CHANAKYA UNIVERSITY
Rooted in ideals • Ascending with ideas

CU Anveshan (BotBrain Campus Navigator)

An Intelligent Pathfinding Application for Seamless Navigation

Abhilash N S Reddy 24UG00303

Mission: Simplifying Campus Navigation

Goal: To develop a smart, user-friendly desktop application that simplifies navigation across a university campus.

Problem: New students, visitors, and even existing members of the campus community can find it challenging to locate buildings and find the most efficient routes.

Solution: Create an application that combines a clear, interactive map with powerful pathfinding algorithms and an intelligent, AI-powered search function. This tool will allow users to find the best path using either manual selections or natural, everyday language.

What the Application Can Do

Interactive Campus Map: A dynamic visual representation of the campus where users can see all key locations and the paths connecting them.

Dual Pathfinding Modes:

- Automatic Mode: Intelligently selects the best algorithm to find the optimal path without requiring technical knowledge from the user.
- Manual Mode: Allows users to experiment and compare different classic pathfinding algorithms: BFS, UCS, A*, and Dijkstra.

AI-Powered Search (BotBrain): The core innovation—users can type queries in plain English, such as "Show me the path from the main gate to the hostel."

- Dynamic Path Visualization: The calculated route is instantly highlighted on the map for easy understanding.
- Location Information & Image Previews: Displays key details and a picture of the destination to help users recognize it upon arrival.

Path Finding Algorithms

1. Breadth-First Search (BFS)
Principle: A "brute-force" algorithm that explores a graph layer by layer, moving outward from the starting point.
How it Searches:
Starts at the source node.
Visits all of the source's immediate neighbors.
Then, for each of those neighbors, it visits their unvisited neighbors.
Continues this process layer by layer until the goal is found.
Uses a queue (First-In, First-Out) to manage which node to visit next.
Guarantees: Finding the path with the fewest number of edges (or steps). It does not consider the cost or distance of the edges.

2. Uniform Cost Search (UCS)
Principle: An algorithm that explores a graph by always expanding the path that has the lowest total cost from the start.
How it Searches:
Starts at the source node.
Continuously expands the unvisited node that has the lowest total path cost from the source.
Uses a priority queue to keep track of paths, ordered by their cost.
Guarantees: Finding the path with the minimum possible total cost. It is functionally the same as Dijkstra's algorithm when edge costs are non-negative.

3. Dijkstra's Algorithm
Principle: A foundational algorithm for finding the shortest paths between nodes in a weighted graph with non-negative edge weights.
How it Searches:
Starts at the source node and marks its initial distance as 0.
At each step, it selects the unvisited node with the smallest known distance.
It then "relaxes" the path to each of its neighbors, updating their distances if a shorter path is found.
Once a node is visited, its shortest path is considered final.
Guarantees: Finding the absolute lowest-cost path from a single source node to all other nodes in the graph.

4. A* (A-Star) Search
Principle: An "informed" or "smart" search algorithm that uses a heuristic (an educated guess) to find the destination more quickly.
How it Searches:
It combines the approach of Dijkstra's algorithm with a heuristic function.
It determines which path to expand next based on two factors:
g(n): The actual cost of the path from the start to the current node n.
h(n): The estimated cost from the current node n to the goal.
It prioritizes paths with the lowest combined value of g(n) + h(n).
Guarantees: Finding the lowest-cost path (like Dijkstra) but is generally much more efficient because it avoids exploring paths that are headed in the wrong direction.

