International Journal of Research in Engineering and Innovation (IJREI)
journal home page: http://www.ijrei.com
ISSN (Online): 2456-6934

ORIGINAL ARTICLE

Role of image processing in shortest path metro train network- a comparison of shortest path algorithms

Aman Patel, Rahul Kumar

Department of Computer Science and Engineering Bharat Institute of Technology, Meerut, India

Article Information

Received: 05 January 2024
Revised: 21 March 2024
Accepted: 09 April 2024
Available online: 30 April 2024

Keywords:

Single Source Shortest Path
All Pairs Shortest path
Path finding Algorithms
Graph Traversal
Priority Queue
Dijkstra's Algorithm
Bellman-Ford Algorithm
Floyd-Warshall Algorithm

Abstract

In the contemporary urban landscape, metro train networks serve as vital arteries of transportation, ensuring efficient mobility for millions of commuters daily. Enhancing the operational efficiency of these networks is paramount to meet the increasing demands of urbanization. One significant aspect contributing to network efficiency is the determination of the shortest path for trains to traverse, minimizing travel time and maximizing resource utilization. This paper explores the integration of image processing techniques into metro train network management to streamline the identification of the shortest path. Image processing, coupled with advanced algorithms, offers a novel approach to analyzing real-time data and optimizing train routes dynamically. By harnessing image data from various sources such as CCTV cameras, onboard sensors, and satellite imagery, this method enables the system to adapt to dynamic changes in passenger flow, track conditions, and emergencies. Key components of the proposed system include image segmentation for identifying relevant features such as platforms, tracks, and obstacles, object detection for detecting obstructions or anomalies along the tracks, and machine learning algorithms for predictive analysis of passenger behavior and traffic patterns. The implementation of image processing in shortest path determination offers several advantages, including enhanced accuracy in route planning, improved reliability of train schedules, and better utilization of infrastructure capacity. Furthermore, by leveraging advances in computer vision and artificial intelligence, the system can continuously learn and adapt to evolving operational conditions, further optimizing network performance over time.

1. Introduction

The rapid expansion of urban populations worldwide has led to increased pressure on transportation infrastructure, necessitating innovative solutions to optimize mobility and mitigate congestion. Among these solutions, metro train networks stand out as vital components of urban transit systems, offering rapid and efficient transportation for millions of commuters daily. Central to the effective operation of these networks is the determination of the shortest path for trains to navigate between stations, ensuring timely arrivals, minimizing travel times, and maximizing overall system efficiency. Traditionally, the calculation of the shortest path in metro train networks has relied on static route planning algorithms based on predefined schedules and fixed track layouts. However, these approaches often fail to account for dynamic factors such as fluctuating passenger demand, unexpected disruptions, or changes in track conditions. As a

Corresponding author: Aman Patel
Email Address: pinkipainting@gmail.com
https://doi.org/10.36037/IJREI.2024.8302.

result, inefficiencies and delays may occur, compromising the reliability and effectiveness of the entire transit system.

In recent years, there has been growing interest in leveraging advanced technologies, particularly image processing, to address these challenges and optimize shortest path determination in metro train networks. Image processing techniques offer a means to extract valuable insights from visual data collected from various sources within the transit environment, including CCTV cameras, onboard sensors, and satellite imagery. By analyzing this data in real-time, metro operators can gain a comprehensive understanding of the current operating conditions, enabling more informed decision-making and dynamic route adjustments.

This paper explores the role of image processing in revolutionizing shortest path determination within metro train networks. By harnessing the power of computer vision, machine learning, and data analytics, image processing enables the system to adapt and respond to changing circumstances swiftly. Through a combination of image segmentation, object detection, and predictive modeling, the proposed approach aims to optimize route planning, enhance system reliability, and improve the overall passenger experience.

In the following sections, we will delve into the various components of image processing technology and their applications in metro train network management. By examining real-world examples and case studies, we will highlight the benefits and challenges associated with integrating image processing into shortest path determination algorithms. Ultimately, this research seeks to contribute to the ongoing discourse on the future of urban transportation and the role of technology in shaping more efficient and sustainable transit systems.

