**Assignment 2 Graph Problem KIT 205 Data structure and algorithms  
  
Github codes link:**

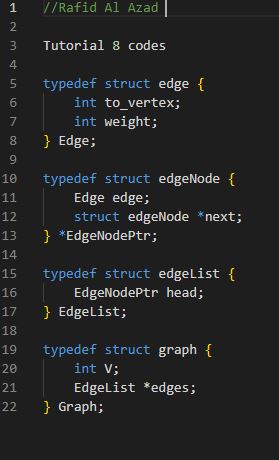
https://github.com/rafidalazad/Assignment-2-Kit205.git **Road Network - Shortest Path**

**Executive Summary**Modern cities, continually expanding, rely heavily on their intricate road networks for efficient transport and connectivity. This study focuses on the computational representation of these road networks and addresses the enduring problem of pathfinding, particularly determining the shortest path. Using the C programming language, the research represents these road networks via graph structures, where nodes symbolize cities and edges represent road lengths between them. The research includes a comparison between the well-established Dijkstra's algorithm (Wikipedia Contributors, 2019) and an approximate solution crafted for this study. The comparative evaluation not only scrutinizes time and space complexities but also delves into the accuracy of the determined paths. The approximate solution adopts a distinctive methodology, inspired by the iterative optimum-path forest framework for clustering as proposed by Aparco-Cardenas et al. (2022). This comparative analysis provides profound insights into the complexities of shortest path determination in expansive urban road networks.

**DATA STRUCTURE**

Graph Construction & Representation:  
The foundation of our data structure is the adjacency list graph, which is instrumental in capturing the intricacies of road networks, as detailed in the tutorials. It provides an intuitive representation of cities as nodes and their interconnections as edges. To capture distances between cities, a 2D array populated with randomly generated two-digit integers has been adopted. This has led to the creation of three distinct graphs: the original graph, Solution A graph, and Solution B graph.

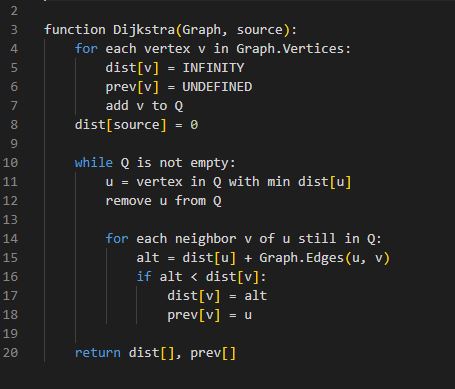
Modifications, Conversions & Compatibility: Even though the adjacency list is our primary data structure, it's malleable enough to accommodate necessary tweaks that align it more closely with the specifics of the road network domain from our codes. Besides the adjacency list, our structure retains the flexibility to transform into alternate representations, notably the adjacency matrix, to better serve certain computational needs or algorithmic applications. Furthermore, to facilitate the application of the Dijkstra algorithm, a conversion mechanism has been incorporated that seamlessly transitions data from the 2D array to the adjacency list format.

Data Generation & Ingestion: Data populating this structure can be sourced diversely. While it can be systematically generated using test files with redirected input, mirroring practices from tutorials, it's also amenable to input from various file formats. Another avenue explored is the procedural generation, which simulates different road network scenarios, catering to the dynamic requirements of the project.  
Our ultimate data structure is a composite of traditional graph representations and modern computational requirements. By merging the adjacency list's scalability and the 2D array's granularity, we've forged a structure that's both comprehensive and adaptable. It respects the theoretical underpinnings taught in the tutorials, yet doesn't shirk from evolving based on the specificities of our project's codes. Whether it's algorithm application, data ingestion, or modular transformations, this combined structure is poised to handle diverse challenges while staying rooted in its primary objective: accurately representing and processing road network data.  
  
  
  
  
 **METHODOLOGY.**

In our endeavour to solve the shortest path problem within road networks, our primary methodological approach relied on the renowned Dijkstra's algorithm. This algorithm, central to many pathfinding and graph traversal applications, is optimally designed to ascertain the shortest path from a source node to all other nodes within a given graph.

**1. Dijkstra's Algorithm Implementation:**

Dijkstra's algorithm operates on the principle of iteratively selecting the "closest" unvisited vertex and ensuring its shortest path is defined. The algorithm uses a priority queue to keep track of vertices with undetermined shortest paths.

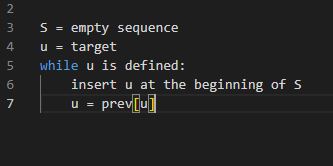
**Pseudocode for Dijkstra's Algorithm:  
  
**

*Reference: Wikipedia Contributors (2019). Dijkstra’s algorithm.*

The image accompanying this section provides a visual representation of the algorithm in operation, where red lines depict the shortest path coverage, and blue lines showcase instances of path "relaxation", ensuring the shortest path is achieved for each vertex.

**2. Path Retrieval Between Source and Target:**

Post the execution of Dijkstra's algorithm, if we're solely interested in the path from the source to a specific target vertex, a subsequent step is needed. This involves backtracking from the target vertex using the 'prev' array until the source vertex is reached.