Technical Architecture

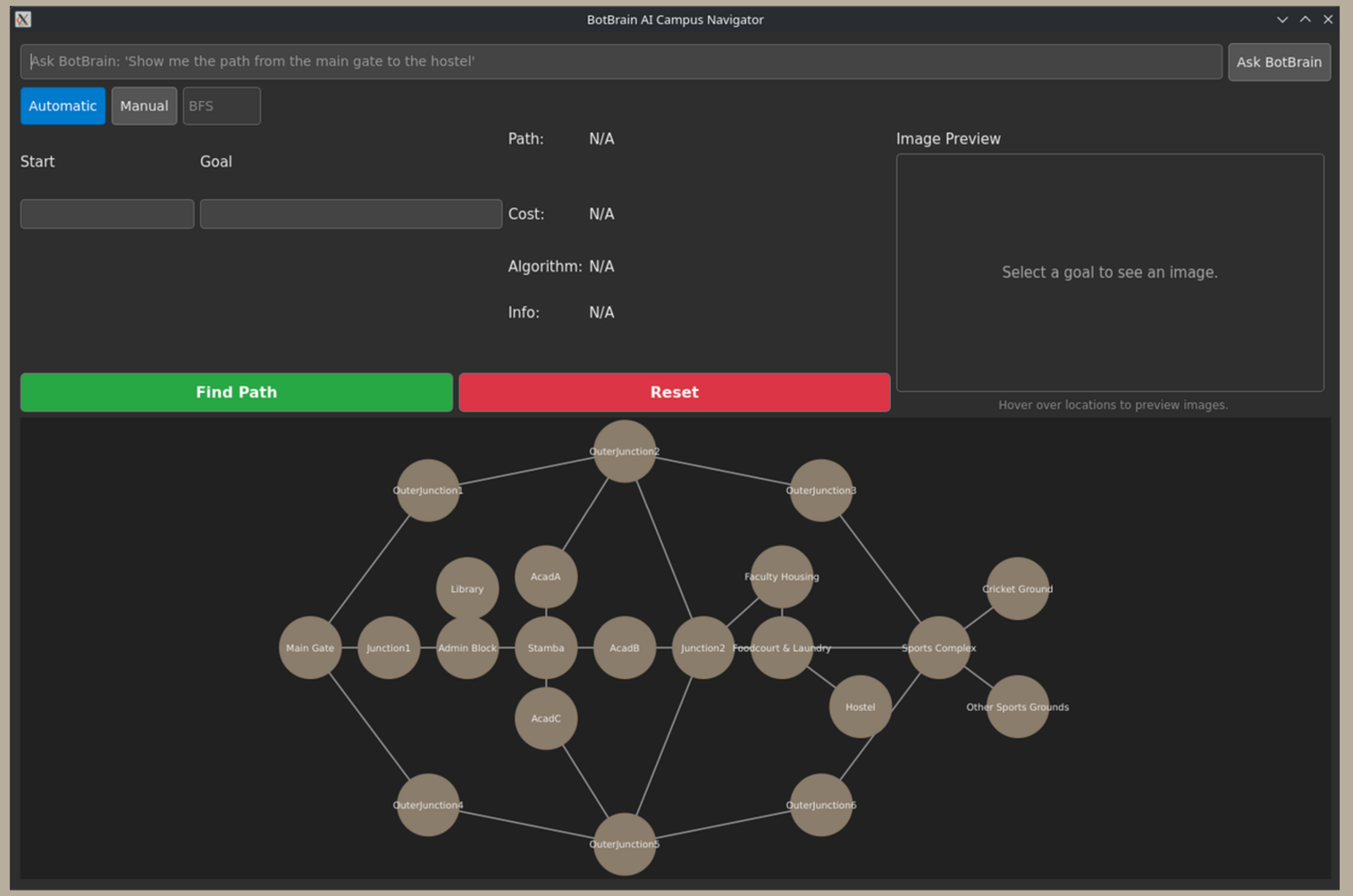
Programming Language: Python – Chosen for its extensive libraries and readability.

User Interface (GUI): PyQt5 – A robust framework used to build the interactive and responsive desktop application.

Campus Representation: The entire campus layout is modeled as a graph data structure:

Nodes: Represent campus locations (e.g., "Library," "Admin Block").

Edges: Represent the paths between locations, weighted by distance.



The "BotBrain" AI Integration

Natural Language Processing (NLP): The application leverages a local Large Language Model (LLM), phi3:mini, to interpret user commands.

The Process:

A user types a query like, "I'm at the food court and need to get to the sports complex."

The botbrain_ai.py module securely sends this text to the local phi3:mini model.

The AI has been trained with a specific prompt to parse the text and accurately identify the "source" and "destination" locations from a predefined list.

It returns a clean, structured JSON output (e.g., {"source": "Foodcourt & Laundry", "destination": "Sports Complex"}).

The main application uses this structured data to set the start and end points for the pathfinding algorithm.

