha="center", va="center")

    # Mark the shortest path if available
    if path:
```

```
        for i in range(len(path) - 1):
            r1, c1 = path[i]
            r2, c2 = path[i + 1]
            plt.plot([c1, c2], [r1, r2], color="green", linewidth=2)
        for r, c in path:
            plt.text(c, r, "Path", color="green", fontsize=12, ha="center", va="center", fontweight="bold")

    # Label start and goal nodes with specific colors
    plt.text(start[1], start[0], "S", color="red", fontsize=20, ha="center", va="center", fontweight="bold")
    plt.text(goal[1], goal[0], "G", color="blue", fontsize=20, ha="center", va="center", fontweight="bold")

    # Set axis limits and hide ticks for better visualization
    plt.xlim(-0.5, cols - 0.5)
    plt.ylim(rows - 0.5, -0.5)
    plt.xticks([])
    plt.yticks([])

    # Pause for better visualization; adjust time for final step
    plt.pause(0.5 if not final else 3)
    plt.show()  # Use plt.show() instead of plt.close() to display the plot


# Function to solve the maze using backward chaining and visualize the trace
def solve_maze_with_backward_trace(start, goal):
    facts = {goal}  # Initialize known facts with the goal node
    visited = set()  # Keep track of visited nodes
    path = {goal: None}  # Dictionary to reconstruct the path later
    step = 1  # Step counter

    # Loop until the start node is reached
    while start not in facts:
        print(f"Step {step}:")  # Print current step
        new_facts = set()  # Facts to be inferred in this step
        # Process each fact
        for fact in facts:
            if fact not in visited:  # Skip already visited nodes
                neighbors = get_neighbors(fact)  # Get valid neighbors
                print(f"  Expanding node {fact}, valid node: {neighbors}")
                for neighbor in neighbors:
                    if neighbor not in facts:  # If it's a new fact, add it
                        new_facts.add(neighbor)
                        path[neighbor] = fact  # Record the parent for path reconstruction
                visited.add(fact)  # Mark as visited
        # Visualize the current step
        visualize_maze(facts, new_facts, visited, step)
        # Update known facts with newly inferred facts
        facts.update(new_facts)
        print(f"  Derived facts this step: {new_facts}")
        print(f"  Knowledge base after update: {facts}\n")
        # Increment step
        step += 1
        # Stop if no new facts can be inferred
        if not new_facts:
            print("No more inferences can be made. Start is unreachable!")
            return None
    # Start reached
    print(f"Start {start} reached in {step - 1} steps!")
    print("\nReconstructing path:")
    # Reconstruct the path from start to goal
```

```python
    current = start
    shortest_path = []
    while current:
        shortest_path.append(current)  # Add current node to path
        current = path[current]  # Move to the parent node
    print(f"Shortest path: {shortest_path}")
    print(f"Path cost: {len(shortest_path) - 1} steps")  # Calculate and print
   path cost
    # Visualize the final path
    visualize_maze(facts, new_facts, visited, step, shortest_path, final=True)
    return shortest_path  # Return the shortest path

# Solve the maze and visualize the trace
shortest_path = solve_maze_with_backward_trace(start, goal)
```

Listing 2: Backward Chaining Code

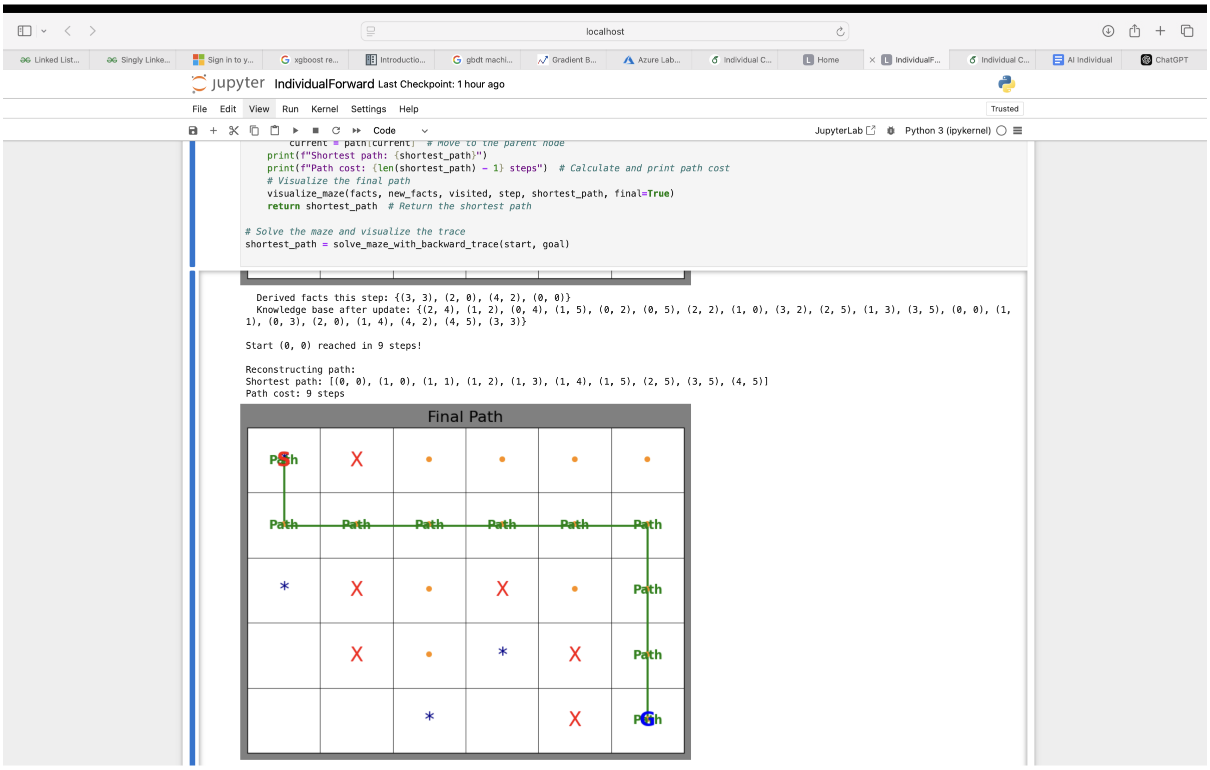

Figure 2: Backward Chaining