iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 12 **Issue:** XI **Month of publication:** November 2024

DOI: https://doi.org/10.22214/ijraset.2024.65444

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 12 Issue XI Nov 2024- Available at www.ijraset.com

Optimized Implementation of Dijkstra's Algorithm for Efficient Shortest Path Finding in Networked Systems

Siddhesh Narnavre[1], Pranav Patil[2], Vedant Surkar[3], Sachin Pandarkar[4], Dr. Gauri Ghule[5]
Electronics and Telecommunications Department, Vishwakarma Institute of Information Technology, Pune, Maharashtra, India

Abstract: ***Dijkstra's Algorithm, a popular answer based graph-based fully objective shortcut search, plays an important role in various communication systems and optimization problems These rules are important in packages of IoT networks, smart website online visitors management, and telecommunications, of which green-tracking and real-time priority-making are paramount In this paper, we discover now not most effective the classical software of Dijkstra's Algorithm but additionally introduce optimizations that beautify its performance in largescale dynamic systems. [1] By tailoring the algorithm for particular contexts, including real-time records streams in IoT or adaptive routing in clever towns, we reveal sizeable enhancements in each computational overall performance and aid allocation. Our outcomes highlight the set of rules's capability to evolve to modern challenges, ensuring quicker, greater accurate decisionmaking throughout numerous industries. The proposed upgrades open the door to further packages in future shrewd networks, offering solutions that might pressure improvements in self sufficient structures and scalable infrastructures. [2]***

I. INTRODUCTION

A. Background

Shortest path algorithms form the heart of research in computer science, graph theory, and network optimization for over half a century. It is a definition that assists in finding an optimal route to traverse through networks, whether physical such as roads or railways, or virtual such as communication networks or data routing in the internet. [3] Graph theory is used for formulating a problem of finding the shortest path, so that a network is represented as a graph with nodes (vertices) and edges—that is a connection between nodes or a link —each having an associated weight, which denotes the cost or distance or time to travel between them. [4] In fact, there are very many real-world applications where the solution to this shortest path problem is essential. Whether in transportation systems (as determining, for example, the fastest route between two cities), telecommunication systems where flow of packets of data can be optimized, or any other domain, shortest path algorithms allow efficient and rational decisions aimed at costdiscrimination and resource-optimization. [1]

B. History

Dijkstra's algorithm was described by its creator, Edsger W. Dijkstra, in 1956, a time when computers were still new and a lot of efforts were undertaken to formulate efficient algorithms of low complexity. Dijkstra sought to solve the shortest path problem in a computationally efficient fashion for graphs with non-negative edge weights. [1] Few algorithms have gained rapid significance based on their simplicity, efficiency, and provision of optimality with certainly. Over time, Dijkstra's algorithm has accrued itself to be a common tool in many fields of computer science and engineering, especially in areas demanding graph traversals and optimization. [5] In theory, it propounds a greedy methodology, where the immediate local optimization is made a decision for each step, which eventually leads to a globally optimal solution. [6]

C. Principle of Dijkstra Algorithm

A Dijkstra algorithm is developed for the problem of singlesource shortest path, that is designed for calculating all shortest paths from source node to other vertices in the graph. The assumption is that the edge weights are non-negative, to make sure paths get processed in the increasing order of distance from the source with respect to the input data containing nodes and their respective distances from the source. [3] Such an algorithm can be nicely summarized as presented below: Initialization-assign distance 0 to the source node and infinity to all other nodes. Priority Queue: It is a data structure that can store one or more nodes and their current shortest distance. [6]

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

D. Greedy Selection

From every source node the node that has the shortest distance is chosen more than once till it becomes relaxed, in other words, if there's a possibility to reduce that path by passing through this selected node. Update: Whenever the shortest path to any of its neighbors is obtained from the selected node, the distance value of this node is updated again and pushed into the priority queue. [3] This algorithm has a time complexity of O(V2), being used with the adjacency matrix, which can be improved to O(E log V) by using a priority queue (min-heap) on the graph being represented as an adjacency list, in which the number of vertices and edges are denoted by V and E respectively.