Retrieval-Augmented Generation (RAG): The app also uses a simple RAG approach to pull building information from campus_info.json, ensuring the user gets relevant details about their destination.

```python
VALID_LOCATIONS = [
    "Main Gate", "Junction1", "Admin Block", "Library", "Stamba", "AcadA",
    "AcadB", "AcadC", "Junction2", "Faculty Housing", "Foodcourt & Laundry",
    "Hostel", "Sports Complex", "Cricket Ground", "Other Sports Grounds",
    "OuterJunction1", "OuterJunction2", "OuterJunction3", "OuterJunction4",
    "OuterJunction5", "OuterJunction6"
]

def parse_user_query(user_text: str) -> tuple[str | None, str | None]:
    """
    Uses a local LLM (phi-3:mini) to parse a natural language query and extract
    a valid source and destination.

    Args:
        user_text: The natural language input from the user.

    Returns:
        A tuple containing (source, destination) if successful, otherwise (None, None).
    """
    # This detailed prompt guides the LLM to give us a clean, predictable JSON output.
    prompt = f"""
    You are a parsing assistant for a campus navigation bot. Your only job is to extract the source and destination from the user's text.
    The locations MUST be one of these exact names: {VALID_LOCATIONS}.
    Map common names to their official names (e.g., "academic block a" -> "AcadA", "hostel" -> "Hostel").
    Respond ONLY with a single, clean JSON object containing "source" and "destination" keys. Do not add any other text or explanations.

    Example 1:
    User: "Show me the way from the main gate to the admin building"
    {{"source": "Main Gate", "destination": "Admin Block"}}

    Example 2:
    User: "I'm at the hostel and need to go to the library"
    {{"source": "Hostel", "destination": "Library"}}

    Example 3:
    User: "route from food court to cricket ground"
    {{"source": "Foodcourt & Laundry", "destination": "Cricket Ground"}}

    Now, parse this user's text:
    User: "{user_text}"
    """
    try:
        response = ollama.chat(
            model='phi3:mini',
            messages=[{'role': 'user', 'content': prompt}],
            format='json', # Force the model to output valid JSON
        )
        data = json.loads(response['message']['content'])
        source = data.get('source')
        destination = data.get('destination')

        # Final validation to ensure the LLM didn't hallucinate a location
        if source in VALID_LOCATIONS and destination in VALID_LOCATIONS:
            return source, destination
        return None, None
    except Exception as e:
        print(f"An error occurred while parsing the query: {e}")
        return None, None

def get_building_info(location_name: str) -> str:
    """
    Retrieves information about a building from the campus_info.json file.
    This acts as the 'Retrieval' step in a simple RAG system.
    """
    try:
        with open('campus_info.json', 'r') as f:
            knowledge_base = json.load(f)
        return knowledge_base.get(location_name, "No specific information available.")
    except FileNotFoundError:
        return "Knowledge base file (campus_info.json) not found."
```

Application Showcase

Key UI Elements:

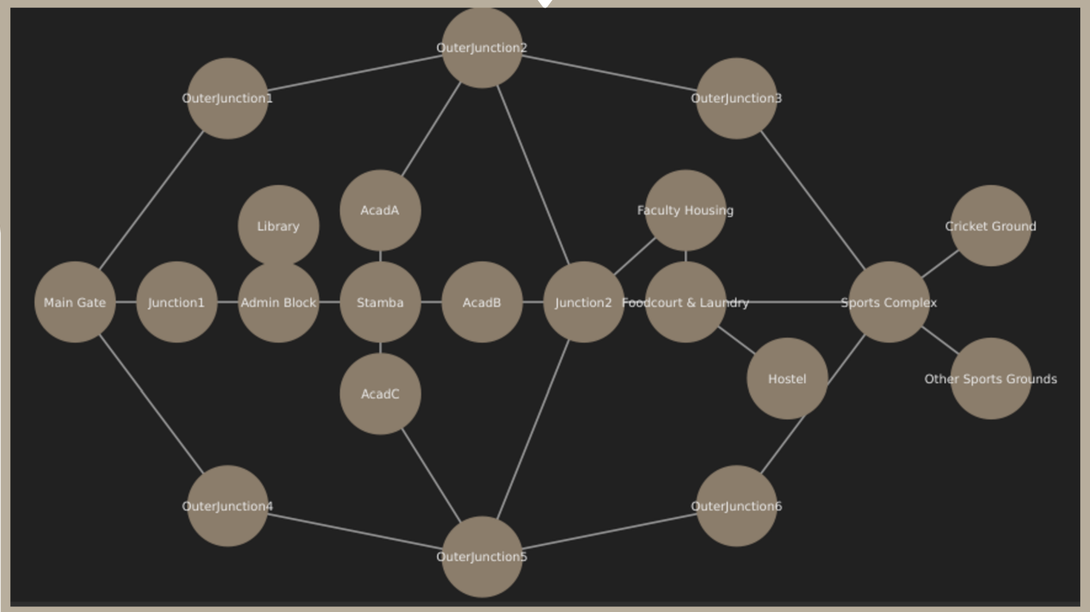
Map View: The central, interactive map where paths are visualized.

