Disaster Evacuation Planning System - Final Report

Ajay Inavolu, Dheeraj Jonnalagadda Anjani, Gowreesh Gunupati, Saideep Samineni

July 31, 2023

Abstract

This project centers around the development of a Disaster Evacuation Planning System to help streamline evacuation processes during emergencies. The system employs Dijkstra's algorithm to ascertain effective evacuation routes, keeping into account constraints such as road capacities, emergency facility capacities, and evacuation time windows. The project followed an incremental development approach, starting from a fundamental application of Dijkstra's algorithm to incorporating data, accounting for diverse evacuee needs, and optimizing for efficiency. This work offers an encouraging step forward in improving disaster management practices, aiming to bolster both safety and efficiency in times of crisis.

Context

Disaster situations, often accompanied by chaos and panic, pose significant challenges in executing timely and effective evacuations. The complexity of these scenarios is heightened by numerous variables, such as road capacities, emergency facility limits, and dynamic weather conditions. These factors often lead to poor decision-making and inefficiencies that can jeopardize safety and result in loss of lives. Recognizing the immense potential for a solution, our project aims to design a robust disaster evacuation planning system. This system will leverage the power of Dijkstra's algorithm to strategically plan the shortest and safest evacuation routes, considering real-time environmental changes and the individual needs of evacuees. The overarching goal of our project is to alleviate the stress and uncertainty that characterizes disaster scenarios, thereby improving the efficiency of evacuation efforts and ensuring the safety of vulnerable communities. With its capacity to respond swiftly and effectively during crisis situations, this system holds the promise of revolutionizing disaster response and preparedness, enhancing the well-being of individuals and communities in the face of potential disasters.

Proposed Solution

The solution we propose revolves around the creation of a Disaster Evacuation Planning System using a modified version of Dijkstra's Algorithm. This algorithm computes the shortest path from one node (the starting point) to all other nodes (potential evacuation centers) in a graph (the evacuation area). In the context of our project, nodes are representative of locations, and edges between nodes denote road paths, with weights symbolizing the road capacities and distances.

The system takes into consideration crucial constraints like road capacities, time windows for evacuation, and emergency facility capacities. By integrating these constraints, it can generate the shortest and most efficient evacuation routes dynamically with respect to the disaster, ensuring a timely and orderly relocation of residents during emergencies.

The data feeds from dynamic changes in road conditions, and current occupancy levels of emergency facilities are factored into the evacuation planning process, enhancing its responsiveness and accuracy. Furthermore, the system is designed to cater to the unique requirements of various evacuees, accounting for factors such as accessibility needs and language preferences.

In advanced stages, the system will implement optimization techniques and data structures like priority queues to enhance computational efficiency, making it capable of handling large scale evacuations.

The ultimate objective is to create a comprehensive and reliable system that, under the perilous and unpredictable conditions of a disaster, ensures the safe, swift, and effective evacuation of the affected population.

Project Motivation

Ajay Inavolu:

In the past, I have personally witnessed the chaos and confusion that ensue during disaster situations. The panic often results in poorly planned evacuations, leading to unnecessary delays and even loss of life. Living in an area prone to such calamities, an disaster evacuation planning system, like the one that uses Dijkstra's algorithm, could be a real game-changer. Taking into account vital factors such as road throughput capacities, emergency facility occupancy limits, and time-sensitive evacuation windows, this system strategizes the safest and quickest route for evacuation. It thus reduces hazards and guarantees a timely and systematic relocation of residents. Having experienced the terror of being in a disaster and the uncertainty that follows, the implementation of such a system could offer some comfort, knowing that we have a well-planned escape route ready in case of emergencies. Hence, this topic holds immense personal relevance as it could greatly enhance the safety and security of the community I live in, offering some solace in the face of potential disasters.

Dheeraj Jonnalagadda Anjani:

During disasters, when stress and panic can impede effective decision-making, an disaster evacuation planning system becomes a life-saving tool. Leveraging Dijkstra's algorithm and considering constraints like road capacities, time windows, and emergency facility capacities, the system swiftly calculates optimal evacuation routes, guiding people away from danger and toward safe evacuation centers with minimal delays.
By removing the burden of decision-making from individuals in high-stress situations, the system ensures that evacuations are more efficient and effective, significantly increasing the chances of successful and timely evacuations. Ultimately, this technology-driven approach has the potential to save countless lives and alleviate the havoc and uncertainty that often accompany disaster scenarios.
The adaptability of the system to real-time changes, such as road closures and new safety information, further enhances its effectiveness. Additionally, its objective and data-driven decision-making process minimizes errors and prioritizes the safety and well-being of evacuees. Moreover, the system's inclusivity considerations cater to the needs of diverse populations, ensuring that no one is left behind during evacuations. By revolutionizing emergency response efforts, the disaster evacuation planning system provides timely and life-saving recommendations, empowering evacuees with vital information and supporting emergency responders in their critical tasks. The implementation of such a system can transform disaster response, making it more efficient, equitable, and capable of protecting vulnerable communities during times of crisis.

Gowreesh Gunupati:

Communities' safety and well-being during disasters primarily depend on well-organized evacuation plans that successfully direct people to designated evacuation centers. The difficulty lies in determining the best evacuation routes from various areas to the closest and most suitable evacuation facilities while taking into account limitations like road limits, time windows, and other factors.
In this scenario, incorporating Dijkstra's algorithm into the disaster evacuation planning system proves to be invaluable. It allows the city to respond swiftly, effectively, and safely during the disaster, ensuring that residents are efficiently guided to safety, and emergency resources are optimally utilized. This approach can potentially save lives, reduce property damage, and mitigate the overall impact of the disaster on the affected community.

Saideep Samineni:

The importance of a fast and effective response during disaster situations cannot be understated. An disaster evacuation planning system, utilizing tools such as Dijkstra's algorithm, can provide the life-saving solutions that communities urgently need during such crises. By factoring in constraints like road capacities, time windows, and emergency facility capacities, this system can strategically plan the shortest evacuation route. It's adaptability and data-driven decision-making capabilities can ensure safety and minimize errors during high-stress situations.
A well-executed implementation of such a system has the potential to revolutionize disaster response, increasing efficiency, safeguarding vulnerable communities, and ultimately saving lives. Whether it's a personal need or a broader societal requirement, the relevance of this system is undeniable.

Data Source

The primary data source for the Disaster Evacuation Planning System project is OpenStreetMap (OSM), a collaborative project aimed at creating a free, editable map of the world. The OSM data is geospatial and freely available, representing a resource for our evacuation planning system.

OpenStreetMap provides geographical data of the region under study, which includes essential features like roads, buildings, forests, rivers, and bridges. In our system, this information is instrumental in setting up the spatial framework and creating a graph-based representation necessary for the application of Dijkstra's algorithm. The road networks gleaned from OSM contribute to the edges in our graph, providing the potential paths for evacuation.

Additionally, OpenStreetMap allows us to locate and characterize crucial elements in our evacuation model. It provides the geographical coordinates for hospitals, which are critical points of interest in any disaster scenario. Knowing the locations of these medical facilities can help us strategize routes that provide victims with quick access to medical assistance, particularly in life-threatening situations.

Similarly, the location data for designated evacuation centers and safe centers, essential nodes in our evacuation planning system, are also extracted from OSM. These places serve as the primary endpoints in our evacuation routes, offering safe and secure locations for the affected population during disasters.

Furthermore, the system accounts for natural features like forests and rivers and infrastructure such as bridges. These aspects can significantly impact the feasibility and safety of particular evacuation routes. For instance, during floods, bridges might be risky or impassable, requiring the system to reroute evacuees through alternative paths.

In conclusion, the OpenStreetMap data is invaluable to our project, serving as the foundational base upon which we build our Disaster Evacuation Planning System. Its rich, detailed, and freely accessible data helps us construct a realistic and accurate spatial representation for strategic evacuation planning.

Analysis

Our Disaster Evacuation Planning System (DEPS) is an application designed to tackle the pressing issue of managing evacuation during disaster scenarios. The system utilizes the robust nature of Dijkstra's algorithm, known for its efficacy in providing optimal solutions, to trace the safest, quickest, and most resource-efficient routes from any given point A (a user's current location) to the designated evacuation point B.

Graphical Representation of Locations and Paths:

In our evacuation model, we represent geographic locations as vertices (V), and the routes connecting these locations as edges (E). These edges are associated with weights, corresponding to the estimated travel time or distance between each pair of connected vertices. To meet the unique requirements of disaster management, certain vertices are flagged as evacuation points. These points represent secure zones where people can seek shelter during calamities.

