Campus Navigation: Pathfinding and Mapping using Dijkstra

2024005625 M.S.S. VARUN

Acknowledgement

For the most part, this has been a solo journey—and honestly, it's been both fun and challenging. It had its ups and downs, but throughout the process, I learned a lot—especially about Python's syntax.

This project made me realize that the limits of a program are really just the limits of your imagination.
Going from an idea to a working solution taught me an important lesson: there are always unexpected problems, and things can go wrong in ways you would've never planned for.
And maybe because of that, I was definitely a different student before this project, because this project, and not just what is in the end product, but also everything that was done in the back-end, it taught me a great deal on how to bring your idea to life.
This experience helped me understand how important it is to think ahead and stay flexible during development.

A special mention goes to **Anany Levitin's *The Design and Analysis of Algorithms***, which helped me build a foundation for the algorithmic part of the project.

I'm also really thankful for today's AI tools. They've been like virtual teammates—helping me debug issues, explore ideas, and test different code approaches. A special mention to ChatGPT and Stack Overflow, the two that played a big part in guiding me and speeding up my learning.

Finally, thanks to every blog post, forum comment, YouTube video and documentation page I stumbled upon during this journey—you were quiet but powerful teachers.

This project pushed me to grow as a problem solver, and I'm proud of the skills and mindset I've built, and insights gathered along the way.

Introduction

Project Overview

This project presents a comprehensive campus-wide pathfinding and analytics system built on top of Dijkstra's Algorithm, visualized in real-time over a custom-drawn, scaled map of GITAM University. More than just a shortest-path application, this project explores how efficiently the campus is connected algorithmically.

At its core, the system allows users to select any two valid points—such as departments, hostels, or schools—and computes the shortest route between them using a modularised python code. The process is then animated live using OpenCV, showcasing not just the result but the full algorithmic journey.

This project emphasizes:

- Real-World Graph Modelling: Each location and intersection on campus are represented as a node, with all paths modelled as weighted, bidirectional edges based on real distances taken from google earth.
- Handcrafted Dataset & Mapping: manually curated to reflect the actual campus layout. This includes accurate coordinate mapping, node labelling, and real-world distance calculations.
- Dijkstra's Algorithm Implementation with Live visualization: implementation of the shortest path algorithm, optimized for step-by-step visualization and animation of both exploration phase and final shortest path found.
- Statistical Analytics: Rank locations based on centrality and accessibility, identify bridge nodes, Highlight the most frequently used segments

Real-World Relevance of Pathfinding

The need to determine optimal paths through complex environments—whether for efficiency, safety, or accessibility—is a fundamental problem with broad practical implications.
In academic campuses, hospitals, malls, and smart cities, effective pathfinding systems enable for Faster and more informed navigation and efficient Resource optimization in logistics and transport

By applying Dijkstra's algorithm, this project bridges the gap between theoretical computer science and physical spatial layouts, showing algorithmic thinking can be used to solve practical location-based challenges in environments where distances, turns, and landmarks play a critical role.

This project is built in such a way that it can be scaled further up, which will shown in detail later.

Use of GITAM Campus as a Model

The GITAM University campus was chosen as the model environment for this project due to its diverse layout, real-world relevance, and accessibility for manual mapping. With a mix of academic blocks, hostels, canteens, and intersections, it provides a realistic yet manageable setting to simulate and test a pathfinding system.

This allows one to understand how Dijkstra's algorithm works in real world environments.

Scope of the Project

The visual map used is a model inspired by the actual GITAM campus layout. While proportions and connections reflect real-world logic, it is not an exact or pinpoint-accurate replica. Instead, it offers a simplified and structured framework designed for clarity and demonstrative value.

- Functional Scope:

Models the environment as a weighted undirected graph, where nodes represent intersections or locations, and edges represent walkable paths with assigned distances.
Supports modular expansion with additional nodes, distance recalibration, or different search algorithms.

- Visual Demonstration Scope:

Displays the progression of the algorithm in real-time: node exploration, path updates, and final path.
Logs and highlights the search process, including explored nodes and final optimal path.

Expected Outcomes:

Users can find the shortest walking path between any two campus locations with case as pathfinding process is shown live through a visual animation.
This project can also be a learning tool for understanding Dijkstra's algorithm.

Future Scalability & Expansion Plans

- Multiple Algorithm Support:

Add support for A*, Bidirectional Dijkstra, and Floyd-Warshall to enable performance comparisons and advanced routing options.

- Real-Time Edge Weight Adjustments:

Dynamically update path costs using live data like traffic density, crowd levels, mode of transportation or temporary blockages (some parts of campus are restricted for vehicular access)

- Interactive GUI Integration:

Replace terminal input with a user-friendly interface that allows users to pick start and end points directly on the map with instant visual feedback.

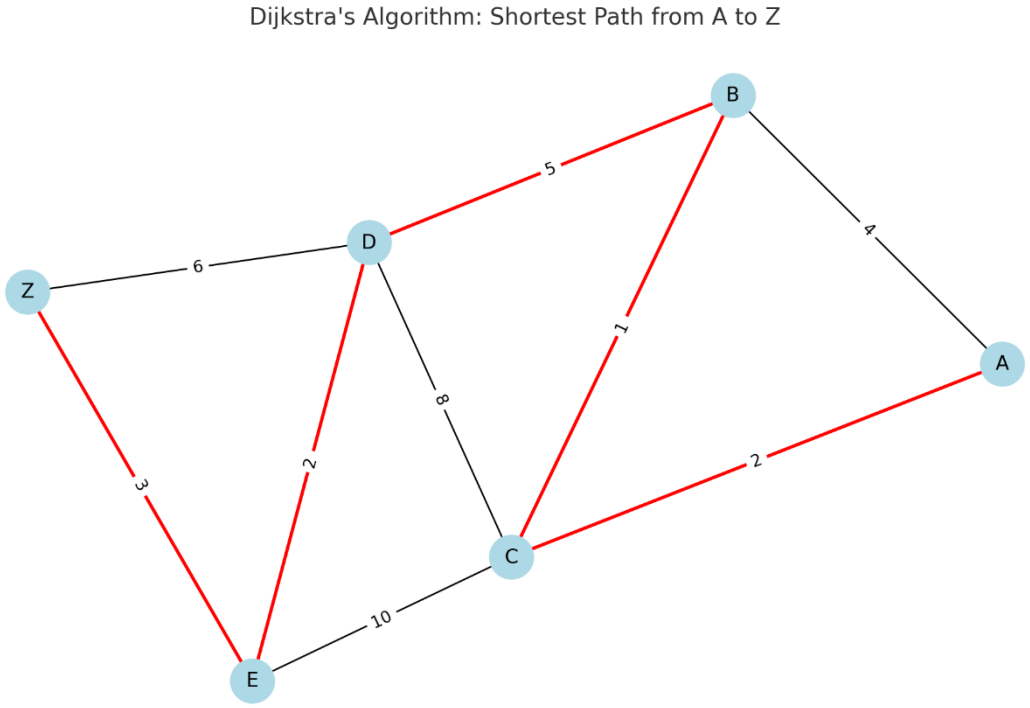
Understanding Dijkstra's Algorithm in this Project

Concept and Use-Cases

Dijkstra's Algorithm is a classic graph-based algorithm used to find the **shortest path** between nodes in a graph with **non-negative edge weights** (its biggest if not only shortcomings).
This algorithm being a greedy guarantees the shortest distance from a **starting node to all other reachable nodes**, making it ideal for routing problems.