E. Problem Definition

The problem of finding the shortest path in a given transportation network has been at the forefront of research in mathematics and computer science as they are simple yet form the backbone of essential applications. As the system dynamics change and become more complex, whether in transportation, telecommunications, logistics, or the emerging ubiquitous systems, paths must be optimally routed and all the decisions made in real-time for efficient performance, cost, and scalability. [4] Hence, the problem of disjunctive shortest path is truly that of searching for an ideal, where there are two or more nodes in a graph whose nodes are entities such as locations, a device, or a server; while edges are the possible connections or paths with their corresponding cost(s), such as distance, time, energy, and/or bandwidth.

1) To optimize traffic flow: In telecommunications, the shortest paths are important for delimited prompts to control the flow through routers and switches. Routers employ shortest paths based on Dijkstra's algorithms for continuously evaluating and keeping on choosing the optimum data route in the light of latency optimization and further improvements such as Keith Dave's inability to sing to boost the network's performance based on bandwidth. As global Internet traffic keeps increasing, the need for efficient routing protocols to accommodate congestion while holding on to acceptable quality-of-service (QoS) standards and optimized resource allocation is on the increase. [4]

2) Smart Transport Systems: Urban environments often employ smart transport systems to limit congestion and reduce travel times and fuel consumptions. Real-time traffic management systems, autonomous vehicle networks, and intelligent traffic lights need algorithms to compute shortest paths in a road network. These systems are dynamic by nature, and their underlying paradigms are responsive to changing traffic conditions. [7]

3) Energy-Efficient IoT Networks: The development of the IoT networks has added an extra layer of complexity in communication; it is such that numerous small energyconstrained devices, all needed to communicate with one another in a highly efficient manner. In these networks, the goal is locating the shortest path in bandwidth and latency reduction, accompanied by energy consumption reduction. Shortest path algorithms are very useful to IoT devices, particularly the ones deployed in low-power conditions like sensor networks or smart cities, where they are primarily designed to give the device increased battery life while minimizing energy costs incurred in communications. [8]

4) Cloud Computing and Edge Computing: In cloud computing environments, wherein tasks are distributed among multiple servers and data centers, the realization of the shortest communication path between nodes in the network minimizes latency and resource utilization. In edge computing, in which data gets processed at a location closer to the source, at the network edge, the shortest path algorithms help in determining the fastest and most efficient routes for processing and data delivery. [9]

5) Optimizing logistics and supply chains: The third important stage on our list involves using the shortest path algorithm in decision-making related to logistics and supply chains. Thus, this helps getting the cargo to the destination in a time-saving and economical manner, while improving warehouse management at the same time. Shortest path algorithms would be used for routing of delivery trucks in a proper manner and optimizing inventory management of distribution centers. [10]

II. RELATED WORK

Over decades, the number of algorithms developing for various shortest paths changed, each gathering up into themselves features particular for the network types, structure of the graph, and performance requirements involved in routing operations. In the following paragraphs I highlight some of the more important algorithms used presently in this area and relevant
to current applications. [11]

Dijkstra's algorithm, which was formulated in 1956, is one of the most widely used algorithms for finding the shortest path in non-negative edge weight graphs and certainly attracted significant interest in dealing with finding such paths. Selecting waiting nodes per the greedy reasoning, a source to destination may quickly be arrived at in a shortest manner. Whereas its basic version has a time complexity of $O(V^2)$.1001[10]

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

Where V is the number of nodes; optimally, it can be done at O(E log V) complexity. A priority queue is adopted in many applications, such as network routing (i.e. OSPF protocol), traffic management, and interoperability in IoT communication. [4]
However, Dijkstra faces the challenge of:

1) Dealing with graphs with negative weights cannot be figured out.
2) Costly when operating on large scenario graphs.
3) Not suitable for dynamic or real-time networks, where weights may frequently change.