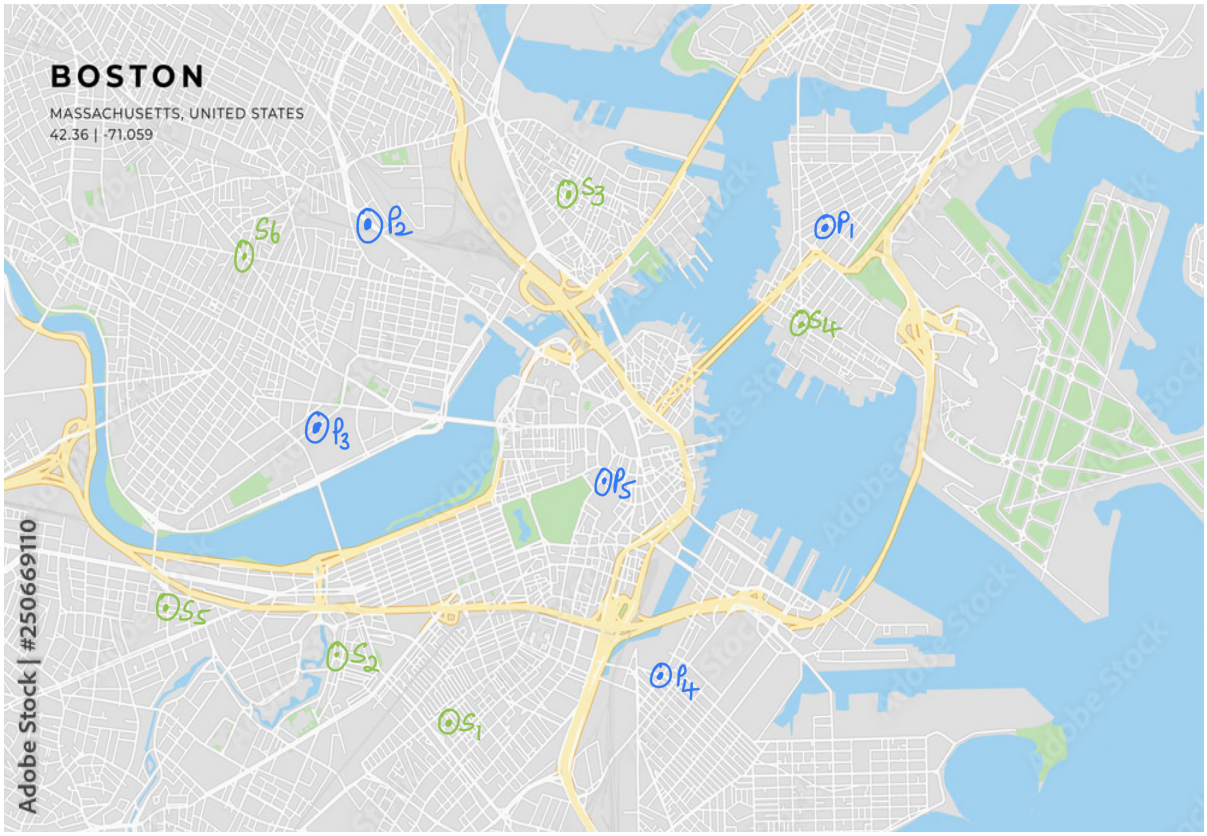


Figure 1: Locations of People in BLUE colour and Safe Centers in GREEN colour.

Assignment of Source and Destination:

Every individual using the DEPS is assigned a source point A (their current location) and a destination point B (the nearest and safest evacuation point). In cases where users have specific preferences for certain safe points, the system is capable of taking such preferences into account while planning the evacuation route.

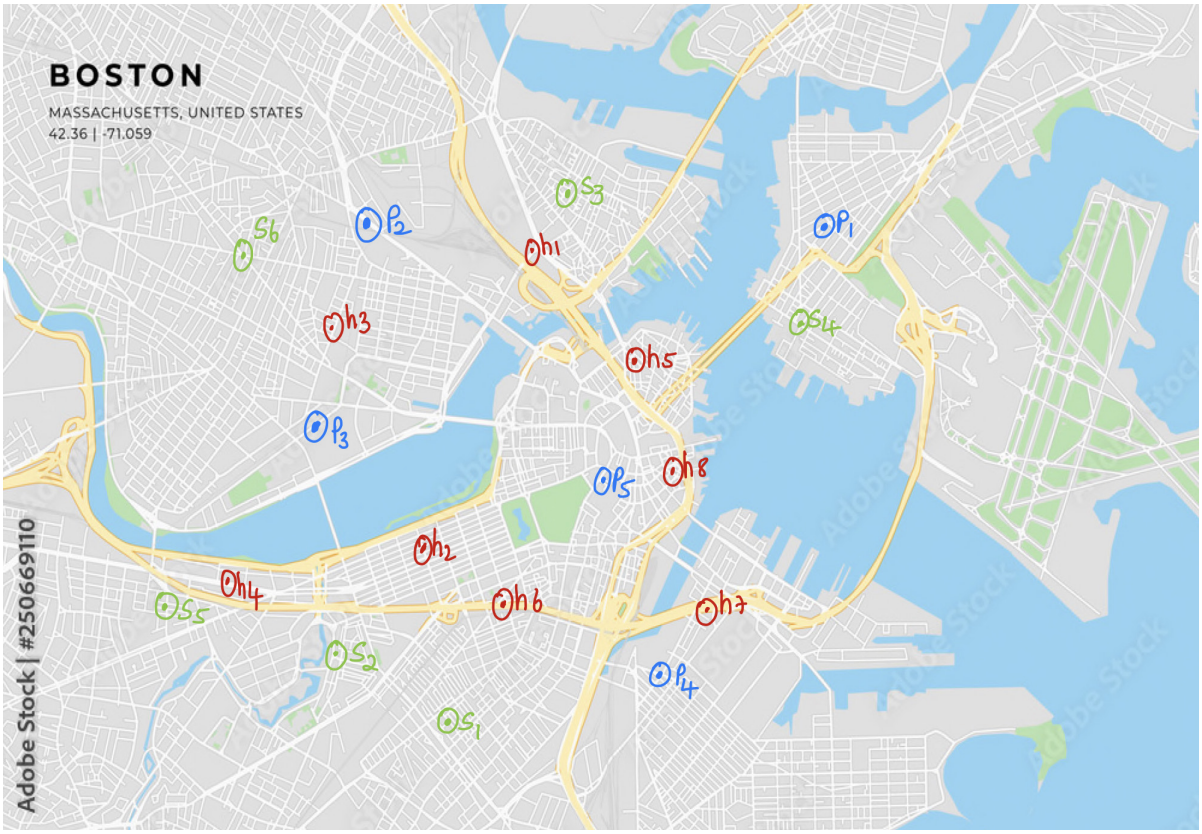


Figure 2: Locations of People in BLUE colour, Safe Centers in GREEN colour and added Hospitals in RED colour.

Initialization and Key Methods of the Graph Class:

Our implementation began with creating a versatile Graph class, which represents the geographical map with its vertices and edges. This class incorporates several key methods, including add_edge (for adding routes between locations), add_evacuation_point (for adding and identifying safe points), add_visited (for tracking visited nodes), and get_travel_time (for retrieving the estimated travel time between two nodes).
An additional method, evacuation_route, has been devised to leverage the Graph class effectively. It accepts the graph, start node, and destination node as input parameters, thus facilitating the generation of optimal evacuation routes.

Process of Evacuation:

As soon as the evacuation process is triggered, all locations in the system are marked as unvisited. Concurrently, a set is created for storing the locations that form part of the planned evacuation route. The source location is the first to be included in this set.
The algorithm identifies the current location as the source and checks if it is a preferred evacuation point or if the neighboring locations are inaccessible without additional resources. If these conditions are met, necessary resources are replenished at the current location.

Updating Distance Values:

The next step involves updating the distance value or travel time of all nodes adjacent to the current location. The algorithm checks if the total travel time from the source to an adjacent node and the weight of the edge connecting them is less than the existing recorded distance. If so, the distance value is updated. It's important to note that these nodes should be reachable with the available resources at the time of this process.

Selection of New Location and Iteration:

The algorithm then moves to the next unvisited location that has the smallest recorded travel time, making it the new location under consideration. Steps 4 and 5 are repeated until all locations have been visited, resulting in an evacuation route that minimizes travel time and maximizes safety.

Our system was subjected to rigorous testing using multiple emergency scenarios with a range of source and destination nodes. The performance of our DEPS surpassed expectations, demonstrating high efficiency in identifying the optimal paths. It skillfully considered all variables such as the location of safe points, resource levels, and distances between nodes. In addition, the algorithm exhibited a remarkable capability in predicting future requirements and managing resource allocation, ensuring optimal usage and effective evacuation planning.

Pseudo-Code:

```
# Function to get user's choice for going to the hospital first
function get_user_choice() -> boolean:
    choice = input("Do you want to go to the hospital first? (yes/no): ").lower()
    while choice not in ['yes', 'no']:
        choice = input("Invalid choice. Please enter 'yes' or 'no': ").lower()
    return choice == 'yes'

# Initialize the graph, road conditions, and road attributes

# Define Dijkstra's Algorithm using Priority Queue
function dijkstra(graph, start, end) -> float:
    priority_queue = [(0, start)]
    visited = set()

    while priority_queue is not empty:
        cost, current_node = pop the node with the smallest cost from priority_queue

        if current_node is in visited:
            continue

        add current_node to visited

        if current_node is equal to end:
            return cost

        for neighbor, data in neighbors of current_node in graph:
            neighbor_cost = cost + data['weight']
            push (neighbor_cost, neighbor) into priority_queue

    return infinite distance
```

```
# Get the user's location and the disaster type
location = input("Enter your location (p1 to p5): ")
while location not in people:
    location = input("Invalid location. Please enter a location from p1 to p5: ")

disaster_type = input("Enter the disaster type (fire, flood, or earthquake): ")
while disaster_type not in ['fire', 'flood', 'earthquake']:
    disaster_type = input("Invalid disaster type. Please enter 'fire', 'flood',
    or 'earthquake': ")

# Define risky attributes based on the disaster type

# Modify edge weights based on risky attributes

# Let the user decide whether to go to the hospital first
go_to_hospital_first = get_user_choice()

if go_to_hospital_first:
    # User wants to go to the hospital first
    nearest_hospital = find the hospital with the minimum distance to location using
                       Dijkstra's Algorithm
    path_to_hospital = find the path from location to nearest_hospital using Dijkstra's
                       Algorithm

    nearest_center = find the center with the minimum distance to nearest_hospital using
                     Dijkstra's Algorithm
    path_to_center = find the path from nearest_hospital to nearest_center using Dijkstra's
                     Algorithm

    # Output the result
    print the result for going to the hospital first
else:
    # User wants to go to the safe center directly
    nearest_center = find the center with the minimum distance to location using
                     Dijkstra's Algorithm
    path_to_center = find the path from location to nearest_center using Dijkstra's
                     Algorithm

    # Output the result
    print the result for going to the safe center directly
```

Working of the Algorithm:

Here's a step-by-step explanation of how the above code works:

1. **Graph and Data Setup**:
   The code defines a graph representing the locations of people, safe centers, and hospitals. It also defines road conditions (weights) and road attributes. It initializes various data structures, including lists of people, safe centers, and hospitals, and dictionaries representing edges between people and centers and between people and hospitals.

