



---

# DONATION COLLECTION ROUTE OPTIMIZER FOR DISASTERS USING DIJKSTRA'S ALGORITHM

---

By: Brandon Everett



MARCH 2, 2025

CSC-506  
Module 7

## Donation Collection Route Optimizer for Disasters Using Dijkstra's Algorithm

**Introduction:** For this assignment, I adapted a food delivery route planning problem to create a program that uses Dijkstra's algorithm to find the best routes for collecting donations during wildfire relief efforts. I chose to modify the original food delivery context because of the recent devastation caused by wildfires and wanted to explore how similar routing algorithms could help emergency responders collect donations more efficiently.

```
# Global dictionaries
centers = {}
routes = {}

# Define urgency levels (higher number = more important)
urgency_values = {
    "Critical": 4,
    "High": 3,
    "Medium": 2,
    "Low": 1
}

# Define traffic slowdown factors
traffic_slowdown = {
    "Clear": 1.0,
    "Light": 1.2,
    "Moderate": 1.5,
    "Heavy": 2.0,
    "Severe": 3.0
}

def add_center(id, name, urgency, items):
    """Add a donation center"""
    centers[id] = {
        "name": name,
        "urgency": urgency,
        "items": items
    }
    routes[id] = []

def add_road(from_id, to_id, miles, traffic):
    """Add a road between centers"""
    # Calculate travel time (miles x traffic factor)
    travel_time = miles * traffic_slowdown[traffic]

    # Add the road in both directions
    routes[from_id].append((to_id, travel_time))
    routes[to_id].append((from_id, travel_time))
```

**Implementation:** I used Python with dictionaries to implement Dijkstra's algorithm. When a disaster like wildfire happens, getting supplies to victims quickly is crucial. My program considers three main factors: how urgently items are needed, how far apart donation centers are, and what the traffic conditions are like. The program lets users define donation centers with details like name, urgency level (Critical, High, Medium, Low), and types of items needed (water, medical supplies, food, etc.). It also stores information about routes between centers, including distance in miles and traffic conditions (Clear, Light, Moderate, Heavy, Severe).

```
if main():
    print("Wildfire Donation Route Finder")
    print("-" * 30)

    # Create our test network
    setup_test_network()

    # Print the centers
    print("\nOur Donation Centers:")
    for id in centers:
        print(f"{id}. {centers[id]['name']} - {centers[id]['urgency']} priority")
        print(f"    Needs: {', '.join(centers[id]['items'])}")

    # Find a route from Main Hub to Hospital (Example 1)
    print_route_info(1, 2) # Main Hub to Hospital

    # Find a route from Main Hub to Shelter (Example 2)
    print_route_info(1, 4) # Main Hub to Shelter

    # Find a route from Main Hub to Clinic (Example 3)
    print_route_info(1, 5) # Main Hub to Clinic

    print("\nProgram completed successfully.")

def main():
    pass
```

**Key Features:** My implementation includes a simple urgency priority system. Instead of using complex weighting formulas, I chose a straightforward numbering system (4 for Critical, 3 for

High, etc.) that makes routes to urgent locations appear shorter. The program also accounts for traffic adjustment where each type of traffic condition applies a multiplier to the actual distance (for example, Heavy traffic multiplies distance by 2.0). All roads between centers are bidirectional, allowing for flexible route planning. When the program runs, it displays all the donation centers in the network, and calculates the optimal path from the main hub to a critical destination.

```
def find_best_route(start_id, end_id):
    """Find the best route from start to end"""
    # Keep track of distances
    distances = {}
    for center_id in centers:
        distances[center_id] = float('inf') # Start with infinity
    distances[start_id] = 0 # Distance to start is 0

    # Keep track of visited centers
    visited = []

    # Remember how we got to each center
    previous = {}

    # Visit all centers
    while len(visited) < len(centers):
        # Find the unvisited center with the smallest distance
        current_id = None
        smallest_distance = float('inf')

        for center_id in centers:
            if center_id not in visited and distances[center_id] < smallest_distance:
                current_id = center_id
                smallest_distance = distances[center_id]

        # If we can't reach any more centers, stop
        if current_id is None:
            break

        # Look at all neighbors
        for neighbor_id, travel_time in routes[current_id]:
            # Make urgent centers appear closer
            urgency_bonus = urgency_values[centers[neighbor_id]]["urgency"] * 2
            adjusted_time = travel_time - urgency_bonus
            if adjusted_time < 0: # Don't let it go negative
                adjusted_time = travel_time / urgency_values[centers[neighbor_id]]["urgency"]

            # If we found a better path, update
            if distances[current_id] + adjusted_time < distances[neighbor_id]:
                distances[neighbor_id] = distances[current_id] + adjusted_time
                previous[neighbor_id] = current_id

        # No path found
        if end_id not in previous and end_id != start_id:
            return [], "No path found"

        # Create the path list
        path = []
        current = end_id

        # Work backwards from the end
        while current != start_id:
            path.insert(0, current)
            current = previous[current]

        # Add the start to the beginning
        path.insert(0, start_id)

    return path, distances[end_id]
```