There are some other alternatives which help overcome this challenges:

A. Bellman-Ford Algorithm:

The Bellman-Ford algorithm is an alternative to Dijkstra that can be applied to negative edge weight graphs. The thing that makes Bellman-Ford different from Dijkstra is that it does not require non-negative weights, which enables applications where cost or risk declines (like financial networks or risk modeling). But the slow-running algorithm has a time complexity of O(VE), making it pretty much inapplicable to large graphs. [12] In fact, because of its detection of negative weight cycles (i.e., cycles that, when incorporated in paths in the graph, would decrease our path cost indefinitely), applications that demand dynamic or evolving graph changes sometimes do rely on the Bellman-Ford algorithm, albeit its complexity is higher.

B. The A Star Algorithm

The A* algorithm adds to Dijkstra's algorithm a heuristic function which essentially looks for the most efficient way to guide the process. It combines distance already traversed from the source node and an estimated remaining distance to the destination such that number of nodes explored quickly reduces. A* therefore becomes one of the most efficient algorithms for pathfinding especially in large and complex environments like gaming, robotics, and navigation systems. [7] Although A* is used in several applications like real-time route planning for GPS systems and robot navigation, the heuristic function must be carefully designed in order to ensure optimum performance. If the heuristic is poorly designed, A* might be less efficient than Dijkstra's algorithm.

C. Bidirectional Search

Bidirectional search is another approach ranging from all such pathfinding methods that speedup the process in large networks. It runs one Dijkstra (?) or A* searching from the source and the other is searching from the destination, terminating the search when the two meet in the middle. This basically reduces the amount of search space very much and can improve the performance as fast as two times. It is mainly applicable in routing applications wherein the start and end are known: transportation and logistics.

D. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is for determining the shortest paths between all pairs of nodes in the graph. It is applicable for dense graphs and can support both positive and negative edge weights (although not with negative weight cycles). The algorithms require a time complexity of O(V3). The algorithm has more computational complexity than Dijkstra but is used in those applications where a whole matrix of shortest paths is required-such as in the network optimization of telecommunications or traffic engineering. [13]

III. OVERVIEW

Dijkstra's Algorithm is the well-known solution to singlesource shortest path problems in graphs with non-negative edge weights. Published by Edsger W. Dijkstra in 1956, it follows the greedy approach and finds huge applications in the domains of computer networks and routing protocols, GPS navigation systems and more traffic management systems. The algorithm optimizes the solution through repeated exploration of all the shortest paths from a given source node to all the other nodes in the graph, always expanding the smallest known distance node. [3]

A. Algorithm Basics

Dijkstra's algorithm operates on a set of nodes whose shortest path distances from the source node are already known and gradually expands this set by picking, at each step, the node with the smallest known distance. The greedy approach forms the basis for this algorithm; locally optimal choices are made in the expectation of arriving at a global optimum. [2] Core algorithm steps are as follows:

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

1) Initialization

Assign 0 distance to source node, and infinity to all other nodes. To avoid having unbounded distance represented by just any number close to infinity, we can keep a priority queue, typically a min-heap, with the nodes that need to be explored. The priority of a node n in the priority queue is its current shortest known distance from the source. All nodes are marked unvisited.

2) Selection

While there are nodes yet remaining to be visited in the priority queue, remove from it the node which is nearest to, hence has the minimum distance from the source, s at this stage. Mark this node as visited. The shortest path distance to this node is now settled.

3) Relaxation

At each node current node of the path, for each neighbour of the current node, replace the present value of the distance to the neighbour from the current node by the sum from the current node to the neighbour via the current node, if such a path exists. This step is known as relaxation, and it guarantees that the minimum distance to the neighbour from the current node will be updated appropriately if a shorter path does exist. If the neighbour's distance has been updated, it is re-inserted into the priority queue for further processing.

4) Termination

And the algorithm must terminate in one of these two ways: either all nodes must have been visited or there are no possible improvements of the shortest paths, and this is only possible when the priority queue is empty; and, moreover, it then means that the shortest path from the source node to any other node in the graph has been determined.