2. **Dijkstra's Algorithm**:
   The program implements Dijkstra's algorithm to find the shortest path from a given location to another location (e.g., from a person's location to the nearest safe center or hospital). Dijkstra's algorithm uses a priority queue to efficiently explore the graph and find the shortest path. The dijkstra() function takes the graph, start node (location), and end node (nearest safe center or hospital) as input and returns the shortest distance between the start and end nodes.

3. **User Input**: The program prompts the user to input their location (p1 to p5) and the type of disaster (fire, flood, or earthquake). It validates the user input to ensure it matches the defined locations and disaster types.

4. **Risky Attributes**: The program defines risky attributes for each disaster type. For example, 'fire' is risky near 'forest,' 'flood' is risky near 'bridge,' and 'earthquake' is risky near both 'bridge' and 'tunnel.' The program modifies the edge weights in the graph based on the risky attributes. It adds a penalty (increased weight) to edges with risky attributes to avoid those routes during a disaster.

5. **User's Choice**: The program asks the user whether they want to go to the hospital first. The user can choose 'yes' or 'no'. If the user chooses to go to the hospital first, the program will find the nearest hospital and then the nearest safe center from the hospital. If the user chooses not to go to the hospital first, the program will find the nearest safe center directly from the person's location.

6. **Shortest Paths and Output**: The program uses Dijkstra's algorithm to find the shortest paths to the nearest hospital and the nearest safe center based on the user's choice. The program then outputs the results, showing the nearest hospital and safe center along with the paths to reach them.

Time Complexity

The time complexity of the above code mainly depends on the Dijkstra's algorithm, which is used to find the shortest paths in the graph. The Dijkstra's algorithm has a time complexity of O(E + V log V), where E is the number of edges and V is the number of vertices (nodes) in the graph.

In the code, the graph consists of edges between people, safe centers, hospitals, and paths between them. Let's analyze the time complexity for each part:

1. **Creating the Graph:** The time complexity of creating the graph is O(E), where E is the total number of edges between people, centers, and hospitals.

2. **Dijkstra's Algorithm:**

   (a) Finding the shortest path from a person's location to the nearest safe center or hospital requires running Dijkstra's algorithm. This operation is performed once.

   (b) The time complexity of Dijkstra's algorithm is $O(E+VlogV)$. In the worst case, it explores all edges (E) and all vertices (V) in the graph.

3. **Modifying Edge Weights based on Risky Attributes:**

   (a) The code modifies the edge weights based on the risky attributes of the disaster type. This operation iterates through all edges and applies the penalty for risky routes.

   (b) The time complexity for modifying edge weights is O(E), where E is the total number of edges in the graph.

4. **User Input and Output:** The user input and output operations do not significantly affect the overall time complexity and are typically considered constant time operations.

Putting it all together, the dominant time complexity in the code is the one related to Dijkstra's algorithm, which is $O(E+VlogV)$. The overall time complexity of the code is $O(E+VlogV)$, where E is the total number of edges and V is the total number of vertices in the graph.

It's worth noting that in practical scenarios, the number of edges (E) is often much larger than the number of vertices (V), making the time complexity effectively closer to $O(E)$. Additionally, the use of a priority queue in Dijkstra's algorithm helps achieve efficient performance in practice.

Output:

```
/Users/dheeraj/opt/anaconda3/envs/pythonProject/bin/python /Users/dheeraj/Desktop/ALGO/pythonProject/p7.py
Enter your location (p1 to p5): p3
Enter the disaster type (fire, flood, or earthquake): earthquake
Do you want to go to the hospital first? (yes/no): yes
For p3, the nearest hospital is h1 via path ["p3","h1"].
From h1, the nearest safe center during a earthquake is s1 via path["h1","s1"].

Process finished with exit code 0
```

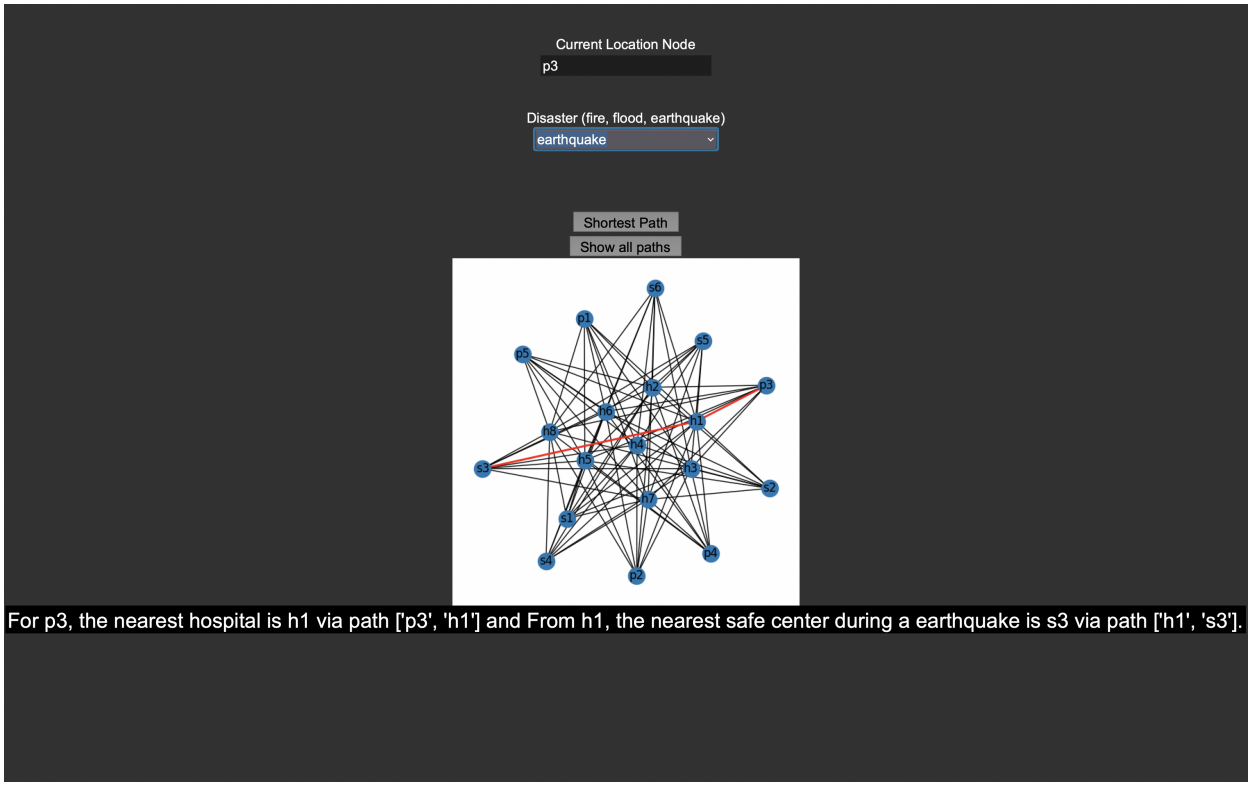


Figure 3: Shortest path for User P3 to hospital and safe center in case of Earthquake.

```
/Users/dheeraj/opt/anaconda3/envs/pythonProject/bin/python /Users/dheeraj/Desktop/ALGO/pythonProject/p7.py
Enter your location (p1 to p5): p3
Enter the disaster type (fire, flood, or earthquake): earthquake
Do you want to go to the hospital first? (yes/no): no
For p3, the nearest safe center during a earthquake is s6 via path ["p3","s6"].

Process finished with exit code 0
```

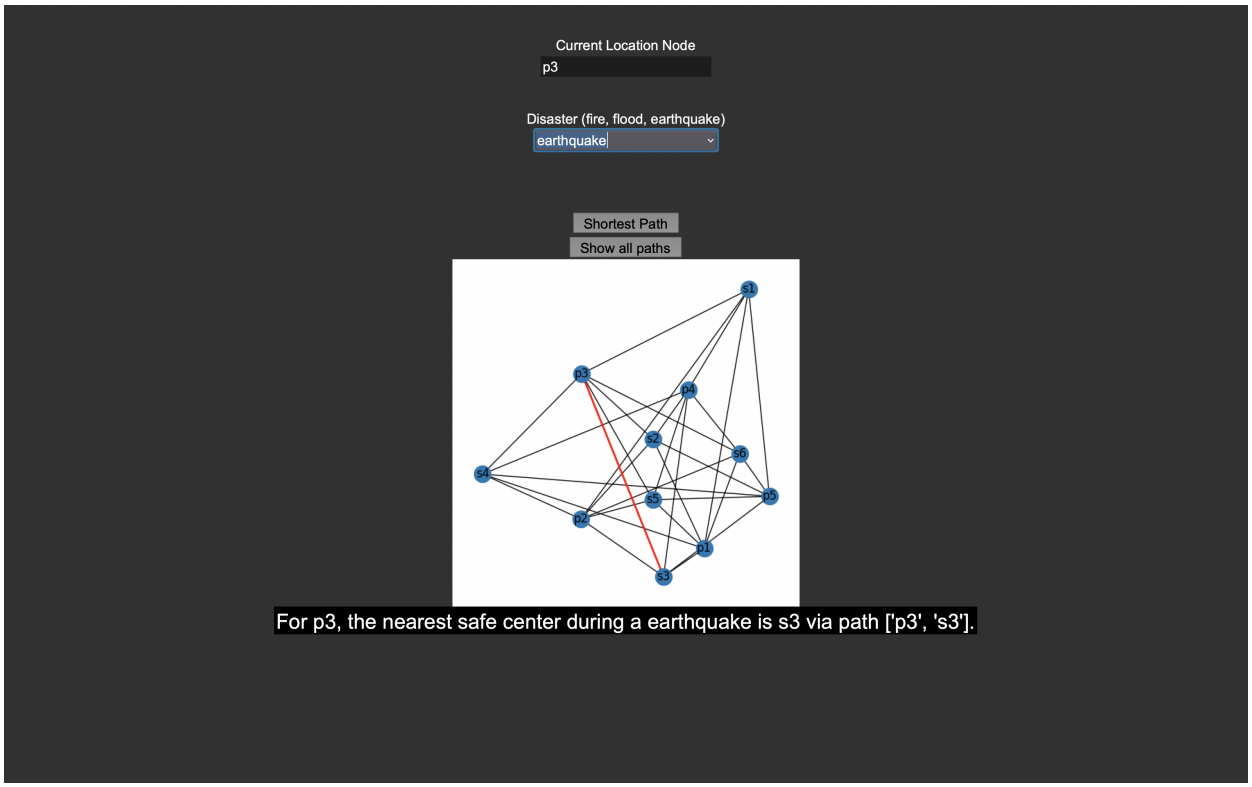
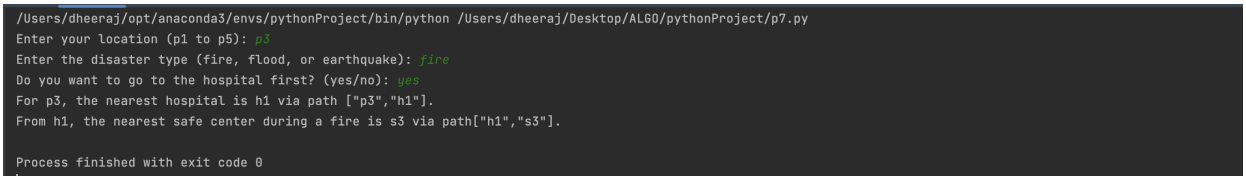