Detailed Explanation on a Simple 7–10 Node Graph

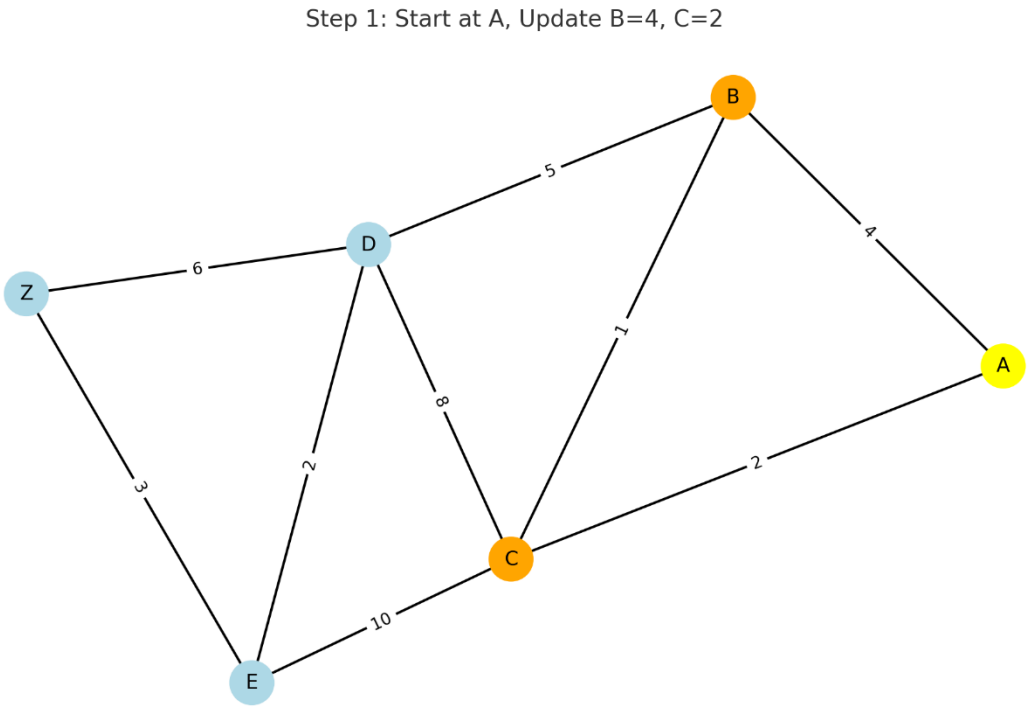
Let's take a small example to understand how Dijkstra works.



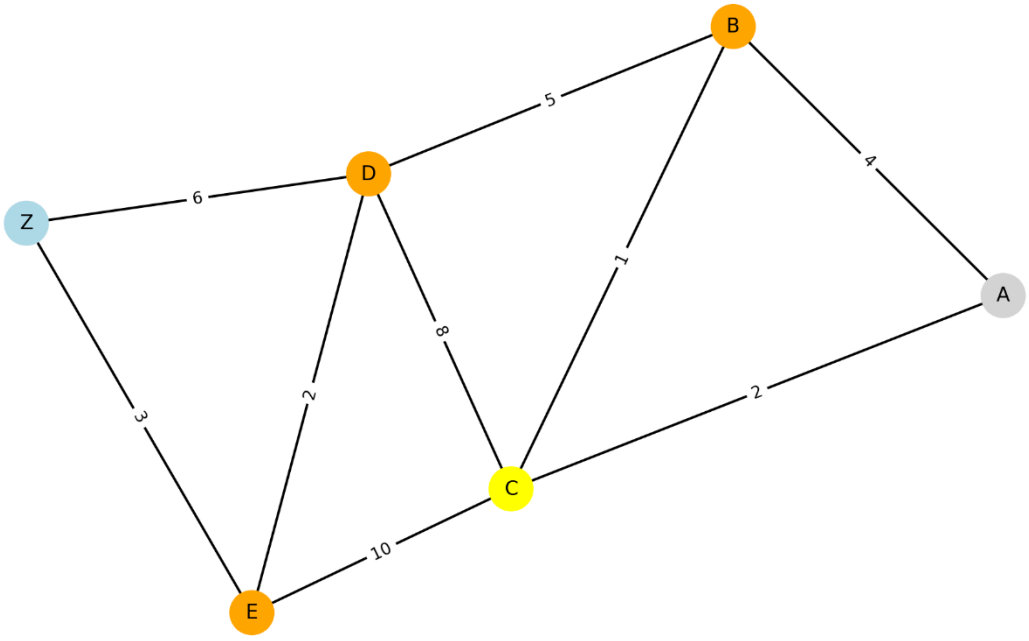
Consider the above graph, lets find the shortest path between A to Z.
It has Nodes: A, B, C, D, E, Z with non-negative weights representing distances.

Let's set all node distances to ∞ (except start node A = 0) and also maintain a **queue** that is sorted every single time, an entry is made to it, which would enable us to select the lowest cost node always, making it a greedy approach.
Start at A:
A → B = 4, A → C = 2
Update: B = 4, C = 2

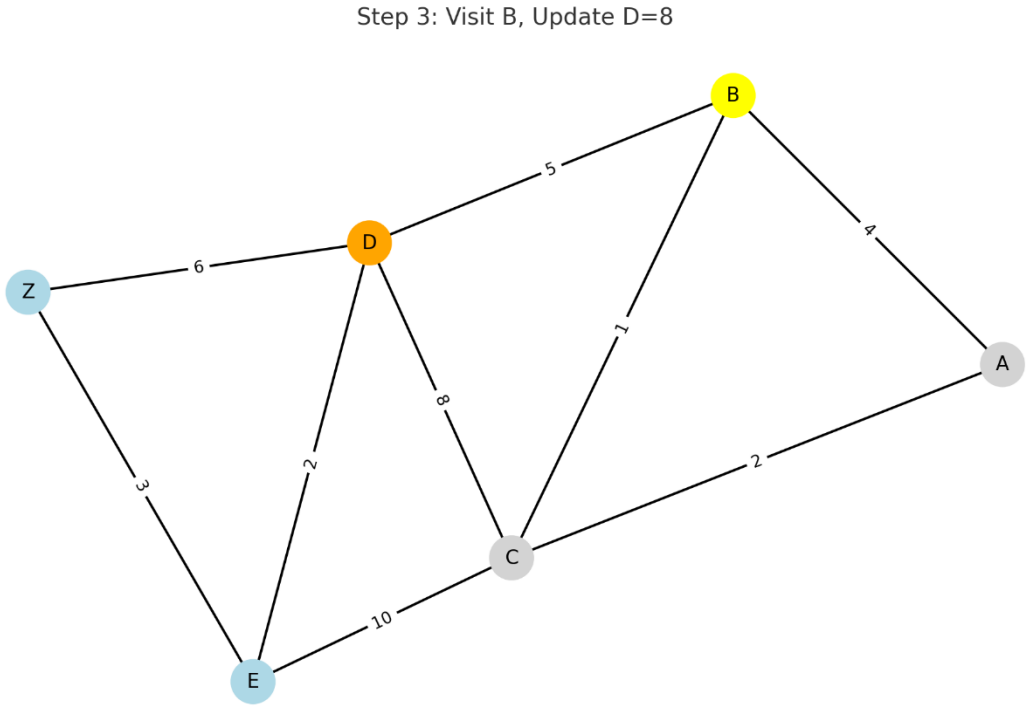
Campus Navigation: Pathfinding and Mapping using Dijkstra