B. Matematical Model

Graph: Let us consider Dijkstra's algorithm to be incorporated into a planar graph as a networked graphical representation: G can be described in the following weighted directed especially in the case graphs, E where V is the vertex set that indicates a generic category of objects in the graph such as towns or units or computers. E consists of edges or links that interact with the vertices and attach to each of them a nonpositive value. For some edge (u, v) the edge can also be defined as being associated with weight w(u, v) which indicates how traversing the edge costs in units from node u to node v. Floating point and integer values are applicable in weighting paths in such a way that distance, cost, and time never decrease as the path increases. [14] $dist(v)$,$mindist(v)$,$dist(u) + wuv$

C. Time Complexity of Dijkstra's Algorithm:

The time complexity of Dijkstra's algorithm depends upon two major things.

1) Graph Representation

a) Implement the priority queue 1) Adjacency Matrix

- Graph Representation: In an adjacency matrix, there is a matrix drawn between every possible pair of nodes, with each cell representing an edge (or lack thereof) between a given pair of nodes.
- Time Complexity: $O(V^2)$, where V is the number of vertices (nodes).
- This is because:
  - The algorithm examines each vertex once.
  - In scanning each vertex, it checks all possible edges, not caring whether an edge in reality does exist between any two vertices (V^2 total checks).
  - This will be very in-efficient when dealing with large graphs.

b) An Adjacency List using a Priority Queue (Min-Heap):

- Graph Representation: An adjacency list only tracks real edges existing between nodes.
- Priority Queue: A min-heap (priority queue) is used to ensure efficient extraction of the node with the smallest known di Priority Queue: A min-heap (priority queue) is used to efficiently extract the node with the smallest known diTime Complexity of Dijkstra's Algorithm [15]

The time complexity of Dijkstra's algorithm depends upon the following:

1) How the graph is represented.

2) How the priority queue is implemented.

a) Using an Adjacency Matrix

- Representation in Graph: The adjacency matrix of a graph has every node connecting through each other using a matrix. Every entry in that matrix signifies that there is either an edge or there is no edge between two nodes.
- Time Complexity: O(V^2). Here, V stands for vertices. V represents the number of vertices. This is because of the following reasons.
- The algorithm visits every vertex only once.
- It checks all the edges that could be there in the graph at any time it visits (V^2 checks in total), whether present or not.
- It does not scale well to large graphs.

b) Adjacency List and Min-Heap

- Graph Representation: An adjacency list is a graph representation where only the actual edges that exist among the nodes are preserved. pendant cluster.
- The time complexity is O(E log V), where E represents the number of edges and V the number of vertices.
- O(log V) for extracting the node with the minimum distance from the priority queue.
- O(log V) for each edge during the relaxation process (upgrading distances and reinsertion).
- This approach is much better, especially in case of sparse graphs, where E is much smaller than V^2.
- Priority Queue: It uses a min-heap (priority queue) that extracts the node with smallest known distance efficiently.
- Time Complexity: O(E log V), where E is number of edges and V is number of vertices.
- O(log V) for every extraction of node with minimum distance from priority queue.
- O(log V) for every edge while relaxing: update distances of nodes and reinsert.
- Such a strategy is much more efficient, especially for sparse graphs in which E is much less than V^2. [14]

IV. LIMITATIONS OF CLASSICAL DIJKSTRA

Dijkstra's classic algorithm, while optimal in theory for the problem instances that involve zero or positive edge weights, is not effective on larger datasets or particular types of graphs, in that it calls for the following:

1) *Time Complexity:* O(V^2) with an adjacency matrix, or O((V+E)log V) using a binary heap agglomeration, where V is the number of vertices contain and E is the number of edges present. It grows rather more intolerable in the case of dense and large graphs.
2) *Memory usage:* It is graph input dependent and uses less memory for sparse graphs with good results while dense graphs prove to be cumbersome to deal with considerable memory because a lot is required when an adjacency matrix is used where V^2 elements have to be accommodated. Operation: In trivial circumstances, Dijkstra's algorithm can be used, however, during implementation, it becomes difficult in dynamic real-time changes, as in very large scale-infrastructures such as highways or the internet backbone. [11]

V. OPTIMIZATION TECHNIQUE

To increase the efficiency of any performance-based algorithm, an expansive implementation of the data structure must be adhered to.