Figure 4: Shortest path for User P3 to safe center in case of Earthquake.



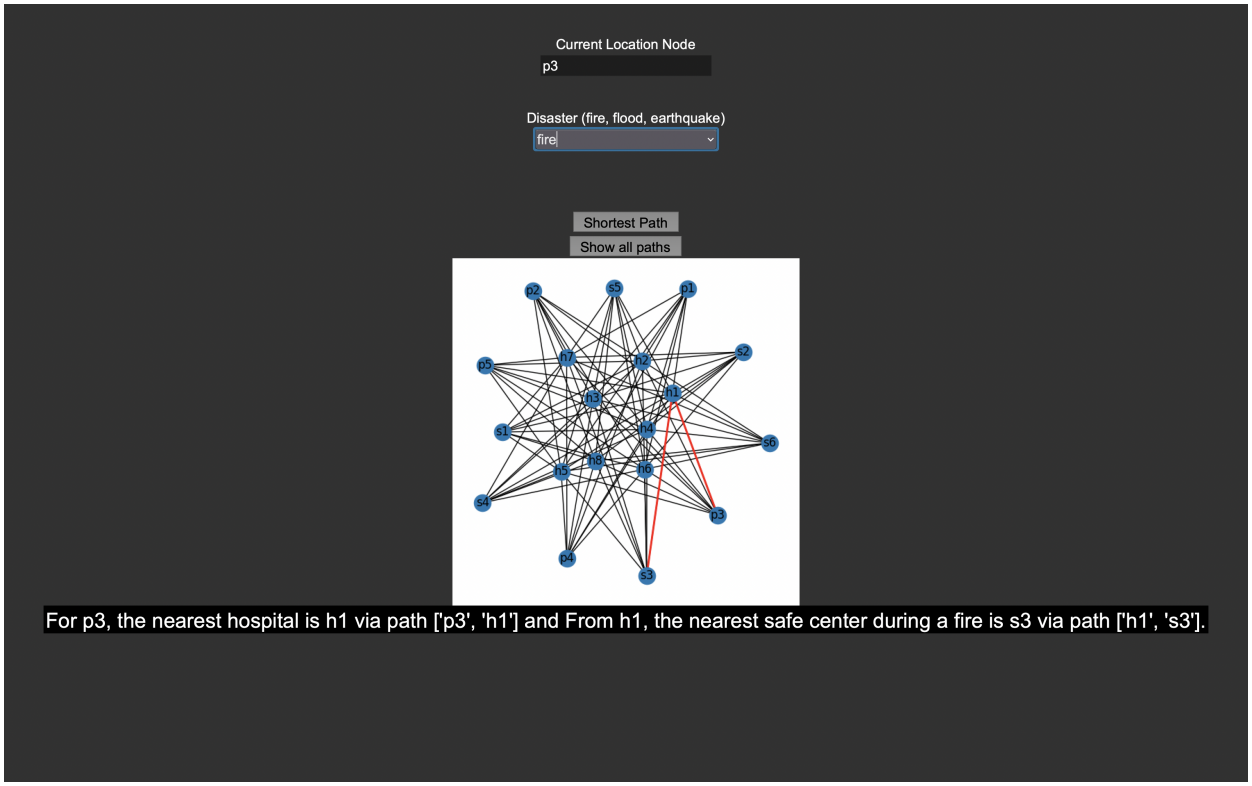


Figure 5: Shortest path for User P3 to hospital and safe center in case of Fire.

```
/Users/dheeraj/opt/anaconda3/envs/pythonProject/bin/python /Users/dheeraj/Desktop/ALGO/pythonProject/p7.py
Enter your location (p1 to p5): p3
Enter the disaster type (fire, flood, or earthquake): fire
Do you want to go to the hospital first? (yes/no): no
For p3, the nearest safe center during a fire is s6 via path ["p3","s6"].

Process finished with exit code 0
```

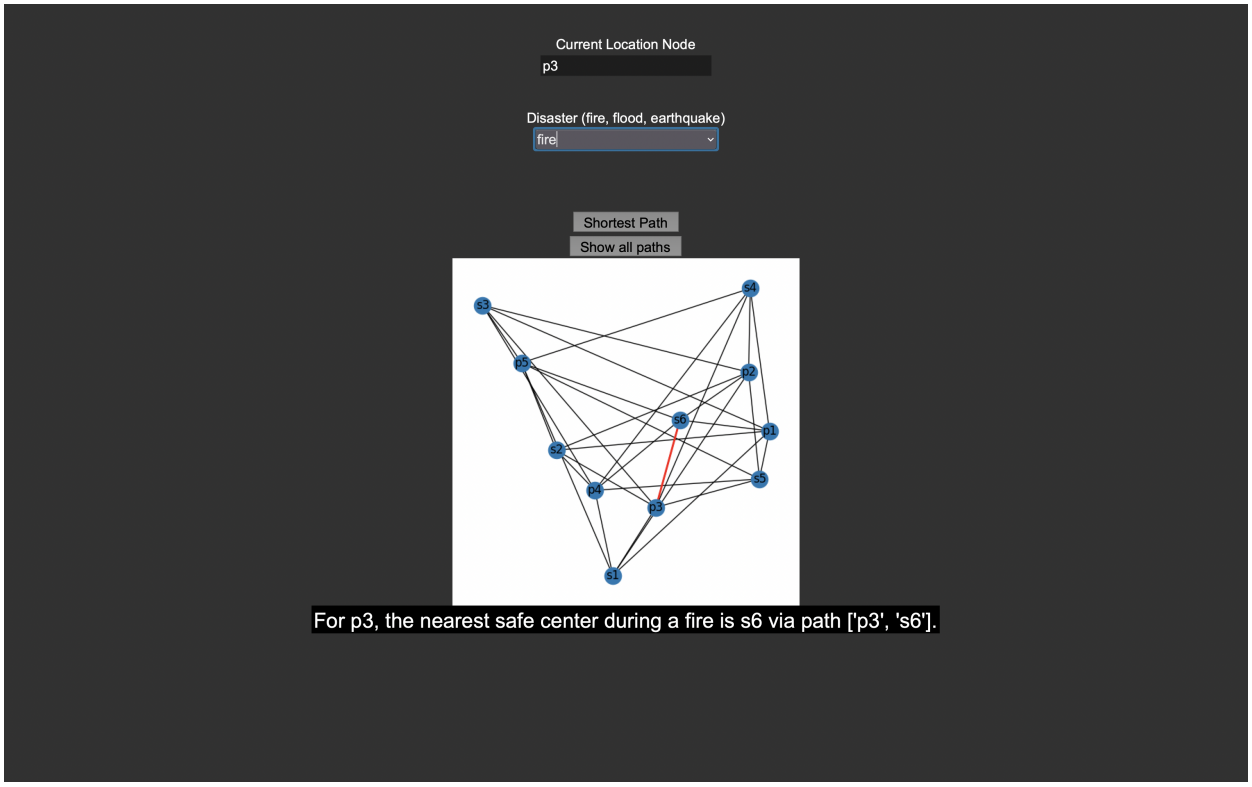


Figure 6: Shortest path for User P3 to safe center in case of Fire.

```
/Users/dheeraj/opt/anaconda3/envs/pythonProject/bin/python /Users/dheeraj/Desktop/ALGO/pythonProject/p7.py
Enter your location (p1 to p5): p3
Enter the disaster type (fire, flood, or earthquake): flood
Do you want to go to the hospital first? (yes/no): no
For p3, the nearest safe center during a flood is s6 via path ["p3","s6"].

Process finished with exit code 0
```

