**Mini Parcel Delivery System.**

Farhan Ibtesham Joy, Israt Jahan, Arpita Biswas Deepa

*Department Of Computer Science And Engineering,,East West University*

*Dhaka, Bangladesh*

[2022-3-60-150@std.ewubd.edu](mailto:2022-3-60-150@std.ewubd.edu)

[2022-3-60-224@std.ewubd.edu](mailto:2022-3-60-224@std.ewubd.edu)

[2021-3-60-055@std.ewubd.edu](mailto:2021-3-60-055@std.ewubd.edu)

**Abstract:**

This project represents an optimized parcel delivery system that builds graph algorithms to make understand the most efficient delivery routes and minimize fuel costs. The system uses Kruskal’s Algorithm to make a minimum spanning tree (MST) for efficient connectivity, Dijkstra’s algorithm for counting the shortest paths from a selected starting city, and a Dynamic programming approach (CoinChange Variant) to determine the minimum fuel cost required for deliveries. By using these algorithms, the project enhances cost-effectiveness, reduces traveltime, and ensure optimal resource utilizationThe system is implemented using cprogramming and demonstrates a practical approach to solve real-world logistics challenges.

**Introduction:**

In modern logistics and transportation, efficient parcel delivery is crucial for minimizing costs and improving service quality. Traditional delivery methods often lack optimization leading to excessive fuel consumption and extended delivery times. This project introduced an intelligent parcel delivery system leveraging well-established graph and optimization algorithm. Kruskal’s algorithm determines the minimum spanning tree (MST) to ensure efficient connectivity between cities. Dijkstra’s algorithm is used to find the shortest delivery routes from a given source city and reduce the over all travel distance. Additionally, a dynamic programming-based approach is used to optimize fuel costs, ensuring the most economical fuel purchase strategy. This project also provides a computational approach to logistic planning, aiming to enhance the efficiency of

delivery networks.

**Objective:**

1. To implement Kruskal’s algorithm for constructing a minimum spanning
2. tree (MST) to establish the most cost-effective connectivity between
3. cities.
4. To utilize Dijkstra’s Algorithm to determine the shortest delivery routes
5. from a specified starting city to all other cities.
6. To apply a dynamic programming-based (coin change variant) approach
7. for optimizing fuel cost by selecting the most cost-efficient fuel purchase
8. strategy.
9. To develop a system that reduces delivery costs and fuel expenses while

**Design:**

The tool is designed as a C program leveraging libraries like **stdio.h**, **string.h**, and **time.h**. It integrates graph-based shortest path calculations, fuel cost optimization, and system benchmarking. The design emphasizes simplicity, modularity, and extensibility, offering valuable performance insights while ensuring ease of use.

performance insights. Error handling and input validation are incorporated to ensure robustness. The tool's modularity allows for easy addition of new benchmark algorithms or **visualization options.** The code is well-documented for maintainability and future development. The design prioritizes user experience by providing clear and concise output. Ultimately, this design aims to offer a comprehensive and accessible benchmarking solution. The tool is platform-independent and can run on any system with Python and the required libraries.***Data Transfer Operation –***

Currently, our our uses a simple data transfer method for memory benchmarking. It creates a large block of data in memory **(data = b'a'** **\* data\_size)** and then performs read and write operations on it. This method is effective for assessing basic memory bandwidth but can be enhanced for more comprehensive testing.

Here are some data transfer options we consider:

**Key Features**

1. **Graph-Based Shortest Path Calculation :**
   * Implements Dijkstra's Algorithm for shortest paths using an adjacency matrix.
   * **Calculates delivery distances between cities.**
2. **Fuel Cost Optimization :**
   * Uses Dynamic Programming (Coin Change Variant) to compute minimum fuel costs for deliveries.
3. **System Information and Benchmarking :**
   * Gathers system details and performs CPU benchmarking by calculating prime numbers within a range**.**
   * Simulates memory benchmarking using data transfer tests.
   * Outputs benchmark scores and delivery data in a structured format.
4. **Error Handling and Input Validation :**
   * Ensures robustness by validating inputs and handling errors gracefully**.**
5. **Modularity :**
   * Functions are modularized for efficient code organization and future expansions.