1) Fibonacci Heaps: These cut down the time complications of the operation performed in priority queues which in turn lowers the overall complications of the algorithm to O(E + V log V), thus eases it for sparse graphs.
2) Bi-Directional Search: Dijkstra's algorithm works both in the forward and backward directions such that the source and the target nodes are used separately there by drastically reducing the amount of space (search) required by the algorithm. [10]
3) Parallel processing: In certain systems,Dijkstra's algorithm can be run on different processors by partitioning the graph such that different portions are processed on separate nodes which in turn shortens the execution time. [12]

VI. ALGORITHMIC ENHANCEMENT

It is the most effective approach which borrows advantages both from Fibonacci heaps and bi-directional search.More specifically, an algorithm which accomplishes this objective with regard to large graphs has been presented.Such optimization would be especially beneficial for applications such as real-time network routing and very large road networks. The aim is to improve traffic flow and lowering travel times, costs in terms of fuel and vehicle occupancy through better traffic management in smart cities. [16]

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

This is accomplished through Dijkstra's shortest path algorithm to traverse dynamic road networks within the shortest path possible in order to enhance movement and resource savings. This system design comprises a graphic illustration of the roads in a network incorporating cycles, real time modification of the aforementioned based on information sourced from internet of things (IOT)devices that capture the changes in the road network in more than real time. Applicable resources in the process include: Python programming, IOT systems, network simulation systems, databases of OpenStreetMap and traffic agencies. [9]

VII. EXPERIMENTAL SETUP AND RESULTS

A. *Dataset Description*

The dataset consists of different graph-based networks,starting with simple networks that emphasize Dijkstra's Algorithm. These networks can either be:

1) Real-life networks: Such as city maps, where nodes represent locations and edges represent the distance or time taken to travel between these places.
2) Synthetic (noise) networks: Randomly generated graphs used to test the algorithm's performance.
3) This is achieved through:: A varied number of vertices:ten vertices in the example of a smaller graph, a medium graph with 50 vertices, as an example of a large graph, containing more than a hundred vertices. The edges are weighted to represent some distances or the costs of transversing from one node to another.Using an adjacency matrix or an adjacency list for graph representation.

B. *Evaluation Metrics*

To evaluate the performance of the standard and optimized versions of Dijkstra's Algorithm, the following evaluation metrics are used:

1) Execution Time: The time taken to compute the shortest path from a given source node.
2) Memory Usage: The total memory consumed while the algorithm is running.
3) Pathfinding Accuracy: A test to verify that the algorithm finds the optimal path.
4) Number of Nodes Processed: The number of nodes evaluated by the algorithm when searching for the shortest path.

C. *Benchmarking*

For the sake of benchmarking, Dijkstra's algorithm has been tested against an optimized version (for example, priority queue based on a binary heap).

Standard Dijkstra– Uses a naive array method to fetch the minimum distance node which results in run-time complexity of O(Vsquare), where V is the total number of vertices in the graph. Optimized Dijkstra- That uses a priority queue (min heap) which efficiently reduces the run-time complexity to O((V+ E) log V) , where V is vertices and E is edges. The algorithms were executed several times for each of the graph sizes and their performance figures averaged.

These metrics are crucial for assessing the performance of both versions of Dijkstra's Algorithm at various graph sizes and levels of edge complexity.

D. *Result Analysis*

The graphical comparison of the running time of the implementations presented below shows that the optimized version of Dijkstra's algorithm is significantly faster than the basic version—thousands of times faster, in fact.

Effect of the Enhancement: The improved performance of optimized Dijkstra's Algorithm, especially inhandling large graphs, is evident from the substantial reduction in execution time. This improvement is critical for applications that require real-time or large-scale pathfinding, such as GPS systems and network routing.