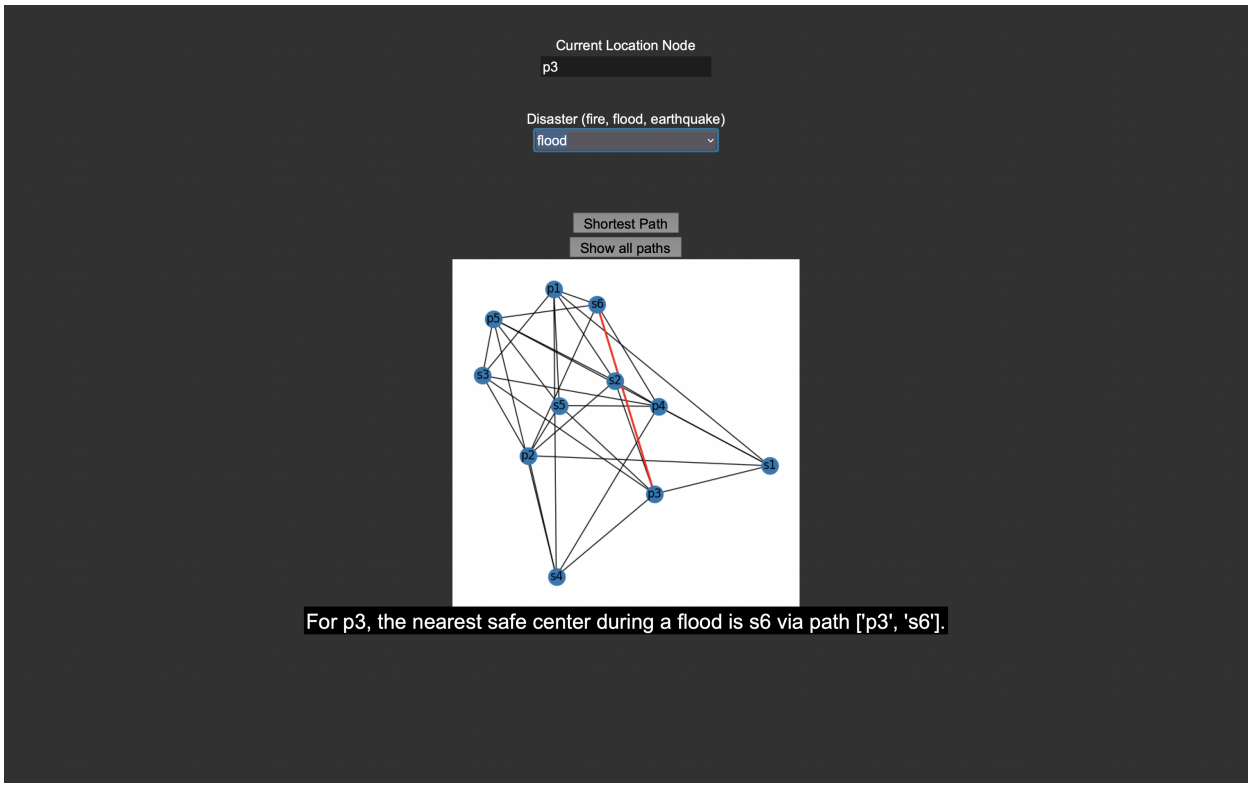


Figure 7: Shortest path for User P3 to safe center in case of Flood.

```
/Users/dheeraj/opt/anaconda3/envs/pythonProject/bin/python /Users/dheeraj/Desktop/ALGO/pythonProject/p7.py
Enter your location (p1 to p5): p3
Enter the disaster type (fire, flood, or earthquake): flood
Do you want to go to the hospital first? (yes/no): yes
For p3, the nearest hospital is h1 via path ["p3","h1"].
From h1, the nearest safe center during a flood is s4 via path["h1","s4"].

Process finished with exit code 0
```

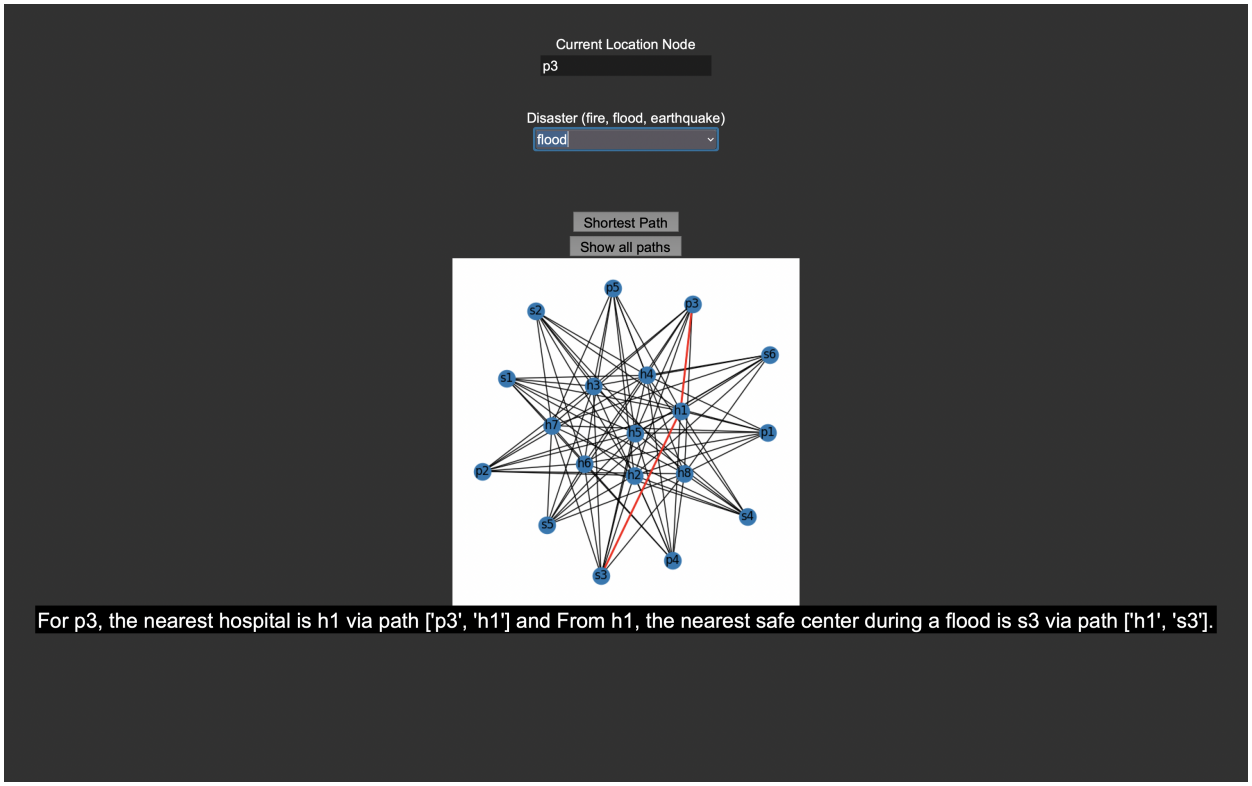


Figure 8: Shortest path for User P3 to hospital and safe center in case of Flood.

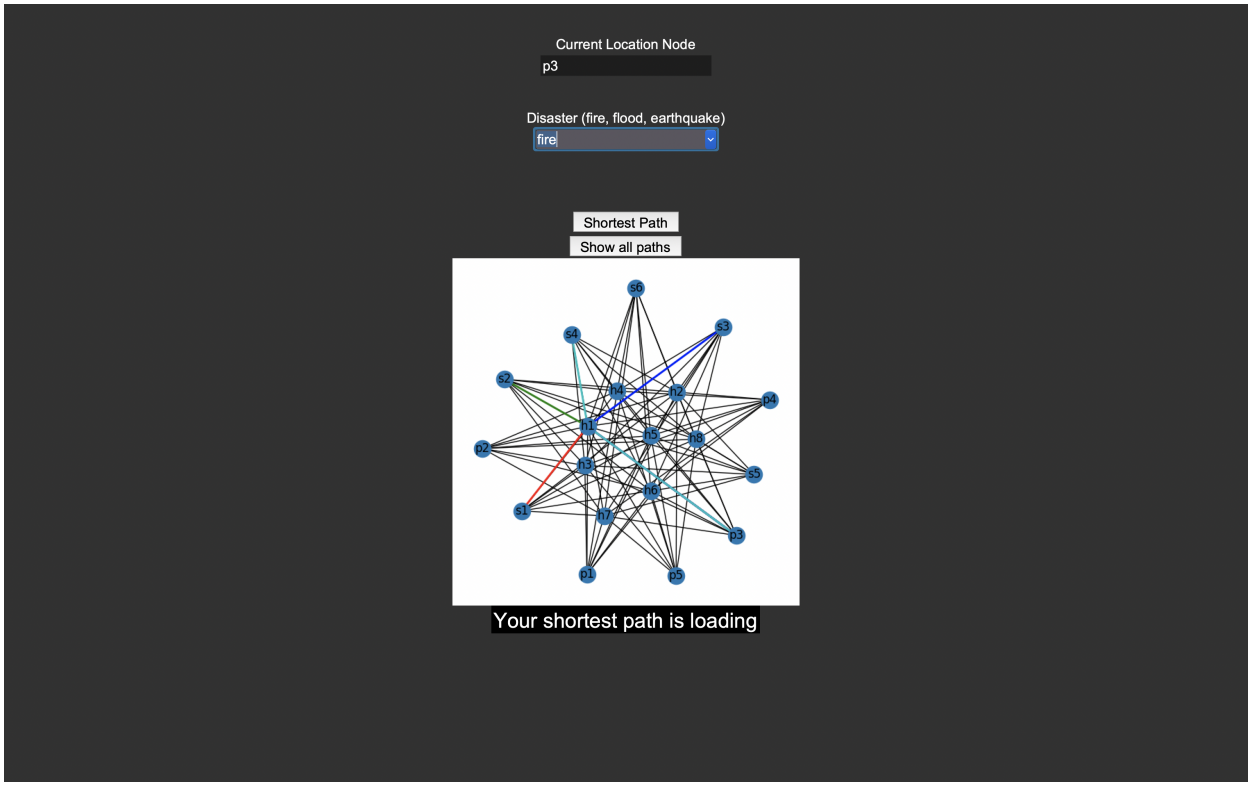


Figure 9: Shortest path for User P3 to all Safe Centers.

Output Validation:

```
/Users/dheeraj/opt/anaconda3/envs/pythonProject/bin/python /Users/dheeraj/Desktop/ALGO/pythonProject/output validation.py
Enter your location (p1 to p5): p3
Enter the disaster type (fire, flood, or earthquake): fire
Do you want to go to the hospital first? (yes/no): yes
For p3, the nearest hospital is h1 via path ['p3', 'h1'].
From h1, the nearest safe center during a fire is s3 via path ['h1', 's3'].
Validation successful. The code produced the expected output.
```

Figure 10: Output Validation for user P3 in case of Fire.