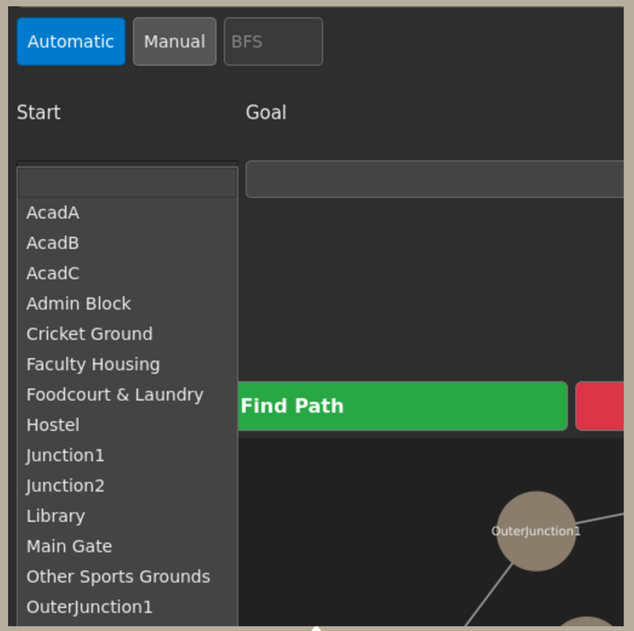
Control Panel: Dropdown menus for manually selecting start and goal locations and choosing an algorithm.

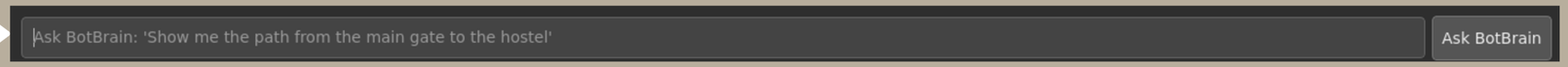
AI Query Bar: The "Ask BotBrain" text field where users can type their natural language queries.

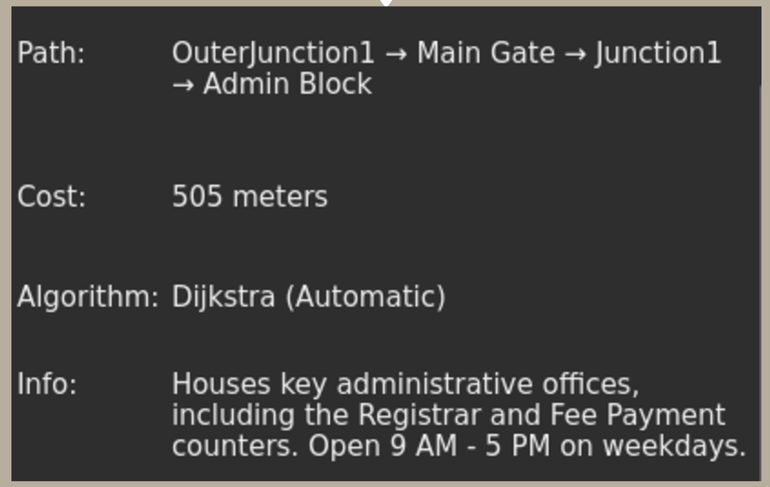
Results Panel: Clearly displays the step-by-step path, total distance/cost, and information about the destination.

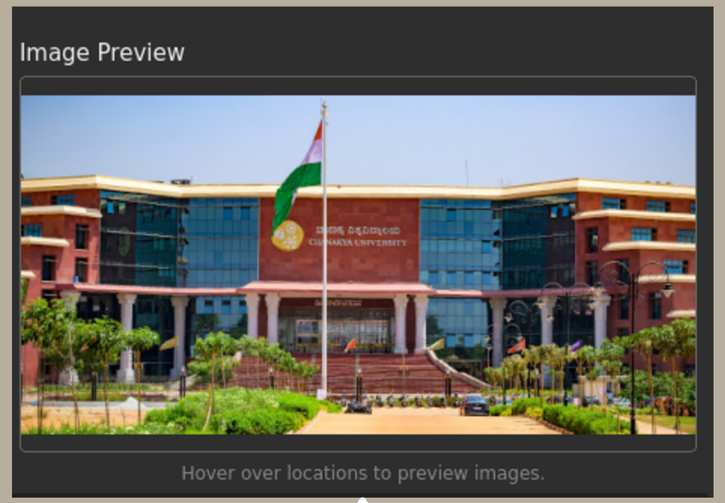
Image Preview: A helpful visual aid showing a picture of the selected location.











Future Scope

Mobile Application: Develop a version for Android and iOS for on-the-go convenience.
Real-time Location: Integrate with GPS to show the user's live position on the map.
Expanded Knowledge Base: Add more dynamic information, such as building hours, ongoing events, and faculty office directories.
Accessibility Features: Introduce options to find routes that are wheelchair-accessible or avoid stairs.

Thank you. :p