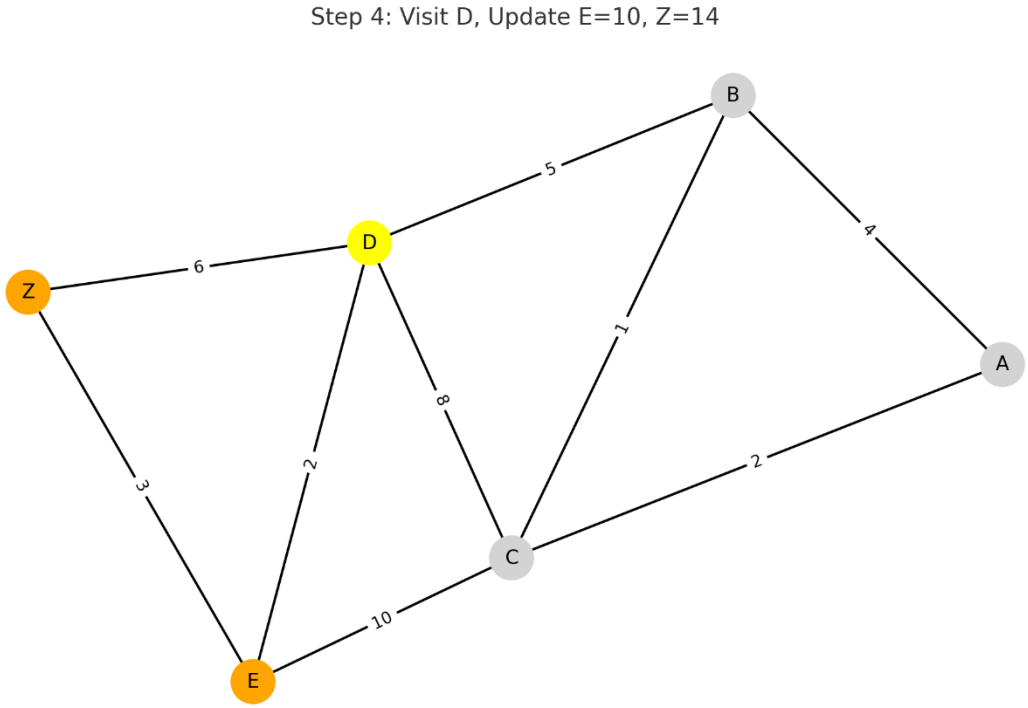
Move to C (lowest current cost = 2):
C → B = 1 → New cost to B = 2 + 1 = 3 (better than 4 → update B)
C → D = 8 → D = 10
C → E = 10 → E = 12



Next lowest: B (cost 3):
B → D = 5 → New cost = 3 + 5 = 8 (better than 10 → update D)

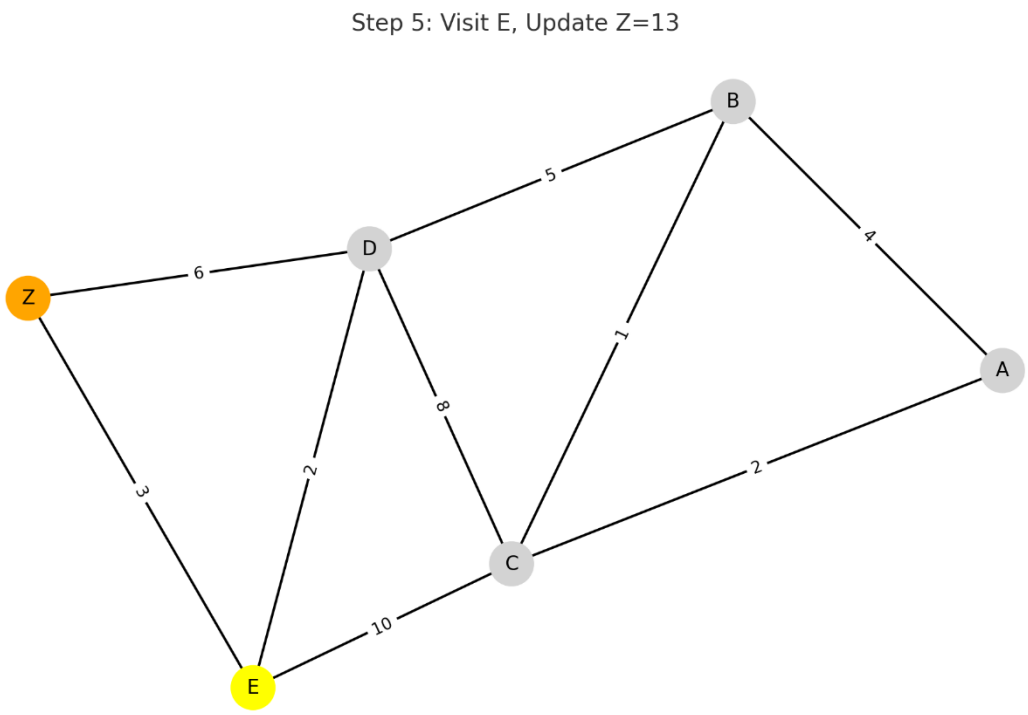


Next lowest: D (cost 8):
D → E = 2 → New cost = 8 + 2 = 10 (better than 12 → update E)
D → Z = 6 → Z = 14

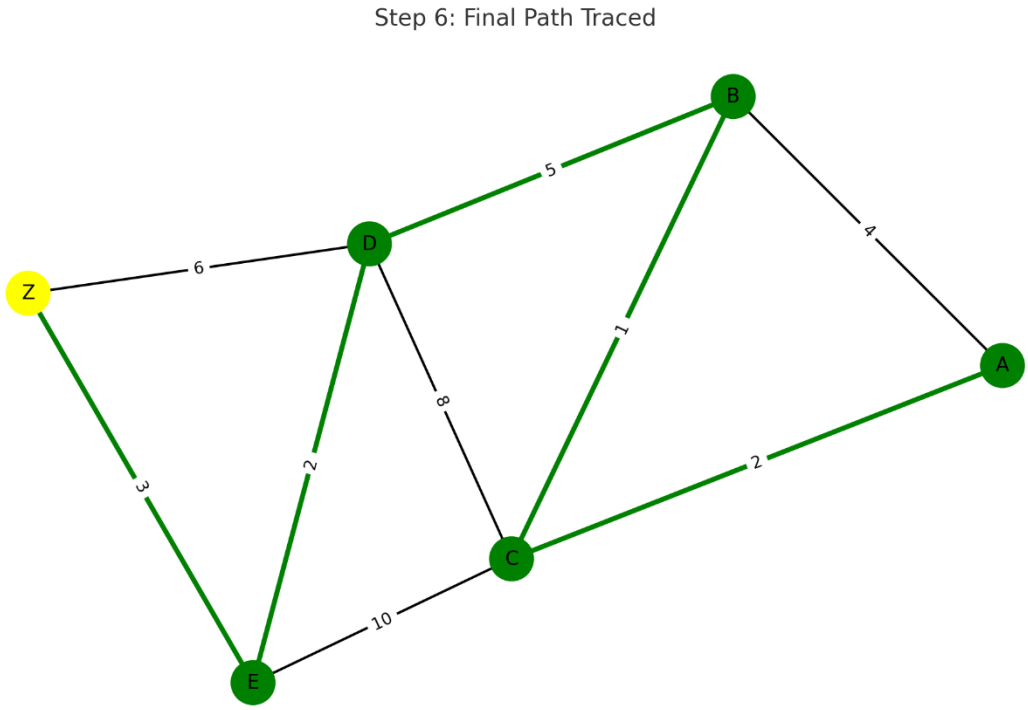


Next: E (cost 10):