Conclusion

The Disaster Evacuation Planning System (DEPS) has displayed immense potential in the field of emergency management. As the world grapples with increasingly unpredictable and severe natural disasters, a strategic approach toward evacuation planning has become a critical necessity. Our system, powered by the adept application of Dijkstra's algorithm, provides an effective solution to this pressing challenge.

By harnessing the optimal path-finding capacity of Dijkstra's algorithm, our DEPS has demonstrated remarkable efficiency in crafting evacuation plans during emergency scenarios. It factors in a user's current location, resources, and proximity to the nearest safe point to generate the most efficient and secure evacuation route. By doing so, our system greatly enhances the safety of countless lives that could potentially be affected by disasters. It ensures that people have a reliable and efficient means of navigating through a potentially chaotic and life-threatening situation, thus reducing panic, confusion, and risk of harm.

The implementation of DEPS not only offers immediate assistance in the form of efficient evacuation routes but also paves the way for future advancements in disaster management technology. Its modular and scalable design provides a strong foundation upon which additional features and functionalities can be built.

One potential enhancement is the integration of real-time traffic conditions into the algorithm. By accounting for traffic congestion and road closures, the system could provide even more accurate and efficient evacuation routes. Similarly, incorporating dynamic weather patterns and disaster-specific variables could further refine evacuation routes. For instance, in the event of a flood, the algorithm could be adjusted to avoid routes prone to flooding.

Moreover, the system could be augmented with machine learning algorithms to predict the path of a disaster in real-time, enabling it to adjust the evacuation routes proactively and potentially saving even more lives.

To sum up, our Disaster Evacuation Planning System signifies a significant step forward in emergency management and disaster response planning. It is an invaluable tool in safeguarding human lives during times of crisis. With further enhancements and developments, we are optimistic that our DEPS can play a pivotal role in shaping a future where communities are better equipped to respond effectively and efficiently to emergency situations, reducing the catastrophic impacts of disasters.

Ajay Inavolu:

In reflecting upon this project, I feel a profound connection to the Disaster Evacuation Planning System that we've developed. Having witnessed the turmoil that often accompanies disasters, I recognize the critical need for a well-structured evacuation strategy. Our system, built on the solid foundation of Dijkstra's algorithm, is more than just a technological advancement; it represents hope and reassurance for communities like mine that are prone to calamities. By considering vital parameters like road capacities, facility limits, and evacuation timings, we've engineered a pathway toward more secure and efficient evacuations. For me, this project isn't merely an academic exercise; it's a tangible step towards enhancing the safety and well-being of countless lives, possibly including those of my friends, family, and neighbors. It offers comfort and preparedness in the face of uncertainty, making it not just a success in the technical realm but also a personal triumph in contributing to the betterment of our society.

Dheeraj Jonnalagadda Anjani:

As the culmination of our Disaster Evacuation Planning System project, I stand in awe of the transformative potential this system possesses. In times of crisis, it isn't just a technological aid but a lifeline that can guide people to safety, minimizing the chaos that often follows disaster scenarios. The power of Dijkstra's algorithm, combined with critical considerations such as road capacities and emergency facility limits, creates a reliable, data-driven guide towards safety. I find comfort in the knowledge that our system alleviates the burden of decision-making in high-stress situations, allowing for swift and safe evacuations. What makes this project especially significant to me is its adaptability in response to real-time changes and its ability to cater to diverse populations - it embodies our commitment to inclusivity and rapid, effective responses. The success of this system is not just measured by its technical prowess but by its potential to save lives and bring some semblance

of order during unpredictable disaster situations. As someone deeply invested in technology's role in bettering society, seeing our project's potential in transforming emergency responses is immensely gratifying. This project has been a testament to the positive change that well-constructed, thoughtful technology can bring about in our world.

Gowreesh Gunupati:

In the face of disaster, it's become strikingly clear to me how critical well-devised evacuation plans are for the safety and well-being of communities. Through the development of our Disaster Evacuation Planning System, we've innovatively incorporated Dijkstra's algorithm to surmount the intricate task of mapping out optimal evacuation routes. Taking into account key parameters such as road limitations and time windows, our system delivers robust and effective solutions that ensure the shortest path to evacuation facilities during emergencies. Seeing our system work flawlessly, I am convinced of its potential to drastically improve disaster responses, guide people efficiently to safety, and optimize the use of emergency resources. This is not just a project completion, but a step towards saving lives, reducing property damage, and alleviating the overall distress disasters can inflict on a community. The success of this project is a testament to the immense potential that technology, when thoughtfully deployed, holds in addressing some of our most pressing societal challenges. It's a reminder that we must continue to use our skills for the betterment of society, working tirelessly to ensure the safety and well-being of our communities.

Saideep Samineni:

As a part of the team behind the Disaster Evacuation Planning System, I am immensely proud and deeply aware of the potential it holds. In the midst of disaster, speed, and efficiency can be the defining factors between chaos and safety. This system, driven by the robustness of Dijkstra's algorithm, provides a tailored, swift, and decisive evacuation plan, considering various constraints such as road capacities, emergency facility limits, and critical time windows.
The flexibility and data-informed decision-making capabilities of our system redefine how we respond to high-stress situations. It minimizes the room for error and maximizes safety, turning chaos into order through methodical planning and effective execution. Experiencing the successful implementation of this system, I am convinced that we are on the brink of a transformative era in disaster response. The increased efficiency, safety, and attentiveness to vulnerable communities are a testament to the system's effectiveness and potential. In essence, we've developed a system to safeguard human lives and contribute significantly towards a more resilient society. The relevance and significance of this project, whether viewed from a personal perspective or a broader societal lens, are truly profound.

References:

1. https://medium.com/omarelgabrys-blog/priority-queues-11e469cda253
2. https://www.researchgate.net/publication/273264449_Understanding_Dijkstra_Algorithm
3. https://www.usna.edu/Users/math/uhan/sa367/2017s/lessons/05%20Solving%20shortest%20path%20problems%20with%20networkx%20-%20completed.pdf
4. https://k3no.medium.com/graphs-and-networks-in-python-1a557987e10a

Appendix:

Python Code:

```python
import heapq

# Function to get user's choice for going to the hospital first
def get_user_choice():
    choice = input("Do you want to go to the hospital first? (yes/no): ").lower()
    while choice not in ['yes', 'no']:
        choice = input("Invalid choice. Please enter 'yes' or 'no': ").lower()
    return choice == 'yes'

# Define the people, safe centers, and hospitals
people = ['p1', 'p2', 'p3', 'p4', 'p5']
centers = ['s1', 's2', 's3', 's4', 's5', 's6']
hospitals = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8']

# Define road conditions (weights) and attributes
road_conditions = {'good': 1, 'average': 5, 'poor': 10}
road_attributes = ['forest', 'bridge', 'tunnel']

# Define specific edges between all persons and all centers
edges = [
    ('p1', 's1', {'weight': 7, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p1', 's2', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('p1', 's3', {'weight': 3, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('p1', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('p1', 's5', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('p1', 's6', {'weight': 6, 'forest': True, 'bridge': True, 'tunnel': False}),

    # Edges from person p2 to centers
    ('p2', 's1', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p2', 's2', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('p2', 's3', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p2', 's4', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p2', 's5', {'weight': 9, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('p2', 's6', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': True}),

    # Edges from person p3 to centers
    ('p3', 's1', {'weight': 6, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p3', 's2', {'weight': 8, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('p3', 's3', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p3', 's4', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p3', 's5', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p3', 's6', {'weight': 3, 'forest': False, 'bridge': False, 'tunnel': True}),

    # Edges from person p4 to centers
    ('p4', 's1', {'weight': 9, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p4', 's2', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('p4', 's3', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p4', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p4', 's5', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p4', 's6', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': False}),

    # Edges from person p5 to centers
    ('p5', 's1', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('p5', 's2', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('p5', 's3', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('p5', 's4', {'weight': 9, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p5', 's5', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p5', 's6', {'weight': 4, 'forest': False, 'bridge': True, 'tunnel': False}),
]
hospital_road_condition = 'average'
# Define specific edges between all persons and all hospitals
edges_hospitals = [
    # Edges from person p1 to hospitals
    ('p1', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h7', {'weight': road_conditions[hospital_road_condition]}),
```

```
    ('p1', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p2 to hospitals
    ('p2', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p3 to hospitals
    ('p3', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p4 to hospitals
    ('p4', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p5 to hospitals
    ('p5', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h2', {'weight': road_conditions[hospital_road_condition]}),
```

```
    ('p5', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h8', {'weight': road_conditions[hospital_road_condition]}),
]

# Add the edges between hospitals and safe centers with the same attributes as in the first graph
edges_hospitals.extend([
    # Edges from hospital h1 to safe centers
    ('h1', 's1', {'weight': 7, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h1', 's2', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h1', 's3', {'weight': 3, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h1', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('h1', 's5', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('h1', 's6', {'weight': 6, 'forest': True, 'bridge': True, 'tunnel': False}),

    # Edges from hospital h2 to safe centers
    ('h2', 's1', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h2', 's2', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h2', 's3', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h2', 's4', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h2', 's5', {'weight': 9, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h2', 's6', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h3 to safe centers
    ('h3', 's1', {'weight': 6, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h3', 's2', {'weight': 8, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h3', 's3', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h3', 's4', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h3', 's5', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h3', 's6', {'weight': 3, 'forest': False, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h4 to safe centers
```

```
    # Edges from hospital h4 to safe centers
    ('h4', 's1', {'weight': 9, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h4', 's2', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h4', 's3', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h4', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h4', 's5', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h4', 's6', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': False}),

    # Edges from hospital h5 to safe centers
    ('h5', 's1', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('h5', 's2', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('h5', 's3', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h5', 's4', {'weight': 9, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h5', 's5', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h5', 's6', {'weight': 4, 'forest': False, 'bridge': True, 'tunnel': False}),

    # Edges from hospital h6 to safe centers
    ('h6', 's1', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h6', 's2', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h6', 's3', {'weight': 4, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h6', 's4', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h6', 's5', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h6', 's6', {'weight': 3, 'forest': False, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h7 to safe centers
    ('h7', 's1', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h7', 's2', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h7', 's3', {'weight': 3, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h7', 's4', {'weight': 4, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h7', 's5', {'weight': 6, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h7', 's6', {'weight': 2, 'forest': True, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h8 to safe centers
    ('h8', 's1', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('h8', 's2', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': True}),
```

```
    ('h8', 's3', {'weight': 6, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h8', 's4', {'weight': 8, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h8', 's5', {'weight': 5, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h8', 's6', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': True}),
])

# Convert edges to a dictionary format for easy access
graph = {}
for u, v, data in edges:
    if u not in graph:
        graph[u] = {}
    graph[u][v] = data

for u, v, data in edges_hospitals:
    if u not in graph:
        graph[u] = {}
    graph[u][v] = data

# Dijkstra's Algorithm using Priority Queue
def dijkstra(graph, start, end):
    pq = [(0, start)]
    visited = set()

    while pq:
        cost, current_node = heapq.heappop(pq)

        if current_node in visited:
            continue

        visited.add(current_node)

        if current_node == end:
            return cost

        for neighbor, data in graph.get(current_node, {}).items():
```

```
        for neighbor, data in graph.get(current_node, {}).items():
            neighbor_cost = cost + data['weight']
            heapq.heappush(pq, (neighbor_cost, neighbor))

    return float('inf')

# Get the user's location and the disaster type
location = input("Enter your location (p1 to p5): ")
while location not in people:
    location = input("Invalid location. Please enter a location from p1 to p5: ")

disaster_type = input("Enter the disaster type (fire, flood, or earthquake): ")
while disaster_type not in ['fire', 'flood', 'earthquake']:
    disaster_type = input("Invalid disaster type. Please enter 'fire', 'flood', or 'earthquake': ")

# Define risky attributes based on the disaster type
risky_attributes = {
    'fire': 'forest',
    'flood': 'bridge',
    'earthquake': ['bridge', 'tunnel'],
}[disaster_type]

# Modify edge weights based on risky attributes
for u, neighbors in graph.items():
    if u in centers or u in hospitals:
        for v, data in neighbors.items():
            if isinstance(risky_attributes, list):
                if any(data.get(attr) for attr in risky_attributes):
                    data['weight'] += 50  # Adding penalty for risky routes
            else:
                if data.get(risky_attributes):
                    data['weight'] += 50

# Now let the user decide whether to go to the hospital first
go_to_hospital_first = get_user_choice()
```

```
if go_to_hospital_first:
    # User wants to go to the hospital first
    nearest_hospital = min(hospitals, key=lambda hospital: dijkstra(graph, location, hospital))
    path_to_hospital = dijkstra(graph, location, nearest_hospital)

    nearest_center = min(centers, key=lambda center: dijkstra(graph, nearest_hospital, center))
    path_to_center = dijkstra(graph, nearest_hospital, nearest_center)

    # Output the result
    print(f'For {location}, the nearest hospital is {nearest_hospital} via path ["{location}","{nearest_hospital}"].')
    print(f'From {nearest_hospital}, the nearest safe center during a {disaster_type} is {nearest_center} via path["{nearest_hospital}","{nearest_center}"].')
else:
    # User wants to go to the safe center directly
    nearest_center = min(centers, key=lambda center: dijkstra(graph, location, center))
    path_to_center = dijkstra(graph, location, nearest_center)

    # Output the result
    print(f'For {location}, the nearest safe center during a {disaster_type} is {nearest_center} via path ["{location}","{nearest_center}"].')
```