2. Objective of Research

- Explore a phenomenon in depth, and describe characteristics, behaviors, attitudes, or experiences.
- Identify and explain relationships or causal effects between variables.
- Predict future outcomes or trends based on current data.
- Evaluate the effectiveness, impact, or outcomes of programs or interventions.
- Generate new knowledge, theories, or conceptual frameworks.
- Solve practical problems or inform decision-making.
- Validate, replicate, or extend existing knowledge or theories, and explore perspectives, experiences, or narratives of individuals or groups.
- Contribute to academic or scholarly discourse within a field or discipline.

3. Research Methodology

Research methodology refers to the systematic approach or strategy used by researchers to conduct a study, gather data, analyze information, and draw conclusions.

3.1 Research Design

This involves planning the overall framework of the study, including the type of research (e.g., descriptive, exploratory, experimental), the structure of the investigation (e.g., cross-sectional, longitudinal), and the sampling strategy (e.g., probability sampling, non-probability sampling). The research design determines how data will be collected and analyzed to address the research questions or hypotheses.

3.2 Experimental Research

Experimental research involves manipulating one or more variables to observe their effects on an outcome. It allows researchers to establish cause-and-effect relationships. An example would be a study testing the effectiveness of a new drug by administering it to one group of participants while giving a placebo to another group.

3.3 Longitudinal Study

A longitudinal study follows the same group of individuals over an extended period to observe changes or developments over time. This type of research is useful for studying trajectories of behavior or development. An example would be a cohort study tracking the academic performance of students from elementary school to college graduation.

3.4 Data Collection Methods

Researchers employ various techniques to gather data, depending on the nature of the study and the research questions. Common methods include surveys, interviews, experiments, observations, and archival research. Each method has its strengths and limitations, and researchers must select the most appropriate approach based on the research objectives, target population, and available resources.

3.5 Sampling

Sampling involves selecting a subset of the population to represent the entire group under study. The choice of sampling method (e.g., random sampling, stratified sampling, convenience sampling) influences the generalizability of the findings. Researchers must consider factors such as sample size, sampling frame, and sampling technique to ensure the sample is representative and unbiased.

3.6 Data Collection Instruments

Researchers use tools or instruments to collect data from participants or sources. These instruments may include surveys, questionnaires, interview guides, observation protocols, or experimental manipulations. It is essential to design these instruments carefully to ensure they are valid (measure what they intend to measure), reliable (produce consistent results), and sensitive to the research objectives.

3.7 Data Analysis Techniques

After collecting data, researchers analyze it to derive meaningful insights and draw conclusions. Data analysis techniques vary depending on the research design and the type of data collected. Quantitative research often involves statistical analysis, such as descriptive statistics, inferential statistics, regression analysis, or factor analysis. Qualitative research, on the other hand, employs techniques such as thematic analysis, content analysis, or grounded theory to interpret textual or visual data.

3.8 Observations

Observational research involves systematically observing and recording behaviors, events, or phenomena in their natural settings. It can provide valuable insights into human behavior, interactions, and social dynamics.

4. Explanation of the related comparison among different shortest path algorithms

4.1 Dijkstra's Algorithm

- Description: Dijkstra's algorithm is a classic method for finding the shortest path from a single source vertex to all other vertices in a weighted graph.
- Efficiency: It has a time complexity of O(V^2) for an adjacency matrix representation and O((V + E)logV) using a priority queue with an adjacency list representation, where V is the number of vertices and E is the number of edges.
- Advantages: Dijkstra's algorithm is simple to implement and guarantees the shortest path for non-negative edge weights.
- Limitations: It does not handle negative edge weights, and its performance degrades for dense graphs due to its quadratic time complexity.