E → Z = 3 → New cost = 10 + 3 = 13 (better than 14 → update Z)



Final shortest path from A to Z:
A → C → B → D → E → Z with total cost **13**



Working Principle: Greedy Approach

Dijkstra's Algorithm follows a greedy strategy by always selecting the nearest unvisited node at each step.
It makes locally optimal choices to find the global shortest path.
Once a node's shortest distance is confirmed, it's never revisited.
This works because all edge weights are non-negative, ensuring correctness.
The algorithm commits early, assuming the shortest path so far is also the best—and it usually is.

Tools, Libraries & Tech Stack

Python Version and Environment:

- Python Version: 3.12.7
- Environment: Anaconda (Windows)
- IDE Used: Visual Studio Code with GitHub Codespaces for debugging
- Libraries: cv2 (OpenCV), NumPy.
- Execution Mode: Local runtime (no web hosting used)

Key Libraries:

- **OpenCV (cv2)**
  Used for real-time visualization of the campus map, including node highlighting, path animations, and frame rendering
- **os and sys**
  Handles file paths, execution flow, and modular imports across different environments, not mandatory until you have trouble importing files across
- **itertools**
  Used for generating all possible location pairs in analytics

File Overview

Overview of the Map (Gitam_map.png):

The base visual for this project is a custom-drawn campus map (gitam_map.png), created using Canva.com.
It offers a clear, scalable, and structured representation of GITAM University, tailored for effective algorithmic visualization and routing.
Map Size: 12500 × 12500 pixels (uniformly scaled for accurate coordinate mapping)
Note: The map is not a satellite image. It's a logic-driven model created to support pathfinding clarity, not geographical precision.
Access: map on GitHub
Check the last page for a physical copy

Node System Structure and labelling (Nodes.py):

Declares lists of node names for convenience. It splits nodes into two categories: intersections and locations, then combines them into nodes
The project uses a two-colour node system for clarity:

- 🔴 Red Nodes: User-facing destinations such as departments, hostels, and canteens. These are labelled as acronyms of locations all across Gitam, like 'GB' – Gitam Bhavan, 'GSB' – Gitam School of Business.
- ⚫ Black Nodes: Intersections, turns, and routing points used internally for path logic. These are labelled as i1, i2 ……. i34.

This dual-layer system maintains a clean split between nodes meant for logic (black) and those meant for users (red).

If new locations or intersections added, update these lists

Coordinate Allocation (coords.py):

Each node is mapped to a pixel coordinate on gitam_map.png.
The coordinates enable accurate visual plotting and animated path drawing.
These are used directly during pathfinding and visualization.

Houses the coordinates for each node on the campus map image, this allows the visualizer to know where to draw each node on the image. The coordinate system corresponds to the original gitam_map.png dimensions.

It has two sections in it, as indicated by comments:

- **Intersections** – e.g., 'i2': (5318, 5506), 'i3': (6445, 5502), ...
- **Locations** – e.g., 'NTR': (3761, 5519), 'VDC': (4370, 5519), ... 'BG': (2771, 5526). These correspond to the actual places like NTR Gardens, etc.

These coordinates were manually measured using MS paint so they will need re-designing if the map changes.

Graph Construction (graph.py):

The entire campus is modelled as a weighted, undirected graph, where each node is a key point, and each edge is a walkable path, into a dictionary.
The weights represent approximate distances (since they are visually estimated, not GIS-derived).
This graph dictionary is Optimized for:

- Quick neighbour lookup
- Direct use in Dijkstra's pathfinding without needing pre-processing

Each key in the outer dictionary is a node (intersection or location), and the value is another dictionary of neighbours with their associated edge weights (distances).
This file doesn't have functions or logic – it's essentially data that other files import
The weights are taken in meters,
While the visual layout closely mirrors the actual campus, the core focus is on logic.

Core Algorithm (dijkstra.py):

This file has the core pathfinding algorithm used in the campus map visualizer and analyser.
It calculates the shortest path between a start and end node in the provided graph, also tracking the order of exploration for animated visualization.

```
dijkstra (start: str, end: str, graph: dict)
```

- Parameters:

start (str): The starting node
end (str): The target node.
graph (dict): The adjacency list of the graph in the form.
{node: {neighbour: weight, ...}, ...}.

- Returns:

dict with items:
("distance" (float): Total weight (distance) of the shortest path.)
("path" (list[str]): Ordered nodes in the shortest path from start to end.)
("visited" (list[str]): Nodes permanently processed by the algorithm.)
("explored_order" (list[str]): The sequence of node visits (for animation))

- Key Methods Used:

list.sort(): Used to implement the greedy approach, by picking the smallest quantity first (queue.sort(key=lambda x: x[0])).
This can be replaced with a heapq for large graphs (O(n log n) per sort).

.get() on dictionary: graph.get(node, {}) ensures that even if a node is not in the graph, it won't crash (Safe to fail you may say).

Explanation:

- Initial State: Starting node always has current distance of 0.
  All other nodes are considered at infinite distance until explored.
  A list of tuples called queue holds nodes to visit next, sorted for the shortest distance found so far. Initially, only the start node is in the queue. As nodes are added to the queue with updated costs, pop the one with lowest cost.
  The first popped node represents the shortest known path to that node

- Visit Nodes (Repeatedly):

Choose the minimum most distant node to explore(greedy), If that it was visited already, skip it, otherwise mark it as visited.
Check if your current node is the end location, if so, terminate the process and return the data of explored order and costs.

Now if there exists a neighbour to the current, that has not yet been visited, then calculated the cost till that neighbour, for all such neighbours.

Even if a shorter path is found to a node already in the queue, push it again.
When it's popped later, if it's already visited, skip it—
This ensures only shortest path is finalised

- Loop until the priority queue is empty (meaning all reachable nodes are finalized) or until the target is found.

Return the final path taken to reach the destination, which is found the moment the end node is popped from the queue.

Visualization System (visualizer.py)

Contains all functions related to **visualizing the pathfinding** on the map image. It's the most complex file, because it ties together the data (graph and coordinates), the algorithm (Dijkstra's), and the OpenCV drawing routines.

The key responsibilities of this file include:

- Reading the image and preparing it for drawing (resizing with correct aspect ratio).
- Drawing static elements like the legend
- Animating the pathfinding algorithm's progress

- Drawing the final shortest path.