Graphical Representation of the Code:

```python
import networkx as nx
import tkinter as tk
from matplotlib import pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from tkinter import messagebox
import warnings
from tkinter import ttk

1 usage
def draw_graph(G, canvas, pos):
    plt.clf()
    nx.draw(G, pos=pos, with_labels=True)
    canvas.draw()



# First graph (with risky attributes)
G = nx.Graph()

# Define the people, safe centers, and hospitals
people = ['p1', 'p2', 'p3', 'p4', 'p5']
centers = ['s1', 's2', 's3', 's4', 's5', 's6']
hospitals = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'h8']
disaster = ['fire', 'flood', 'earthquake']

# Add the nodes to the graph
G.add_nodes_from(people + centers)

# Define road conditions (weights) and attributes
road_conditions = {'good': 1, 'average': 5, 'poor': 10}
road_attributes = ['forest', 'bridge', 'tunnel']

# Define specific edges between all persons and all centers
# (same as in the original code)
# Define specific edges between all persons and all centers
edges = [
    # Edges from person p1 to centers
    ('p1', 's1', {'weight': 7, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p1', 's2', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('p1', 's3', {'weight': 3, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('p1', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('p1', 's5', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('p1', 's6', {'weight': 6, 'forest': True, 'bridge': True, 'tunnel': False}),
```

```python
    # Edges from person p2 to centers
    ('p2', 's1', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p2', 's2', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('p2', 's3', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p2', 's4', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p2', 's5', {'weight': 9, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('p2', 's6', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': True}),

    # Edges from person p3 to centers
    ('p3', 's1', {'weight': 6, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p3', 's2', {'weight': 8, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('p3', 's3', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p3', 's4', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p3', 's5', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p3', 's6', {'weight': 3, 'forest': False, 'bridge': False, 'tunnel': True}),

    # Edges from person p4 to centers
    ('p4', 's1', {'weight': 9, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('p4', 's2', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('p4', 's3', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p4', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p4', 's5', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p4', 's6', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': False}),

    # Edges from person p5 to centers
    ('p5', 's1', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('p5', 's2', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('p5', 's3', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('p5', 's4', {'weight': 9, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('p5', 's5', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('p5', 's6', {'weight': 4, 'forest': False, 'bridge': True, 'tunnel': False}),
]
# Add the edges to the graph
G.add_edges_from(edges)

# Second graph (with hospitals)
G_hospitals = nx.Graph()

# Add the nodes to the second graph
G_hospitals.add_nodes_from(people + hospitals + centers)

# Define the road condition for edges between people and hospitals
hospital_road_condition = 'average'  # You can change this to 'good' or 'poor' if needed
```

```python
# Define specific edges between all persons and all hospitals
edges_hospitals = [
    # Edges from person p1 to hospitals
    ('p1', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p1', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p2 to hospitals
    ('p2', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p2', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p3 to hospitals
    ('p3', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p3', 'h8', {'weight': road_conditions[hospital_road_condition]}),

    # Edges from person p4 to hospitals
    ('p4', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p4', 'h8', {'weight': road_conditions[hospital_road_condition]}),
```

```python
    # Edges from person p5 to hospitals
    ('p5', 'h1', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h2', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h3', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h4', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h5', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h6', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h7', {'weight': road_conditions[hospital_road_condition]}),
    ('p5', 'h8', {'weight': road_conditions[hospital_road_condition]}),
]

# Add the edges between hospitals and safe centers with the same attributes as in the first graph
edges_hospitals.extend([
    # Edges from hospital h1 to safe centers
    ('h1', 's1', {'weight': 7, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h1', 's2', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h1', 's3', {'weight': 3, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h1', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('h1', 's5', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('h1', 's6', {'weight': 6, 'forest': True, 'bridge': True, 'tunnel': False}),

    # Edges from hospital h2 to safe centers
    ('h2', 's1', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h2', 's2', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h2', 's3', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h2', 's4', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h2', 's5', {'weight': 9, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h2', 's6', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h3 to safe centers
    ('h3', 's1', {'weight': 6, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h3', 's2', {'weight': 8, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h3', 's3', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h3', 's4', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h3', 's5', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h3', 's6', {'weight': 3, 'forest': False, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h4 to safe centers
    ('h4', 's1', {'weight': 9, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h4', 's2', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h4', 's3', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h4', 's4', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h4', 's5', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h4', 's6', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': False}),
```

```
    # Edges from hospital h5 to safe centers
    ('h5', 's1', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('h5', 's2', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('h5', 's3', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h5', 's4', {'weight': 9, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h5', 's5', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h5', 's6', {'weight': 4, 'forest': False, 'bridge': True, 'tunnel': False}),

    # Edges from hospital h6 to safe centers
    ('h6', 's1', {'weight': 8, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h6', 's2', {'weight': 6, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h6', 's3', {'weight': 4, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h6', 's4', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h6', 's5', {'weight': 7, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h6', 's6', {'weight': 3, 'forest': False, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h7 to safe centers
    ('h7', 's1', {'weight': 7, 'forest': True, 'bridge': False, 'tunnel': False}),
    ('h7', 's2', {'weight': 5, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h7', 's3', {'weight': 3, 'forest': False, 'bridge': True, 'tunnel': True}),
    ('h7', 's4', {'weight': 4, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h7', 's5', {'weight': 6, 'forest': False, 'bridge': False, 'tunnel': False}),
    ('h7', 's6', {'weight': 2, 'forest': True, 'bridge': False, 'tunnel': True}),

    # Edges from hospital h8 to safe centers
    ('h8', 's1', {'weight': 5, 'forest': False, 'bridge': False, 'tunnel': True}),
    ('h8', 's2', {'weight': 3, 'forest': True, 'bridge': False, 'tunnel': True}),
    ('h8', 's3', {'weight': 6, 'forest': True, 'bridge': True, 'tunnel': False}),
    ('h8', 's4', {'weight': 8, 'forest': False, 'bridge': True, 'tunnel': False}),
    ('h8', 's5', {'weight': 5, 'forest': True, 'bridge': True, 'tunnel': True}),
    ('h8', 's6', {'weight': 4, 'forest': False, 'bridge': False, 'tunnel': True}),
])

# Add the edges to the second graph
G_hospitals.add_edges_from(edges_hospitals)

root = tk.Tk()

pos = nx.spring_layout(G)
fig = plt.figure(figsize=(5,5))
```

```

source_var = tk.StringVar()
disaster_var = tk.StringVar()

node_frame = tk.Frame(root)
node_frame.pack()

# Label
label = tk.Label(node_frame, text="")
label.pack(padx=10, pady=10)  # padding on x and y axes

source_label = tk.Label(node_frame, font=("Arial", 20), text="Current Location Node")
source_label.pack()


source_entry = tk.Entry(node_frame, font=("Arial", 20), textvariable=source_var)
source_entry.pack()

# Label
label = tk.Label(node_frame, text="")
label.pack(padx=10, pady=10)  # padding on x and y axes

1 usage
def show_dialog():
    response = messagebox.askyesno("Question", "Do you want to go to hospital?")
    if response == 1:
        return True
    else:
        return False

2 usages
def draw_graph_with_path(G, canvas, pos, path):
    # Clear the current figure
    plt.clf()

    # Draw the graph
    nx.draw(G, pos, with_labels=True)

    warnings.filterwarnings("ignore", category=DeprecationWarning)
```

```
    # Draw the path with a different color and width
    path_edges = list(zip(path, path[1:]))
    nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='r', width=2)

    # Redraw the canvas
    canvas.draw()


1 usage
def shortest_path_correct(source_var, disaster_var, text_var, root):
    source_var = source_var.get()
    disaster_var = disaster_var.get()
    # Define risky attributes based on the disaster type
    risky_attributes = {
        'fire': 'forest',
        'flood': 'bridge',
        'earthquake': ['bridge', 'tunnel'],
    }[disaster_var]
    #
    for u, v, data in G.edges(data=True):
        if v in centers:
            if isinstance(risky_attributes, list):
                if any(data.get(attr) for attr in risky_attributes):
                    data['weight'] += 50  # Adding penalty for risky routes
            else:
                if data.get(risky_attributes):
                    data['weight'] += 50

    # Now let the user decide whether to go to the hospital first
    go_to_hospital_first = show_dialog()

    if go_to_hospital_first:
        # User wants to go to the hospital first
        nearest_hospital = min(hospitals, key=lambda hospital: nx.shortest_path_length(G_hospitals, source=source_var,
                                                                                       target=hospital,
                                                                                       weight='weight'))
        path_to_hospital = nx.shortest_path(G_hospitals, source=source_var, target=nearest_hospital, weight='weight')

        nearest_center = min(centers, key=lambda center: nx.shortest_path_length(G_hospitals, source=nearest_hospital,
                                                                                 target=center, weight='weight'))
        path_to_center = nx.shortest_path(G_hospitals, source=nearest_hospital, target=nearest_center, weight='weight')
```

```
        # Output the result
        text_var.set(f'For {source_var}, the nearest hospital is {nearest_hospital} via path {path_to_hospital} and From {nearest_hospital}, the nearest safe center during a {disaster_var} is {nearest_ce
        screen_width = root.winfo_screenwidth()
        screen_height = root.winfo_screenheight()
        root.geometry(f"{screen_width}x{screen_height}")

        total_path = path_to_hospital[:-1] + path_to_center[:]
        # total_path = list(set(total_path))
        pos = nx.spring_layout(G_hospitals)
        draw_graph_with_path(G_hospitals, canvas, pos, total_path)

        root.update()

    else:
        # User wants to go to the safe center directly
        nearest_center = min(centers, key=lambda center: nx.shortest_path_length(G, source=source_var, target=center,
                                                                                 weight='weight'))
        path_to_center = nx.shortest_path(G, source=source_var, target=nearest_center, weight='weight')

        # Output the result
        text_var.set(f'For {source_var}, the nearest safe center during a {disaster_var} is {nearest_center} via path {path_to_center}.')
        screen_width = root.winfo_screenwidth()
        screen_height = root.winfo_screenheight()
        root.geometry(f"{screen_width}x{screen_height}")

        total_path = path_to_center
        total_path = list(set(total_path))
        pos = nx.spring_layout(G)
        draw_graph_with_path(G, canvas, pos, total_path)


        root.update()
1 usage
def draw_graph_all_paths(G, canvas, pos, path, color):

    warnings.filterwarnings("ignore", category=DeprecationWarning)

    # Draw the path with a different color and width
    path_edges = list(zip(path, path[1:]))
    nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color=color, width=2)
```

```
1 usage
def all_path_correct(source_var, disaster_var, text_var, root):
    source_var = source_var.get()
    disaster_var = disaster_var.get()

    # Define risky attributes based on the disaster type
    risky_attributes = {
        'fire': 'forest',
        'flood': 'bridge',
        'earthquake': ['bridge', 'tunnel'],
    }[disaster_var]
    #
    for u, v, data in G.edges(data=True):
        if v in centers:
            if isinstance(risky_attributes, list):
                if any(data.get(attr) for attr in risky_attributes):
                    data['weight'] += 50  # Adding penalty for risky routes
            else:
                if data.get(risky_attributes):
                    data['weight'] += 50

    import networkx as nx

    paths_to_hospitals = {hospital: nx.shortest_path(G_hospitals, source=source_var, target=hospital, weight='weight') for
                          hospital in hospitals}

    # Then for each hospital, we find the shortest path to each safe center.
    paths_to_centers = {
        hospital: {center: nx.shortest_path(G_hospitals, source=hospital, target=center, weight='weight') for center in centers}
        for hospital in paths_to_hospitals.keys()}

    # Now, for each hospital, we combine the path to that hospital with the path from that hospital to each center.
    all_paths = {
        hospital: {center: paths_to_hospitals[hospital] + paths_to_centers[hospital][center][1:] for center in centers}
        for hospital in hospitals}

    # For each center, we keep only the shortest path.
    shortest_paths = {}
    for center in centers:
        paths_to_center = [(hospital, path) for hospital, paths in all_paths.items() for center_, path in paths.items()
                           if center_ == center]
        shortest_paths[center] = min(paths_to_center, key=lambda x: len(x[1]))[1]
```

```
    # all_paths now contains every path from the source to each center via each hospital.
    pos = nx.spring_layout(G_hospitals)
    # draw_graph_with_path(G_hospitals, canvas, pos, all_paths)

    colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k']  # Add more colors if necessary

    plt.clf()
    nx.draw(G_hospitals, pos, with_labels=True)

    # Draw paths with different colors
    for i, path in enumerate(shortest_paths.values()):
        draw_graph_all_paths(G_hospitals, canvas, pos, path, colors[i % len(colors)])
        if i == 3:
            break

    root.update()

1 usage
def all_path(source_var, disaster_var,text_var,root):
    source = source_var.get()
    target = disaster_var.get()

    text_var.set("Your shortest path is loading")
    root.update()

    if source not in people:
        text_var.set("Invalid location. Please enter a location from p1 to p5: ")
        root.update()
    elif target not in disaster:
        text_var.set("Invalid disaster type. Please enter 'fire', 'flood', or 'earthquake': ")
        root.update()
    else:
        all_path_correct(source_var, disaster_var, text_var, root)

1 usage
def shortest_path(source_var, disaster_var, text_var, root):
    source = source_var.get()
    target = disaster_var.get()

    text_var.set("Your shortest path is loading")
    root.update()

    if source not in people:
        text_var.set("Invalid location. Please enter a location from p1 to p5: ")
```

```
    if source not in people:
        text_var.set("Invalid location. Please enter a location from p1 to p5: ")
        root.update()
    elif target not in disaster:
        text_var.set("Invalid disaster type. Please enter 'fire', 'flood', or 'earthquake': ")
        root.update()
    else:
        shortest_path_correct(source_var, disaster_var, text_var, root)



target_label = tk.Label(node_frame, font=("Arial", 20) , text="Disaster (fire, flood, earthquake)")
target_label.pack()
# target_entry = tk.Entry(node_frame, font=("Arial", 20) , textvariable=disaster_var)
# target_entry.pack()

options = ['fire', 'flood', 'earthquake']
#
# Create a Combobox
target_entry = ttk.Combobox(node_frame,font=("Arial", 20), values=options, textvariable=disaster_var)
target_entry.set('fire')
target_entry.pack()

text_var = tk.StringVar()
text_var.set("Hello, you can give the relevant input!!")


# Label
label = tk.Label(node_frame, text="")
label.pack(padx=10, pady=10)  # padding on x and y axes

shortest_button = tk.Button(root, text="Shortest Path", font=("Arial", 20), command=lambda: shortest_path(source_var, disaster_var,text_var,root))
shortest_button.pack(padx=200)



all_paths_button = tk.Button(root, text="Show all paths", font=("Arial", 20), command=lambda: all_path(source_var, disaster_var,text_var,root))
all_paths_button.pack(padx=200)




canvas = FigureCanvasTkAgg(fig, master=root)
canvas.draw()
```

```
text_var = tk.StringVar()
text_var.set("Hello, you can give the relevant input!!")


# Label
label = tk.Label(node_frame, text="")
label.pack(padx=10, pady=10)  # padding on x and y axes

shortest_button = tk.Button(root, text="Shortest Path", font=("Arial", 20), command=lambda: shortest_path(source_var, disaster_var,text_var,root))
shortest_button.pack(padx=200)



all_paths_button = tk.Button(root, text="Show all paths", font=("Arial", 20), command=lambda: all_path(source_var, disaster_var,text_var,root))
all_paths_button.pack(padx=200)




canvas = FigureCanvasTkAgg(fig, master=root)
canvas.draw()
canvas.get_tk_widget().pack()

draw_graph(G, canvas, pos)

# Label
label = tk.Label(node_frame, text="")
label.pack(padx=10, pady=10)  # padding on x and y axes

label = tk.Label(root, textvariable=text_var, font=("Arial", 30), foreground="white", bg="black")
label.pack()

root.mainloop()
```