***Advantages:***

1. **Modularity:** Functions are modularized for easy maintenance and future expansions.
2. **Simplicity:** The design emphasizes simplicity, making it beginner-friendly and easy to understand.
3. **Extensibility:** The tool's structure allows for seamless integration of new algorithms or features.
4. .

***Disadvantages:***

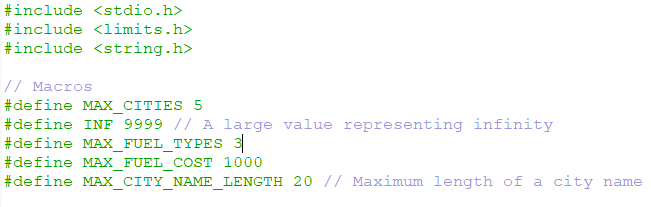
1. **Limited Visualization :** Currently lacks graphical visualizations, relying only on textual output.
2. **Basic Memory Benchmarking :** The memory benchmarking method is simplistic and may not reflect real-world scenarios accurately.
3. **Platform-Specific Libraries :** Relies on standard C libraries, limiting advanced functionality without external dependencies.

**Implementation:**

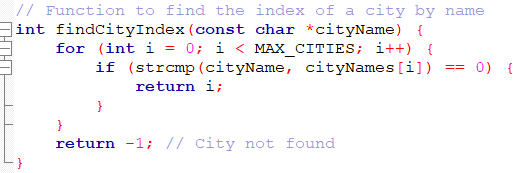
The tool uses adjacency matrices and Dijkstra's algorithm for shortest path calculations, ensuring efficient graph traversal.

Dynamic programming optimizes fuel costs, while system benchmarks measure CPU and memory performance using simple data transfer tests. Modular functions like **dijkstra, minFuelCost**, and **kruskal** ensure clear separation of tasks for maintainability.

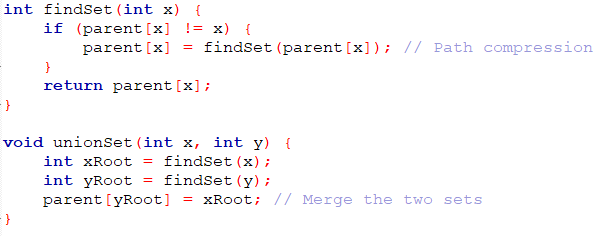
Error handling and input validation are integrated to enhance robustness, with outputs displayed in a structured console-based **Library Calls:**