Function Documentation (order of definition in file)

scale_coords (x: int, y: int, original_shape: tuple, new_shape: tuple, pad_x: int, pad_y: int)

Scale raw image coordinates to the new image size with padding.

- x, y – Original coordinates on the full-size map image.
- original_shape – The height and width of the original image as a tuple (orig_h, orig_w)
- new_shape – The height and width of the resized image
- pad_x, pad_y – The horizontal and vertical padding added to window to place the image in center
- Returns: A tuple (new_x, new_y) which are integer coordinates on the resized+padded image w.r.t (x,y)

This function computes scale factors for x and y and uses them to calculate scale coordinates.

scaled_coords_for(node: str, coords: dict, original_shape: tuple, new_shape: tuple, pad_x: int, pad_y: int)

This was added because the process of fetching scaled coordinates for a given node was getting tedious and clumsy.

Essentially it only simply retrieves the original coordinate of node from coords and then calls scale_coords with those values

- node – The node name whose coordinates we want, which are nothing but keys from the dictionary coords which contains all coordinates of all nodes
- original_shape, new_shape, pad_x, pad_y – Same as described for scale_coords above

This function basically returns the scaled+padded coordinates for that node.

draw_legend(display_map: np.ndarray, target_size: tuple)

- display_map – This is an image array (as loaded by OpenCV, typically a NumPy array of shape (Height, Width, 3 colour channels)) on which we draw
- target_size – The intended display size of the window as a tuple (width, height)

This returns nothing and draws the legend in-place.

It works so by picking a position near the top-right of the image and drawing a white filled rectangle as a background for the legend and then draws line and circle markers with accompanying text labels

- A short yellow line with label "Explored path".
- A yellow dot with label "Explored node".
- A short red line with label "Shortest path".
- A green dot with label "Start point".
- A blue dot with label "End point".

It uses OpenCV drawing functions: cv2.line, cv2.circle, cv2.putText to draw these

These functions use yellow (0,255,255), red (0,0,255), green (0,255,0), blue (255,0,0) in BGR format.
It is called in visualize_path after showing the initial map

draw_summary_box(display_map: np.ndarray, start: str, end: str, total_distance: float, shortest_path: list, target_size: tuple)

This draws the summary information (start, end, distance, nodes crossed) at the end of the run

- display_map – The image array to draw on.
- start, end – The start and end node identifiers
- total_distance – The numeric distance of the shortest path. In code, it's a float
- shortest_path – The list of node identifiers representing the shortest route used to calculate how many intersections crossed
- target_size – The overall (W, H) of the display area, used to position the box.

Again, this function doesn't return anything, but only draws the summary box as in-place

This function does so by preparing a small text box, typically near bottom left, then uses cv2.getTextSize to get the longest line size and then use it as a baseline to determine maximum space needed for the summary box, and it then writes each line of text inside it (with again some padding),

It is used at the very end of visualize_path, after drawing the shortest path, to summarize the results

resize_with_padding (img: np. ndarray, target_size: tuple)

Rescale an image to fit into target_size without distortion adding padding to as calculated before.

- img – The original image array (the campus map) loaded via OpenCV
- target_size – The desired (width, height) for display. The code by default is set to 720p i.e., (1280, 720)
- padded_img – The resized image with white padding to exactly match target_size.
- new_shape – A tuple (new_height, new_width) of the resized image (excluding padding)

  * Shape often refers to image dimensions, in terms array rows (image width) and columns (image height)
- pad_x – Horizontal padding (pixels on left side).
- pad_y – Vertical padding (pixels on top side)

  The function calculates the scaling factor to use for resizing using the minimum of the scales between rows and columns, usually columns mostly of the time. Computes new dimensions: `new_w = original_w * scale`, `new_h = original_h * scale` and then uses `cv2.resize` to get the resized image

  It then proceeds to figure out padding needed, copies the resized image on a white filled background (acts as padding), places it properly in the center. At the end of which it returns the new composite image and the parameters needed to map coordinates `(new_shape, pad_x, pad_y)`

`visualize_path (map_file_path: str, start: str, end: str, output_path: str = "visualized_path.png", target_size: tuple = (1280, 720))`

Main function to orchestrate the path visualization

- map_file_path – Path to the map image file. ("data/gitam_map.png" by default in the call from main)
- start, end – The start and end node identifiers
- output_path – Filename for saving the output image
- target_size – Desired window size, default to 1280x720 p

Returns nothing. It is in-place.

It executes by using `cv2.imread(map_file_path)` to load image into an array, taking the target size and then calls functions accordingly

First it calls `resize_with_padding (map_img, target_size)` which return

- `display_map` – the image that will be shown and drawn on
- `new_shape, pad_x, pad_y` – used for coordinate scaling
- Also gets `original_shape = map_img.shape[:2]` (slicing so that the original image's height, width is obtained)

Then it initializes windows with cv2.startWindowThread() (which is not necessary, but standard procedure taken from documentation, but all in all, it initialises the GUI thread).
After that, a named window is created cv2.namedWindow("Live Pathfinding", cv2.WINDOW_NORMAL) that can be resized manually.
Now that it is made as resizable, a default size is set for 720 p dimensions, with cv2.resizeWindow("Live Pathfinding", target_width, target_height) to set the window size exactly to target_size.

Then we show the base map image without anything else first using cv2.imshow("Live Pathfinding", display_map), wait 2 seconds using cv2.waitKey(2000) which is incorporated for any performance issues while loading up (as I have encountered sometimes).

Next step is to draw the legend by calling draw_legend (display_map, target_size), marking the start and stop points as green and blue respectively.

Then we run the Dijkstra algorithm by calling dijkstra (start, end, graph) from dijkstra.py, which returns all the exploration order we need, along with the shortest path route. We extract them (unpack them), and calculated total number of nodes to explore in the map, and starting the animation

The visual exploration begins by iterating through the explored_order list returned by the algorithm. For each node, a yellow dot is drawn at its coordinate, and yellow lines are drawn to its neighbours to represent edge exploration.

This entire operation is carried by drawing so every 700ms, between nodes (whose coordinates are takes from coords.py), marking each explored node covered with a counter to show a statistically how much exploration has been carried out in the map, which itself is overlayed using
["Explored", f"{explored_percent:.1f} % | {explored_count}/{total_map_nodes} nodes"]
And all of that, is done so, image wise,
because this isn't just a one-time rendering — it's animated.
To create that smooth, step-by-step progression:
The frame is refreshed every 700ms using cv2.imshow() and cv2.waitKey(700).
This delay makes the movement slow enough to observe but fast enough not to bore the viewer.

After everything above is finished, the only thing that remains is drawing the shortest path , and we do so in red.
It is very simply done by taking each consecutive pair of nodes in the shortest path list and drawing a red line between them (w.r.t coordinates in coords.py) ,

and show it on the "Live Pathfinding" window, but wait 120ms every time we draw , this creates a quick animation effect of the red path tracing itself.

To summarise the visualization process in the screen:

- Call draw_summary_box (display_map, start, end, total_distance, shortest_path, target_size) to put the info box on the map,
- Show the final image with all overlays (cv2.imshow). At which point, the window will have everything: legend, explored paths, shortest path, markers, summary.
- Save the image to output_path using cv2.imwrite. It prints a confirmation to the console / terminal
- Print a message to console: "Press any key in the window to exit", cause that's how cv2.waitKey() works, (on a side note, waitkey () also act as according to key board inputs, so the pathfinding visualization can be sped up using keyboard strokes).