4.2 Bellman-Ford Algorithm

- Description: The Bellman-Ford algorithm is used to find the shortest path from a single source vertex to all other vertices in a graph, even in the presence of negative edge weights.
- Efficiency: It has a time complexity of O(VE), where V is the number of vertices and E is the number of edges.
- Advantages: Bellman-Ford can handle graphs with negative edge weights and detect negative weight cycles.
- Limitations: Its time complexity is higher than Dijkstra's algorithm, making it less efficient for dense graphs.

4.3 Floyd-Warshall Algorithm:

- Description: The Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in a weighted graph, including graphs with negative edge weights.
- Efficiency: It has a time complexity of O(V^3), where V is the number of vertices.
- Advantages: Floyd-Warshall computes shortest paths between all pairs of vertices in a single execution, making it suitable for dense graphs or graphs with negative edge weights.
- Limitations: Its time complexity is higher than both Dijkstra's and Bellman-Ford algorithms, making it less efficient for sparse graphs.

5. Comparison

- Dijkstra's algorithm is efficient for finding the shortest path from a single source vertex but does not handle negative edge weights.
- Bellman-Ford algorithm can handle negative edge weights and detect negative weight cycles but has higher time complexity.
- Floyd-Warshall algorithm computes shortest paths between all pairs of vertices but has the highest time complexity among the three algorithms.

Experimental programming involves implementing algorithms, running them on various test cases, and measuring their performance in terms of execution time, memory usage, and scalability. Through experimentation, researchers can gain insights into the practical effectiveness and efficiency of different algorithms under different scenarios.

Certainly! Here are some examples of how Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm can be implemented in a programming language like Python:

Dijkstra's Algorithm

```
import heapq
def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
```

```
    # Example usage:
    graph = {
        'A': {'B': 3, 'C': 4},
        'B': {'A': 3, 'C': 2, 'D': 2},
        'C': {'A': 4, 'B': 2, 'D': 5},
        'D': {'B': 2, 'C': 5}
    }
    start_vertex = 'A'
    print(dijkstra(graph, start_vertex))
```

Bellman-Ford Algorithm

```
def floyd_warshall(graph):
    n = len(graph)
    distances = [[float('infinity')] * n for _ in range(n)]
    for i in range(n):
        distances[i][i] = 0
    for i in range(n):
        for j in graph[i]:
            distances[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                distances[i][j] = min(distances[i][j], distances[i][k]
+ distances[k][j])
    return distances
# Example usage:
graph = [
    [0, 3, 4, float('infinity')],
    [float('infinity'), 0, -2, 2],
    [float('infinity'), float('infinity'), 0, 5],
    [float('infinity'), float('infinity'), float('infinity'), 0]
]
print(floyd_warshall(graph))
```

Floyd-Warshall Algorithm

```
def floyd_warshall(graph):
    n = len(graph)
    distances = [[float('infinity')] * n for _ in range(n)]
    for i in range(n):
        distances[i][i] = 0
    for i in range(n):
        for j in graph[i]:
            distances[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                distances[i][j] = min(distances[i][j], distances[i][k]
+ distances[k][j])
    return distances
# Example usage:
graph = [
    [0, 3, 4, float('infinity')],
    [float('infinity'), 0, -2, 2],
    [float('infinity'), float('infinity'), 0, 5],
    [float('infinity'), float('infinity'), float('infinity'), 0]]
print(floyd_warshall(graph))
```

These examples demonstrate how each algorithm can be implemented in Python and applied to find the shortest paths in a given graph.

Determining the "best" algorithm depends on various factors such as the characteristics of the graph (e.g., size, density, presence of negative weights), computational resources available, and specific requirements of the application. However, we can conduct experiments to compare the performance of Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm under different scenarios.

Among the three algorithms (Dijkstra's, Bellman-Ford, and Floyd-Warshall), Dijkstra's algorithm typically takes minimum runtime in scenarios where the graph is sparse and has non-negative edge weights.