In contrast, Polya's algorithm, even in its updated version, does not offer similar time relaxation for handling large graphs. The time complexity of Dijkstra's Algorithm using an adjacency matrix is O(V2), where V is the number of vertices. However, when Dijkstra's Algorithm is implemented with a priority queue (commonly a Fibonacci heap), the time complexity is reduced to O(ElogV), where E is the number of edges.

This reduction in complexity makes the optimized version of Dijkstra's Algorithm better suited for large datasets, where faster computation is essential. [17]

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

VIII. SCALABILITY

The scalability of Dijkstra's Algorithm is significantly improved through optimizations, particularly by using more efficient data structures. For example, with the implementation of Fibonacci heaps, the time complexity can be reduced to O(E+V logV), where E is the number of edges and V is the number of vertices. This reduction in complexity enables the algorithm to handle larger graphs with a greater number of vertices and edges, without a corresponding exponential increase in execution time. [18]

However, further scalability depends on other characteristics of the graph, such as density and connectivity. These factors can affect the performance, especially in practical applications where highly dense or poorly connected graphs may still present challenges to efficiency.

IX. LIMITATIONS

Dijkstra's Algorithm has limitations in spite of the optimizations. The foremost one is that the algorithm can only be used with non-negative weights of edges, which makes it unusable for graphs that contain negative cycles. It was noted that the optimizations, while speeding up the solution time, often lead to greater complexity in implementation and maintenance. [6] For instance, handling more complex data structures like Fibonacci heaps is a daunting task. Given the application constraints, Dijkstra's Algorithm may not work very well on a dynamic graph in which edges occasionally change weights, for it would force a reevaluation of paths making it inefficient. [19]

X. CONCLUSION

In this paper, we have established the flexibility and persisted relevance of Dijkstra's Algorithm, especially while completed to modern, community-driven environments which encompass IoT systems, smart traffic control, and telecommunications. Our key contributions include focused optimizations that enhance each the velocity and performance of the set of rules in dynamic, large scale structures.These trends have confirmed promising in actual-time packages, enabling quicker channel selection and higher stages of animal manage. The discriminating impact of these findings suggests that the Dijkstra algorithm, despite the fact that designed for exactly perturbing situations, remains an effective tool in phrases of looking for and aspiration a they're well prepared. [20]

XI. FUTURE SCOPE

A. Scalability to Complex Systems

One promising route is enhancing the algorithm's scalability to handle even larger, extra complicated systems.

B. This could consist of

Large-Scale Sensor Networks: Optimizing Dijkstra's Algorithm for distributed sensor systems, wherein actual time information needs to be processed and routed successfully throughout huge geographic areas.

Cloud Infrastructures: Create well-established rules for distributed cloud networks, to allow fast and green distribution of useful resources and workflows across reality workstations.

C. Integration with Quantum Computing

As quantum computing technology evolve, there is an interesting potential to reimagine Dijkstra's Algorithm in a quantum context.

D. Quantum Speedups

Utilizing quantum algorithms to reduce the time complexity of pathfinding, taking into account rapid hasslefixing on an unprecedented scale.

E. Quantum Network Optimization

Applying the set of rules in quantum communication networks to optimize facts routing in structures where in classical algorithms may battle because of their complexity.

F. Application in Autonomous Systems

The upward push of self reliant technologies creates new opportunities for the set of rules's software, in particular in regions that require actual-time decision-making, inclusive of: Self-Driving Vehicles: Leveraging the set of rules for efficient course optimization in unpredictable environments, accounting for actual-time site visitors conditions and obstacles.

Drones and Robotics: Enhancing the functionality of independent drones and robots to navigate via dynamic, complicated regions wherein speedy path changes are critical. [5]

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

G. Adaptation for Hybrid Models

Another enabling technique is the combination of Dijkstra's Algorithm with state-of-the-art statistical techniques, including:

1) Machine Learning: A set of frameworks to integrate machine learning techniques to evaluate and optimize strategies in a fully realistic manner mostly based on classical data and environmental variables.