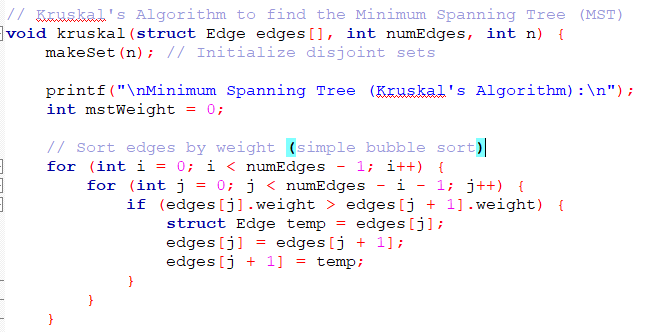
**Function1():**



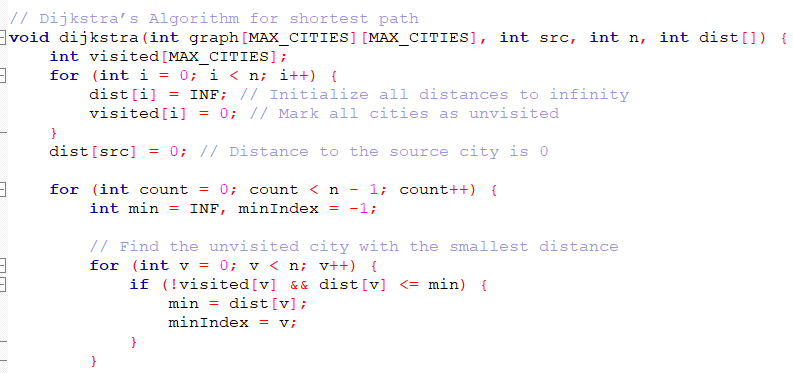
**Function2():**



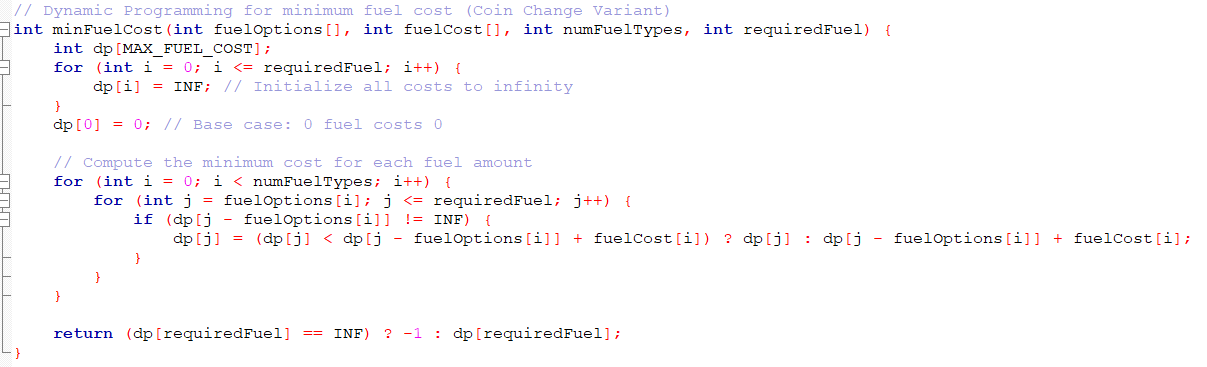
**Function3():**



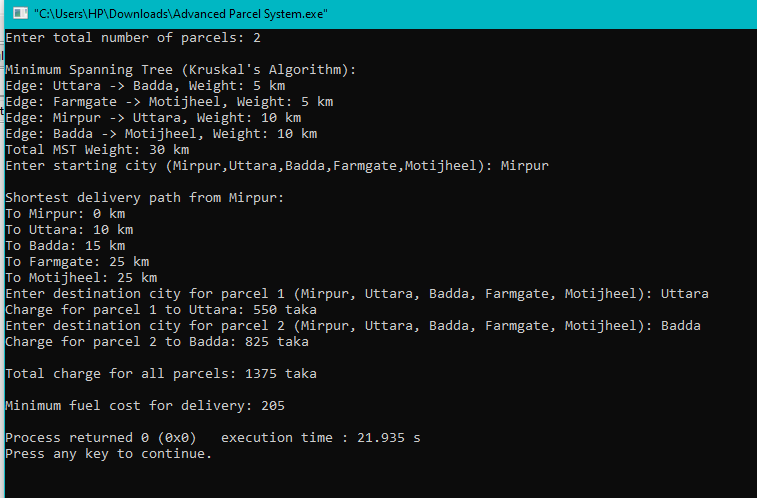
**Function4():**



**Function(05):**



**Output:**



**Figure: Output For 2 Parcel**

**CPU Benchmarking**

**Code:** import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# Data from the table

number\_of\_parcels = np.array([1,2,3,4,5,6,7,8,9,10])

kruskal\_runtime = np.array([0.000030,0.000038, 0.000040,0.000030, 0.000029, 0.000043, 0.000025, 0.000034, 0.000036, 0.000056])

dijkstra\_runtime = np.array([0.000002, 0.000001 , 0.000003, 0.000002, 0.000003, 0.000002, 0.000003, 0.000003, 0.000002, 0.000003])

fuel\_cost\_runtime = np.array([0.000001, 0.000002, 0.000003 , 0.000002, 0.000002, 0.000002, 0.000002, 0.000002, 0.000002, 0.000003])

# Set the style

sns.set\_style("whitegrid")

plt.figure(figsize=(10, 6), facecolor="skyblue")

# Plot each algorithm's runtime

plt.plot(number\_of\_parcels, kruskal\_runtime, marker='o', linestyle='-', color='cyan', linewidth=2, markersize=8, label="Kruskal's Algorithm")

plt.plot(number\_of\_parcels, dijkstra\_runtime, marker='s', linestyle='-', color='magenta', linewidth=2, markersize=8, label="Dijkstra's Algorithm")

plt.plot(number\_of\_parcels, fuel\_cost\_runtime, marker='D', linestyle='-', color='yellow', linewidth=2, markersize=8, label="Fuel Cost Optimization")