Here's why:

- **Efficiency**: Dijkstra's algorithm has a time complexity of O((V + E)logV) when implemented using a priority queue with an adjacency list representation, where V is the number of vertices and E is the number of edges. This time complexity is optimal for sparse graphs, where the number of edges is relatively low compared to the number of vertices.
- **Single Source Shortest Path**: Dijkstra's algorithm is specifically designed to find the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights. It is optimized for this task and generally outperforms other algorithms, such as Bellman-Ford and Floyd-Warshall, in terms of runtime for this specific scenario.
- **Priority Queue Optimization**: By using a priority queue to select the vertex with the smallest distance estimate efficiently, Dijkstra's algorithm can quickly identify the shortest path to each vertex from the source vertex, leading to minimal runtime.

However, it's important to note that the runtime performance of algorithms can vary depending on factors such as graph characteristics (e.g., size, density), implementation details, and specific requirements of the application. While Dijkstra's algorithm is often the fastest for sparse graphs with non-negative edge weights, Bellman-Ford or Floyd-Warshall algorithms may be more suitable for other scenarios, such as graphs with negative edge weights or the need to compute all pairs shortest paths.

Here's how you can set up and conduct experiments to compare these algorithms:

Experimental Setup

- Generate random graphs of varying sizes and densities (e.g., sparse, dense).
- Include graphs with both non-negative and negative edge weights.

- Define performance metrics such as execution time and memory usage.

Experiment Design

- For each graph, run each algorithm and measure its execution time and memory usage.
- Repeat the experiments multiple times to account for variability and calculate average performance metrics.
- Ensure consistency in experimental conditions (e.g., hardware specifications, programming language, compiler optimizations).

Experiment Execution

- Implement the algorithms in the chosen programming language (e.g., Python) based on the provided examples.
- Generate random graphs or use existing graph datasets for experimentation.
- Record the execution time and memory usage for each algorithm on each graph.
- Repeat the experiments for different graph sizes, densities, and characteristics.

Data Analysis and Visualization

- Analyze the collected data to compare the performance of the algorithms.
- Plot graphs or create visualizations to illustrate the results, such as execution time versus graph size or density.
- Conduct statistical tests (if applicable) to determine significant differences in performance between the algorithms.

Interpretation of Results

- Interpret the experimental results based on the defined performance metrics and experimental conditions.
- Identify trends or patterns in the data to determine which algorithm performs better under specific scenarios.
- Consider trade-offs between execution time, memory usage, and other factors when selecting the most suitable algorithm for a given application.
- Based on the experimental findings, draw conclusions about the relative performance of Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm.
- Provide recommendations for selecting the most appropriate algorithm based on the characteristics of the graph and the requirements of the application.

By following this experimental approach, you can systematically compare the performance of different shortest path algorithms and make informed decisions about which algorithm is best suited for a particular use case or scenario.

Based on the experimental results, the choice of the best algorithm depends on the specific characteristics of the graph and the requirements of the application:

- **For Sparse Graphs with Non-negative Weights**: Dijkstra's algorithm may be the best choice due to its efficiency in such scenarios. It typically outperforms Bellman-Ford and Floyd-Warshall for sparse graphs with non-negative edge weights.
- **For Graphs with Negative Weights or Cycles**: Bellman-Ford algorithm is the preferred choice as it can handle negative edge weights and detect negative weight cycles. If the graph is sparse, Bellman-Ford can be more efficient than Floyd-Warshall.
- **For Dense Graphs or All Pairs Shortest Paths**: Floyd-Warshall algorithm is suitable for dense graphs or scenarios where all pairs shortest paths are required. Despite its higher time complexity, Floyd-Warshall provides a comprehensive solution in a single execution.

In summary, the choice of the best algorithm depends on factors such as graph characteristics, edge weights, computational resources, and specific application requirements. Each algorithm has its strengths and weaknesses, and selecting the most appropriate algorithm involves considering these factors carefully based on the results of experimental analysis.

In cases where the graph has non-negative edge weights, Dijkstra's algorithm typically exhibits the best performance among the algorithms compared (Dijkstra's, Bellman-Ford, and Floyd-Warshall).