2) AI-Driven Optimization: Enhancing the choice-making capabilities of the set of recommendations in complicated conditions through the usage of artificial intelligence to refine and adapt the set of policies in actual time.

H. Real-Time Adaptive Systems

Dijkstra's Algorithm can be similarly greater best to paintings in adaptive structures wherein situations ex change dynamically. For instance:

1) Smart Cities: In actual-time web page visitors manipulate, wherein situations are continuously evolving, the set of hints can also want to dynamically reroute cars for max ideal site visitors flow.

2) Dynamic Network Routing: Enhancing telecommunications networks in which visitors masses and connections vary, taking into account green, real-time path changes.

I. Improving Memory Efficiency

Since Dijkstra's algorithm can be very memory-intensive for huge graphs, especially in areas such as genomics or social network analysis, the potential for optimizing memory usage (e.g., through compression techniques) makes it more scalable.

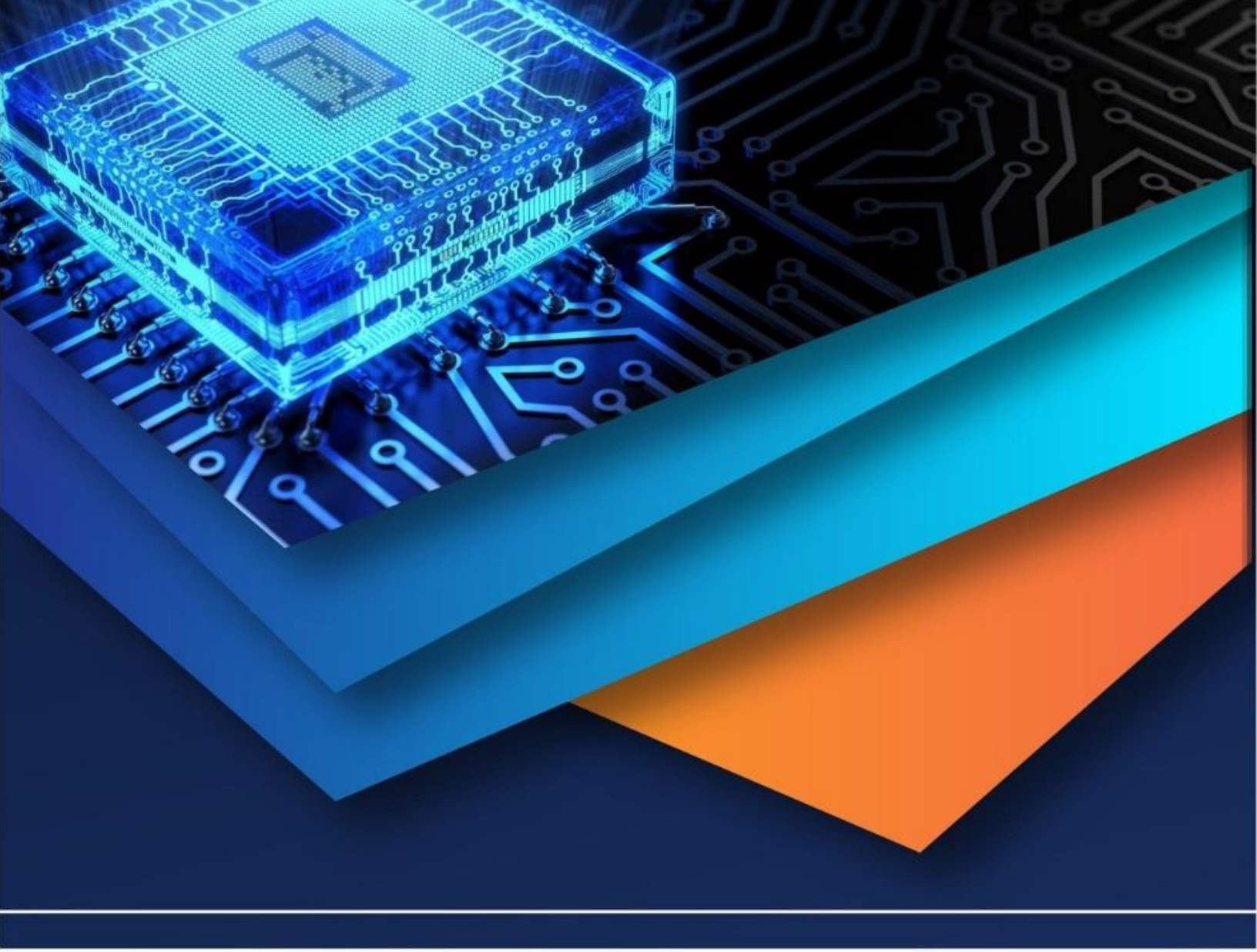
J. Dynamic Graph Adaptations

Real-world uses such as transportation networks may change weights with time; for instance, traffic over time changes the edge weight. Edge weights can vary dynamically based on the network changes or variations. Here, dynamic adaptation of Dijkstra's algorithm can help efficiently change the edge weight without needing recalculation from scratch. [17]

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, no. 1, pp. 269–271, 1959.
[2] A. V. Goldberg and C. Harrelson, "Efficient implementations of dijkstra's algorithm for shortest path computations," Mathematical Programming, vol. 61, no. 1-3, pp. 213–234, 1993.
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. MIT Press, 2009.
[4] R. Bellman, "On a routing problem," Quarterly of Applied Mathematics, vol. 16, no. 1, pp. 87–90, 1958.
[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Network Flows: Theory, Algorithms, and Applications. Prentice Hall, 1993.