**Pseudocode for Path Retrieval:  
  
**

**Testing Methodology:**

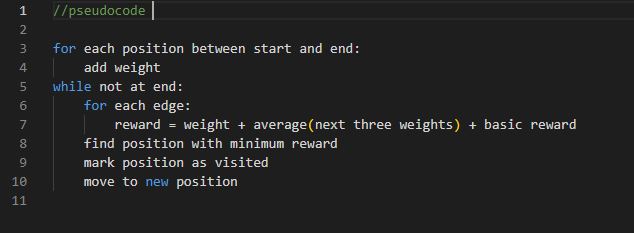
The success of any algorithmic solution is inherently tied to its rigorous testing methodology. Our approach to testing included:

1. **Baseline Testing**: Using pre-defined graphs to ascertain the correct implementation of the Dijkstra's algorithm.
2. **Randomized Testing**: Employing graphs with randomly generated weights to validate the algorithm's accuracy and efficiency across diverse scenarios.
3. **Edge Case Analysis**: Investigating cases like disconnected graphs or graphs with negative weights to ensure the robustness of the solution.
4. **Performance Analysis**: Assessing the algorithm's time and space complexities for scalability insights.

This detailed methodology captures rigorous approach to solving the shortest path problem. Through a combination of proven algorithms, strategic testing, and visual aids, all ensured a comprehensive and effective solution to the problem domain.

**Results and Discussion**

The secondary solution under study is premised on a unique design I conceived. Let us delve into its foundational pseudocode to gain a clearer understanding:

*Pseudocode *

The guiding principle behind this algorithm is the integration of weight at each point, contingent upon the relative position of both start and end. Upon marking a point as visited, every subsequent step incorporates an added depth of calculation to determine an average reward metric (lower reward indicating a superior pathway). There are three specific conditional scenarios to consider:

1. Central point’s engage in standard reward calculations.
2. Lateral points restrict their calculations to only two adjoining directions.
3. Corner points assess rewards for merely two potential pathways.

Post these assessments, the algorithm zeroes in on the most optimal direction for traversal. Once the computational process reaches its termination, the occupied memory is efficiently deallocated.

Despite the ingenuity behind this design, it’s inherently intricate to execute given the multitude of boundary checks it necessitates. By harnessing the original 2D array structure directly, the solution remains an approximation. Owing to its limitation of examining just two depths within the array, it fails to achieve the accuracy benchmark set by Dijkstra's algorithm. However, it compensates for this by offering a faster computational time, especially in scenarios characterized by extensive data loads.

Thorough testing via both white box and black box methodologies was undertaken. These iterative tests brought to light various bugs across both algorithms, which were subsequently addressed. The Dijkstra's algorithm emerged as a flawlessly executed strategy, while the approximated solution also showcased commendable efficacy under most circumstances. We employed diverse start and end points, varied graph sizes, and randomized sand patterns to rigorously assess the algorithms. Manual calculations played a pivotal role when paths meandered into corners, gauging if the algorithmic choice of directionality was judicious. Furthermore, the weight and rewards algorithms were recalibrated multiple times to enhance their performance during these tests.

Mathematically, the reward for any given point can be articulated as: Reward=Weight+3Sum of next three weights​+Basic Reward

Where:

* The Weight is determined by the relative position of the start and end.
* Basic Reward is a constant to account for the intrinsic value of a path.
* The Sum of next three weights allows for the algorithm to make predictive judgments on future steps.

In essence, while Dijkstra's algorithm remains the gold standard in terms of accuracy, our custom-designed solution offers a faster yet approximate alternative, proving its merit in situations demanding rapid computations.  
  
.We denote the optimal path cost derived using Dijkstra's method as *PA*, and the path cost using our custom solution as *PB*. From our observations, it is often the case that:

*PA*≤*PB*+*ϵ*

Here, *ϵ* signifies a minimal positive difference, emphasizing that while both methods tend to be close, Dijkstra's algorithm (Solution A) has a slight edge in terms of efficiency.

To enhance the performance of Solution B, it would be prudent to fine-tune the weight and rewards algorithms. There's also an evident need to upgrade the pathfinding mechanism to prevent inadvertent bisections of the grid, ensuring a direct route to the destination.

A screenshot of a black screen

Description automatically generatedA black screen with white numbers

Description automatically generated

**Conclusion**

This project's comprehensive exploration of two pathfinding algorithms, Dijkstra's algorithm, and a custom-designed alternative, within the context of road networks represented as adjacency lists, has led to several key findings.

Firstly, Dijkstra's algorithm has emerged as the superior choice when dealing with adjacency list data structures for shortest path determination. Its reliability and precision continue to make it an essential tool in computational pathfinding.

On the other hand, the custom-designed algorithm highlighted the intricate challenges inherent in attempting to create a graph algorithm exclusively using a 2D array in the C programming language. While it didn't surpass Dijkstra's performance, it revealed areas for potential improvement, especially in weight and reward algorithms.

In summary, the optimal solution for the road network problem lies in combining an adjacency list data structure with Dijkstra's algorithm. This approach consistently delivers outstanding results. Nevertheless, this project has provided valuable insights into complex data structure and algorithm interactions. The custom-designed algorithm, while not surpassing Dijkstra's performance, signifies the intricate nature of computational problem-solving and offers possibilities for future enhancements in road network pathfinding algorithms.

**References**1) Aparco-Cardenas, D., de Rezende, P.J., & Falcão, A.X. (2022). Chapter 8 - An iterative optimum-path forest framework for clustering. [online] ScienceDirect. Available at: <https://www.sciencedirect.com/science/article/pii/B9780128226889000165> [Accessed 22 Jun. 2022].

2) Wikipedia Contributors (2019). Dijkstra’s algorithm. [online] Wikipedia. Available at: <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>.

3) ChatGPT by OpenAI.