**Time Complexity:** According to Wikipedia, Dijkstra's algorithm has a time complexity of  $O((V+E) \log V)$  when using a priority queue, where  $V$  is the number of vertices (donation centers) and  $E$  is the number of edges (routes between centers). My implementation uses a simpler approach with a time complexity of  $O(V^2)$ , which is sufficient for small networks typical in local emergency response scenarios (Dijkstra's algorithm, 2025).

**Challenges:** The biggest challenge was figuring out how to balance multiple factors in the route optimization. For instance, a shorter route with heavy traffic might be worse than a longer route with clear traffic. I solved this by applying a simple bonus system where critical locations receive larger urgency bonuses, making them appear closer in the algorithm's calculations. This approach was inspired by how Lusiani (2023) modified the algorithm to incorporate multiple factors. Another issue was understanding the algorithm itself. After studying the step-by-step explanations from GeeksforGeeks (2025) and the academic paper by Lusiani, Purwaningsih, and Sartika (2023), I was able to create a working implementation that considers multiple factors when finding the shortest path.

My original plan was to extend this system to track both "Haves" (available supplies) and "Needs" (required items) at each location, creating a more comprehensive resource matching system. This would have allowed the algorithm to prioritize routes where a location with excess supplies could deliver to locations with matching needs. However, after trying I ended up implementing a simpler version focusing on the core route optimization.

```

def print_route_info(start_id, end_id):
    """Print information about a route between two centers"""
    path, distance = find_best_route(start_id, end_id)

    start_name = centers[start_id]["name"]
    end_name = centers[end_id]["name"]

    print(f"\nBest route from {start_name} to {end_name}:")

    if path:
        # Print the path with center names
        route = []
        for id in path:
            route.append(centers[id]["name"])

        print(" -> ".join(route))
        print(f"Route priority score: {distance:.1f}")
    else:
        print("No route found!")

```

Our Donation Centers:

1. Main Hub - High priority  
Needs: Water, Food
2. Hospital - Critical priority  
Needs: Medical
3. School - Medium priority  
Needs: Food, Clothing
4. Shelter - High priority  
Needs: Blankets, Water
5. Clinic - Critical priority  
Needs: Medical, Water

Best route from Main Hub to Hospital:  
Main Hub -> Hospital  
Route priority score: 7.0

Best route from Main Hub to Shelter:  
Main Hub -> School -> Shelter  
Route priority score: 2.0

Best route from Main Hub to Clinic:  
Main Hub -> School -> Shelter -> Clinic  
Route priority score: 4.5

**Skills Learned:** This project taught me how to implement Dijkstra's algorithm in a practical way and how to modify it to handle multiple factors beyond just distance. I also learned about graph theory and how to represent real-world networks as graphs that algorithms can work with. I gained experience with Python's data structures, especially dictionaries, which were essential for building a beginner-friendly implementation. The output display helped me learn about presenting network data in an understandable way

**Conclusion:** This project shows how algorithms can help emergency responders find better routes during wildfires. I kept my program simple to show how Dijkstra's algorithm can find the best paths while considering multiple factors. Lusiani (2023) showed this algorithm works well for delivery services, and I adapted it for emergency response. My program finds the best routes to collect donations by looking at both distance and how urgently items are needed.

## References:

"Dijkstra's algorithm." Wikipedia, 2 March 2025,  
[https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm). Accessed 2 Mar. 2025.

"Find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm." GeeksforGeeks, 26 Feb. 2025, <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>. Accessed 2 Mar. 2025.

Lusiani, Anie, Siti Samsiyah Purwaningsih, and Euis Sartika. "Dijkstra Algorithm in Determining the Shortest Route for Delivery Service by J&T Express in Bandung." *Jurnal Lebesgue: Jurnal Ilmiah Pendidikan Matematika, Matematika dan Statistika*, vol. 4, no. 2, Aug. 2023. DOI: 10.46306/lb.v4i2.337. Accessed 2 Mar. 2025.