- Finally, call cv2.waitKey(0) which waits indefinitely for a key press in the wind. When a key is pressed, it breaks out, closes the window with cv2.destroyAllWindows(), and the function ends

To summarise the usage of modules in this file,

1) OpenCV (cv2) [computer vision 2 library] -
    a. cv2.imread to load images,
    b. cv2.resize to scale images,
    c. cv2.imshow, cv2.namedWindow, cv2.waitKey, cv2.destroyAllWindows for displaying the GUI window and controlling timing
    d. cv2.line (for paths), cv2.circle (for nodes), cv2.rectangle (for legend/summary boxes), cv2.putText (for text labels). When Colors are BGR tuples.
    e. cv2.FONT_HERSHEY_SIMPLEX for font choice, cv2.LINE_AA for anti-aliased lines

Alternatives could be matplotlib or Pygame, but OpenCV was likely chosen for simplicity in drawing

2) NumPy as np – Used in resize_with_padding to create the padded image (np.full) and possibly for some minor array shapes. np.uint8 dtype is used for image arrays (0–255-pixel values) [it stands for NumPy Unsigned Integer 8-bit]
   NumPy is needed cause cv2.imread loads images in form of arrays.
3) coords – Imports the coords dict from coords.py
4) dijkstra – Imports the Dijkstra function to compute shortest paths, without this, we wouldn't have the route to draw.

5) graph – Imports the graph data structure. Passed into Dijkstra and also used within visualizer for drawing neighbour lines of explored nodes, and also so that visualizer knows what to look for

This function is only called from main.py

main.py (Entry / control point)

Serves as the main interface with the user. It prints a list of available destination codes (from nodes.locations), takes the user's chosen start and end, validates them, and starts the visualization process.

It is intentionally kept short and straightforward, delegating complex tasks to other modules, to promote modularization,
This separation means you could reuse the visualization logic in another context by calling visualize_path, for future scalability with other algorithms or other campus maps.

The main function:

1) **Display Options:** Prints a message and all available location names for the user. It does so by printing them by sorting, to present them alphabetically

   The list comes from nodes.locations, which is a list of strings like `"NTR", "VDC", "RBS", ....` The join results in a comma-separated list.

2) Uses .strip () to remove any white spaces entered by the user, accidentally or intentionally

3) **Validate Input:** Checks for –
   If start not in locations: prints an error message listing node is not present

   Similarly for end not in locations: error message

   If start == end: prints a message that they are the same

   If all checks pass, then it will proceed

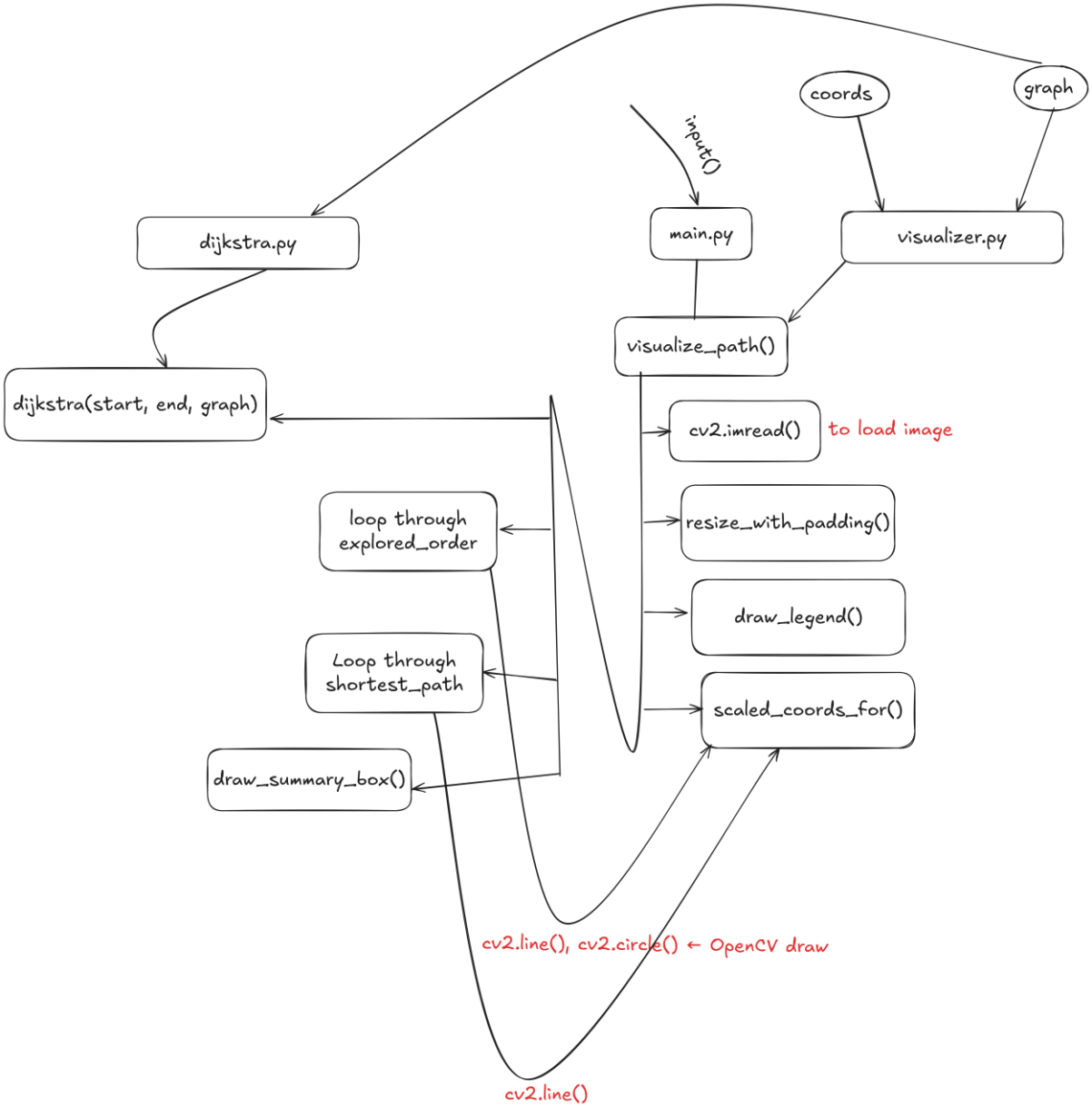
4) Computes the shortest path and asking the user to open the window. Then it calls:

visualize_path ("data/gitam_map.png", start, end)