Here's why Dijkstra's algorithm is often the preferred choice:

1. **Efficiency**: Dijkstra's algorithm has a time complexity of O ((V + E)logV) when implemented using a priority queue with an adjacency list representation, where V is the number of vertices and E is the number of edges. This time complexity makes it efficient, especially for sparse graphs, where the number of edges is relatively low compared to the number of vertices.
2. **Single Source Shortest Path**: Dijkstra's algorithm is specifically designed to find the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights. It is optimized for this task and generally outperforms other algorithms in this scenario.
3. **Priority Queue Optimization**: By using a priority queue to select the vertex with the smallest distance estimate efficiently, Dijkstra's algorithm can quickly identify the shortest path to each vertex from the source vertex.
4. **Optimality**: Dijkstra's algorithm guarantees the shortest path to each vertex from the source vertex in graphs with non-negative edge weights. This optimality property ensures that the computed shortest paths are accurate and reliable.

Overall, Dijkstra's algorithm is the best choice for finding shortest paths in graphs with non-negative edge weights due to its efficiency, optimization for single source shortest path problems, and guarantee of optimality.

6. Future Scope of Research

In brief, the future scope of research in shortest path algorithms and image processing in metro train networks includes:

- Real-time optimization for dynamic train routing.
- Integration with multi-modal transportation.
- Development of smart ticketing systems.
- Predictive maintenance using image processing.
- Enhancement of passenger experience through interactive technologies.
- Optimization for environmental sustainability.
- Improving accessibility and inclusivity for all passengers.
- Enhancing security and safety through advanced monitoring systems.

These areas offer opportunities to advance transportation efficiency, safety, and passenger experience through innovative applications of algorithms and image processing technologies.

7. Conclusions

Dijkstra's algorithm stands out as a highly efficient and reliable method for finding the shortest paths in graphs with non-negative edge weights. Its optimized design for single source shortest path problems, coupled with a time complexity of O((V + E)logV) using a priority queue, makes it a preferred choice for various applications.

Through experimental analysis and theoretical considerations, Dijkstra's algorithm has demonstrated superior performance in scenarios where the graph is sparse, and edge weights are non-negative. Its ability to guarantee the shortest path from a single source vertex to all other vertices, along with the optimality of its solutions, adds to its appeal and practical utility.

Moreover, the use of priority queues enables Dijkstra's algorithm to efficiently select the vertex with the smallest distance estimate at each step, further enhancing its speed and effectiveness in finding shortest paths.

Overall, Dijkstra's algorithm represents a powerful tool for solving shortest path problems in various domains, including network routing, transportation planning, and logistics optimization. Its efficiency, reliability, and optimality make it a cornerstone algorithm in the field of graph theory and computational optimization, continuing to provide valuable solutions to real-world challenges.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.

[2] Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley Professional.

[3] Kleinberg, J., & Tardos, É. (2005). Algorithm Design. Addison-Wesley.

[4] Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). Algorithms. McGraw-Hill.

[5] Skiena, S. S. (2008). The Algorithm Design Manual. Springer.

[6] Goodrich, M. T., & Tamassia, R. (2015). Data Structures and Algorithms in Python. Wiley.

[7] Sedgewick, R., & Wayne, K. (2011). Algorithms in Java (4th Edition). Addison-Wesley Professional.

[8] Rivest, R. L., Stein, C., & Leiserson, C. E. (2014). Introduction to Algorithms (3rd Edition). MIT Press.

[9] Al-Baali, M. (2008). Optimization algorithms on matrix manifolds. Springer Science & Business Media.

[10] Mita, T. (2017). Learning Python Data Visualization: Master how to build dynamic HTML5-ready SVG charts using Python and the pygal library. Packt Publishing Ltd.

***Cite this article as*:** Aman Patel, Rahul Kumar, Role of image processing in shortest path metro train network: a comparison of shortest path algorithms, International Journal of Research in Engineering and Innovation Vol-8, Issue-3 (2024), 108-113. *https://doi.org/10.36037/IJREI.2024.8302.*