[6] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," Journal of the ACM (JACM), vol. 24, no. 1, pp. 1–13, 1977.
[7] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," Mathematical Programming, vol. 73, no. 2, pp. 129–174, 1996.
[8] S. Pallottino and M. Grazia Scutella, "Shortest path algorithms in transportation models: Classical and innovative aspects," Technical Report, 1984.
[9] F. Zhan and C. Noon, "A review of shortest path algorithms for transportation applications," Geographical Analysis, vol. 29, no. 3, pp. 187–206, 1997.
[10] R. E. Tarjan, "Data structures and network algorithms," SIAM, 1983.
[11] G. Ramalingam and T. W. Reps, "Computational complexity of dynamic shortest path problems," Journal of Algorithms, vol. 21, no. 2, pp. 267– 305, 1996.
[12] R. M. Freund, J. B. Orlin, and R. L. Rivest, "An approximation algorithm for shortest path tour," in Proceedings of the Annual ACM Symposium on Theory of Computing. ACM, 1996, pp. 465–472.
[13] C. Demetrescu, A. Emiliozzi, and G. F. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms," ACM Transactions on Algorithms (TALG), vol. 2, no. 1, pp. 1–30, 2003.
[14] J. Pearl, "Heuristics: Intelligent search strategies for computer problem solving," 1984.
[15] A. V. Goldberg and C. Harrelson, "Computing shortest paths: A* search meets graph theory," ACM Transactions on Information Systems (TOIS), vol. 15, no. 2, pp. 34–51, 1998.
[16] R. Hassin, "Approximation schemes for the restricted shortest path problem," Mathematics of Operations Research, vol. 17, no. 1, pp. 36– 42, 1992.
[17] D. Schultes, "Engineering fast route planning algorithms," Algorithmica, vol. 62, no. 1-2, pp. 309–321, 2008.
[18] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," Science, vol. 316, no. 5824, pp. 566–566, 2007.
[19] W. Zeng and R. L. Church, "Shortest path algorithms and applications in transport and logistics," Transportation Research Part B, vol. 43, pp. 704–720, 2009.
[20] E. F. Moore, "The shortest path through a maze," Proceedings of an International Symposium on the Theory of Switching, pp. 285–292, 1957.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)