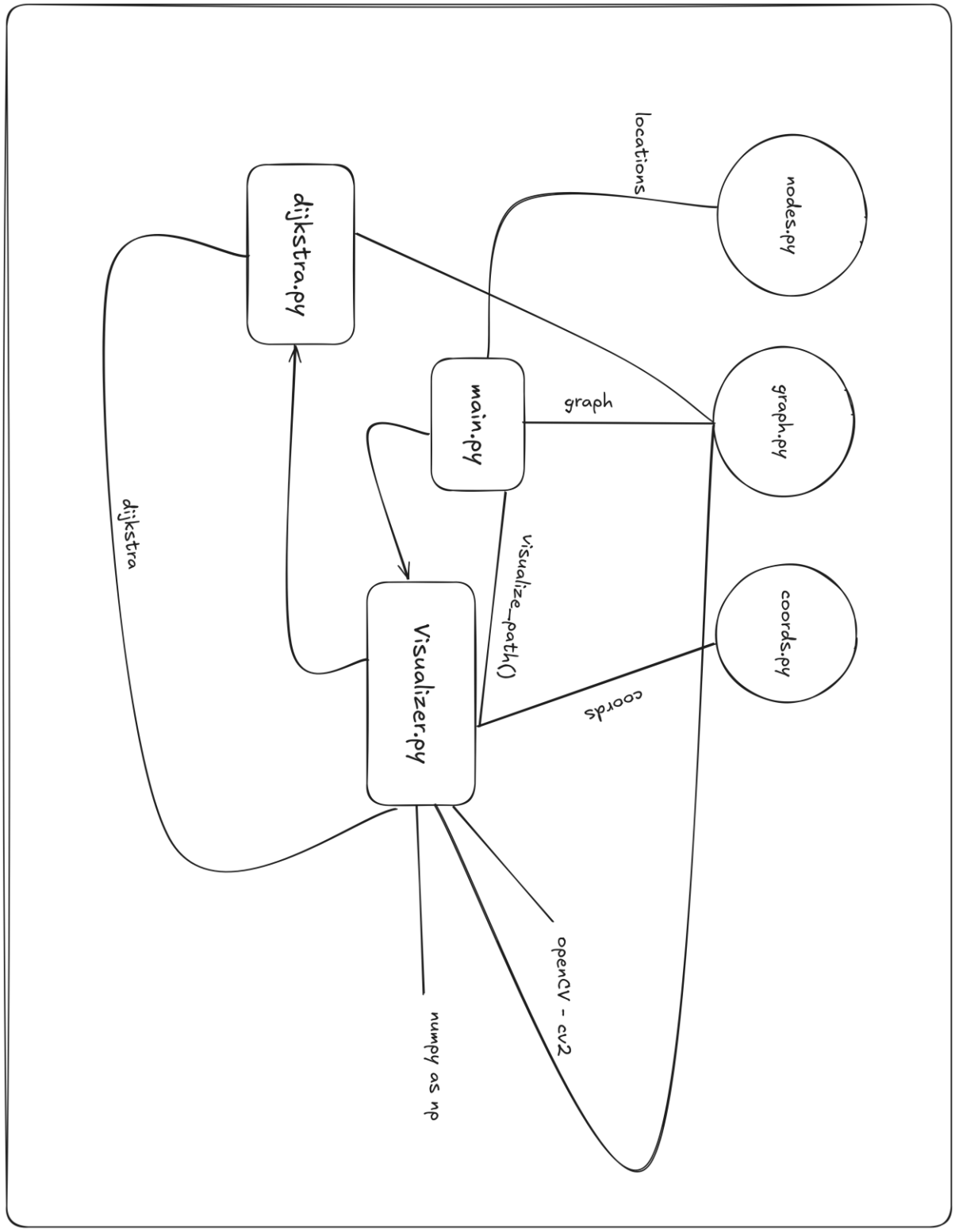
This file imports, nodes from locations, visualize_path from visualizer.

No other functions are defined in main.py. It's intentionally minimal

File execution Overview:



File Control Overview



Complexities of the Algorithm

The dijkstra() function finds the shortest path between two nodes in a weighted, directed graph using a greedy strategy.
It stores paths and distances as tuples in a queue, sorting the queue at each step because it needs to pick the lowest cost neighbour always.

Time complexity:

1. Main While Loop running
   Runs once per node extracted from the queue, and in worst case visits every edge (E) and node (N).
2. Queue needs Sorting
   Since it is sorted in every iteration, and sorting a list of size **n** has a time complexity of **O(n log n) ,**
   the queue very well can grow up to include all edges E and all nodes N, therefore a worst case sorting will be a time complexity of O(E log N).
3. And in absolute worst, the sorting itself needs to be performed for E edges,

Therefore, the combined time complexity would be $O(E^2 \log E)$.

This high time complexity can be remedied using a priority queue, lively implemented using heapq module. But that is an avenue to be explored for larger, much denser graphs.

Space Complexity:

1. For each visited node, it needs to be stored, so it would be a space complexity of O(N)
2. Since the algorithm has an explored_order list, that again would be another O(N)
3. The queue itself can have up to E entries, in a worst-case scenario. So that would be O(E).
4. And at last, path tracking is implemented by copying the entire path to each new node. Across all the entries into the queue, the total space used by these path copies can sum up to $\mathbf{O(V^2)}$ in the worst case.

The combined raw total would be O(2V) + O(E) + $O(V^2)$
Which totals to **$\mathbf{O\ (E + V^2)}$**

Execution Instructions

- Make sure all project files are in the same directory
- From a terminal or command prompt, navigate to the project directory and install the requirements `pip install -r requirements.txt`
- run the `main.py` script using Python
- Upon running, the program will display a prompt listing all available location codes and ask for user input.
- *"Enter START node:"* and *"Enter END node":* (The location codes are case-sensitive)
- Execution starts and you just open the "Live pathfinding" window that pops up in the taskbar. You'll have 2000ms to do so, before execution begins in the background.

Input Requirements: Start & End Location Codes

To use the Campus Pathfinding System, the user must input two valid location identifiers: a Start and an End location. These inputs determine the path for which the shortest route will be calculated and visualized.

Inputs are case-sensitive but any extra spaces are ignored — so entries like " krc " or "GsB" will not be accepted

Below is the complete **list** of accepted location codes and their full forms

AB	Architecture Bhavan
BG	Back Gate
CB	Corton Bhavan
CG	Cricket Ground
CP	Central Parking
CRL	Central Research Laboratory
CS	Coke Station
CVB	C V Raman Bhavan
DB	Dental Block
DDS	Durgabhai Deshmukh Sadan
FG	Front Gate
GB	Gitam Bhavan
GH	Girls Hostel
GP	Gandhi Park
GSB	Gitam School of Business
GSL	Gitam School of Law
GTC	Gitam Talent Café
ICT	ICT Bhavan
IEB	Industrial Engineering Bhavan
IS	Indoor Stadium

KRC	Knowledge Resource Center (Library)
MTP	Mother Theresa Park
NTR	NTR Garden near Back Gate
OA	Open Auditorium
PB	Pharmacy Bhavan
RBS	Rabindranath Sadan
SS	Sadarama Sadan
TC	Tennis Court
VB	Viswesaryya Bhavan
VBA	Viswesaryya Bhavan Annexure
VDC	Venture Development Centre
VS	Vinay Sadan

Dependencies

1) **OpenCV-Python** (`cv2`) – for image loading, drawing graphics
2) **NumPy** – for efficient image array manipulation

Error handling / Edge case handling

1) **Invalid Start/End Input:** In `main()`, after the user inputs
    a. If the entered start or end is not found in the locations list, the program prints an error message and exits. This prevents passing an invalid node into the pathfinding logic.
    b. This covers typos or any string that's not an accepted location code.