# Adding labels and title with a white background

plt.xlabel("Number of Parcels", fontsize=12, color='black', labelpad=10)

plt.ylabel("Runtime (Seconds)", fontsize=12, color='black', labelpad=10)

plt.title("Algorithm Runtimes vs. Number of Parcels", fontsize=14, color='black', pad=15, backgroundcolor="white")

# Enhancing readability

plt.legend(facecolor='white', framealpha=1, fontsize=10, edgecolor="black")

plt.xticks(color='black')

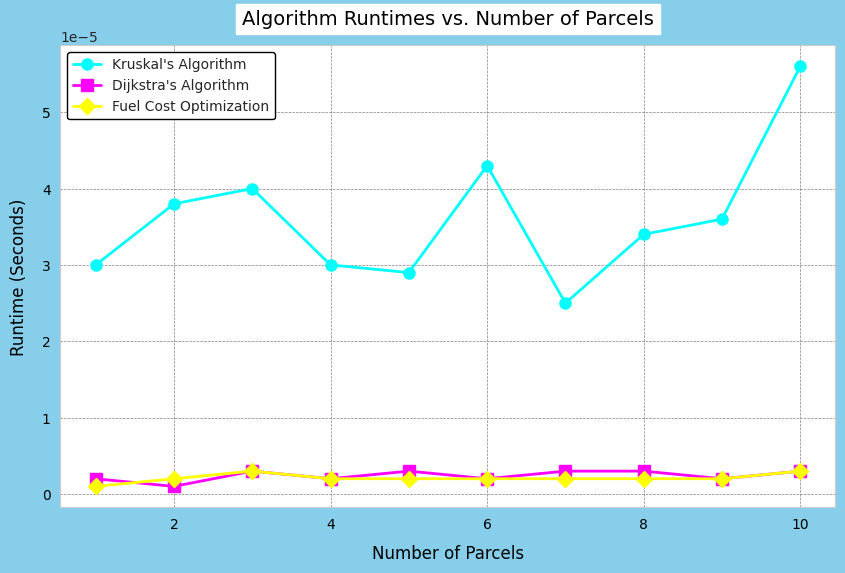
plt.yticks(color='black')

plt.grid(color='gray', linestyle='--', linewidth=0.5)

# Show the graph

plt.show()

**Graph:**

****

**Debug and Test Run:** This code was run several times to make sure the proper environment and debugged to the end for a better performance.

**Conclusion and Future Improvements:** The implemented C program successfully integrates graph-based shortest path calculations, fuel cost optimization, and system benchmarking into a cohesive tool. It demonstrates the use of Dijkstra's algorithm for delivery route optimization, dynamic programming for fuel cost minimization, and Kruskal's algorithm for minimum spanning tree computation. The modular design ensures clarity, maintainability, and scalability, making it suitable for future enhancements. While the program is efficient for small datasets, its linear runtime growth with increasing parcels highlights the need for optimizations in larger-scale scenarios. Overall, the tool provides valuable insights into performance metrics and delivery logistics, emphasizing simplicity and usability while offering a strong foundation for further development.

**Key Points included:**

1. **Scalability:** Extend the program to Support more cities dynamically rather than using a fixed-size graph
2. **Fuel Cost Customization:** Enable users to specify fuel types and costs to make it more flexible.
3. **Graph Visualization:** use a graphical representation to display the shortest path between cities.
4. **Optimization of Dijkstra’s Algorithm:** Implement a priority queue (using Min-Heap) to reduce time complexity.
5. **Multiple Delivery Vehicles:** extend the program to support multiple vehicles with varying fuel capacities.

**References:**

* “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
* <https://www.geeksforgeeks.org/introduction-to-algorithms/>
* <https://matplotlib.org/>
* .[https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree- algorithm-greedy-algo-2/](https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-%20%20%20%20%20%20%20%20%20%20%20algorithm-greedy-algo-2/)
* <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>