2) **Same Start and End:** If the user enters the same location for start and end
    a. Program prints a friendly message: `"Start and End nodes are the same. Which means you're already there!"`

3) **File Not Found (map image)**:
    a. After trying `cv2.imread`, program checks if `map_img` is `None`. If so, it raises a `FileNotFoundError` with a message including the path that failed, helps for debugging

4) **Unreachable Destination:** If no path exists
    a. dijkstra returns `distance = inf, path = []`
    b. The visualization would attempt to draw `shortest_path` which is empty list so the for loop on `shortest_path` would not execute, meaning no red line

Map Analysis:

This is a separate part of the project that deals with analysing how the campus layout is mapped or rather, how campus' layout is constructed,
how well arranged is it?

Statistics Generation for Analysis:

1) Main Program & Orchestration:
    a. main.py – Entry point that orchestrates the entire analytics process. It generates all pairwise paths and then runs all statistical reports in sequence, printing results to the console / terminal.

       Calls run_all_statistics() (imported as run_all from stats.run_all) to execute every analytics report. This, in turn, will call each get_summary() function in the stats submodule and print formatted results

       After that, it prints a completion message

2) Graph Data Definitions (static inputs from the previous part)
    a. nodes.py – Defines the node lists for the campus graph.
    b. graph.py – Defines the campus map graph as a dictionary of adjacency lists
    c. dijkstra.py – Implements Dijkstra's shortest path algorithm for the campus graph

3) Data Computation & I/O
    a. generate_all_paths.py – Core computation script that calculates all-pairs shortest paths among campus locations. It uses the Dijkstra algorithm to find the shortest route for every ordered pair of distinct locations and collects analytics data. The results are then exported to Python files in the data_io directory for use by the stats modules, they are done so using export_dict_to_py(filename, var_name, data_dict) – A utility function that takes a dictionary and writes it out to a file as a Python literal.

       It works by opening the file and writes a Python syntax representation of the dictionary. It writes the line var_name = { then each key-value pair on a new line and then closing using }

       and data_dict is the dictionary that is exported into a file

It uses itertools.permutations(locations, 2) to get every possible ordered pair of two distinct locations. (This means for every pair A, B, it will compute path A→B and B→A separately.) and for each of such pair, it calls dijkstra(start, end, graph) function (from data.dijkstra). The result is a dictionary with "path" and "distance" .

Then after,

i. Stores the path sequence in path_trace[(start, end)] = path
ii. Stores the distance in distance_matrix by setting distance_matrix[start][end] = distance
iii. Updates node_visit_count for intermediate nodes
iv. Updates edge_usage for each adjacent pair of nodes in the path

After computing all pairs, the function ensures the data_io directory exists (creating it if necessary) and then calls export_dict_to_py to write out four files

b. path_data.py –Contains path_trace, a dictionary mapping each pair of locations（start, end）to the list of nodes representing the shortest path between them. This is generated by generate_all_paths.py

c. distance_matrix.py –Contains `distance_matrix`, a dictionary of dictionaries that gives the shortest path distance from every start location to every other location, used for computing centrality metrics.

d. node_heatmap_data.py –node_visit_count, a dictionary counting how many times each node appears as an intermediate node in all shortest paths, this acts as a measure of node “betweenness” and feeds into the Bridge Node ranking.

e. edge_heatmap_data.py – Contains edge_usage, a dictionary counting usage of each edge (undirected path segment between two nodes) across all shortest paths. This feeds into the Most Used Segments ranking

Computing Statistics generated:

1) Centrality_rank.py
   This module computes the **Centrality & Accessibility Ranking** for campus locations. Its purpose is to determine which location is most "central" and the "least" central.
   It reads precomputed distances from distance_matrix.py and produces a structured summary. The output of its main function is used to display the centrality report in the console / terminal.

   It iterates through each entry in the imported distance_matrix, for each location, it computes the average distance to all other locations: avg_distance = total_distance_sum / count_of_destinations, these averages are stored in a dictionary avg_distances where keys are location names and values are the average distance. Next, it sorts this dictionary by the average distance, attaining an ascending order.

   Locations are ranked by average shortest distance to all others.
   Lower average = higher centrality.

   It then picks out the top 5 entries (most central) and bottom 5 entries (least central) from this sorted list.

   The final output may be interpreted as, the main result is the best central location. By also looking at the bottom of the list (highest averages), one can identify the least accessible location.

2) bridge_node_rank.py:
   This module computes the **Bridge Node Ranking**, identifying which nodes are most frequently used as intermediates on shortest paths. The goal is to find important connectors or hub nodes in the network.

   It reads the node_visit_count data and produces a summary of the top "bridge" nodes. The focus is on how often a particular node facilitates journeys between locations.

   It does it by taking the imported node_visit_count dictionary and sorts it by the count in descending order. This yields a list of (node, count) pairs from most used to least used. The node with the highest count is the **top bridge node.**

The function then takes the top 10 entries from this sorted list for reporting.

3) most_used_segments.py:
This module produces the ranking of **Most Frequently Used Edges** (segments) in the campus network. It examines which road segments are part of the largest number of shortest paths. Using the edge_usage data, it identifies the busiest connections on the map. The results show the top edges and can reveal if certain pathways are critical for many routes, or if they need much frequent maintenance.

The function takes the edge_usage dictionary imported from data and sorts it by the usage count in descending order. It then slices the sorted list to take the top 30 edges

And finally, the script that puts everything together, and executes them in a sequence, printing the results in a nicely formatted manner. It doesn't perform calculations itself; instead, it calls each of the get_summary functions from the stats modules and handles consistent formatting, you can say this is a central node for this project of a map!

It prints a header line with the title enclosed by emojis, it calls the corresponding get_data_function to obtain the summary dictionary for that analysis.

It is invoked by main.py (as run_all_statistics ()). It could also be run on its own in an interactive context to display results.

Wrapping It Up:

The project clearly prioritizes **clarity of visualization** over raw performance, intentionally slow down the algorithm via `waitKey (700)` and drawing each exploration step. Normally, Dijkstra would finish in a blink, but then you'd not see how it got there. By pausing and drawing, they allow the user to follow along.

Besides its not a huge map, with ~30 locations the total number of paths is manageable (30*29 = 870 pairs, each Dijkstra on a small graph).

And the same can be said about analysis metrics, since it required running a lot of background process before generating the result,
Hardware wouldn't bottleneck just yet because of the size of the map,
if this is to be expanded further, it would need some minor tweaking in logic and some major tweaking in analysis metrics,
It requires algorithmic thinking rather than simple hard coding.

I expect that bigger maps, say to map a whole town or even a city, would require a change in both approach and execution, as the map formation was done manually.

The code in the entirety of the project is mostly plain Python loops and dictionary operations, which was easy for a new programmer like myself, but I must admit facing multiple challenges with file_handling and with opencv2 methods. But should readability be ever to become an issue, one could optimize that into smaller files, maybe an approach to modularising function into multiple files, which in itself may make the visualizer file into 5 parts. But yeah, scalability and readability are two important things for **Sustainable Programming**.

Has been a fun learning journey

And the entire project is available at my GitHub profile.

https://github.com/VarunMatta5625/

Any queries or even better suggestions to make it much better? always happy to improve, do reach out!

Campus Navigation: Pathfinding and Mapping using Dijkstra

Map: