Dynamic Traffic Optimization in Real-Time Routing Systems

November 3, 2024

Ch.Venkata Rithish Reddy (2023csb1113) ,
Dadi Kumar Naidu(2023csb1115) ,
Polisetty Tharun Sai(2023csb1144)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Monisha Singh

Summary: The project focuses on efficiently finding paths from a single source to multiple destination nodes within a strongly connected, directed graph with positive edge weights, specifically tailored for an Optimal Route Managing System in real-time traffic scenarios. It employs the A* search algorithm enhanced by a Euclidean heuristic, effectively guiding the search process toward optimal route selection by estimating the distance to target nodes and minimizing overall path costs. A notable feature of this approach is the dynamic updating of edge weights, which reflects real-time changes in traffic conditions as nodes are visited. This adaptability ensures the algorithm accurately represents the current state of the graph, addressing fluctuations in edge weights due to varying traffic conditions or user-defined constraints.To manage these edge weight updates and delete unwanted edges based on user-defined routes to be avoided, Red-Black Trees are utilized. This enhances the flexibility of the graph's structure and improves performance during the pathfinding process, allowing for a responsive and efficient navigation experience in the context of real-time traffic management.

---

1. Introduction

1.1. Project Objectives

1. **Develop a pathfinding approach** that balances efficiency with near-optimal paths by allowing node revisits and dynamically updating edge weights based on real-time traffic conditions as nodes are visited, utilizing Red-Black Trees for efficient management of these updates.
2. **Combine principles from TSP and the A* algorithm** to guide path selection, utilizing euclidean heuristic that consider traffic data to prioritize promising paths while maintaining flexibility in traversal.
3. **Explore additional optimizations** through refined heuristics, bidirectional search, and clustering to streamline computation, enhance path quality, and improve responsiveness to fluctuating traffic patterns.

1.2. Core Algorithm

The core algorithm integrates aspects of both A* and TSP principles. It begins by identifying the nearest unvisited destination from the source using a TSP nearest neighbor heuristic. The A* algorithm is then applied to determine the shortest path between the source and this nearest destination. Upon reaching this destination, it becomes the new source, and the process repeats until all destinations are visited. This combined approach leverages the nearest neighbor heuristic to guide each step while utilizing A*'s heuristic-based efficiency to prioritize paths. Red-Black Trees are employed to manage dynamic edge weight updates, ensuring that changes reflecting real-time traffic conditions are efficiently handled. This makes the method well-suited for scenarios where computational feasibility is prioritized over strict TSP optimality, enhancing the algorithm's responsiveness and overall performance in managing real-time routing challenges.

1.3. Dynamic Edge Weight Adjustment

To simulate real-world changes, edge weights are updated in real-time as nodes are visited. This mechanism accounts for conditions that may change during traversal, such as varying travel times, resource consumption, or node priorities. The adjustments ensure that paths evolve, adapting to the updated weights and ultimately leading to efficient yet flexible routes.

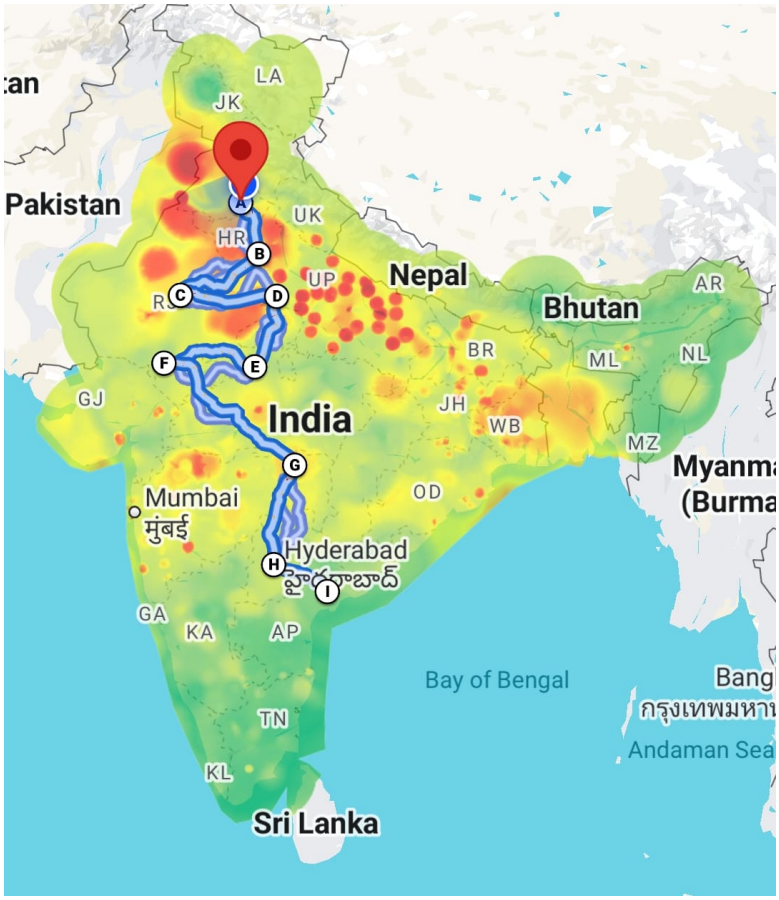

Figure 1: Dynamic Route Optimization in Action.

1.4. User-Centric Navigation Adjustments

To enhance the adaptability of the routing system, the algorithm incorporates a user-driven mechanism for avoiding certain paths. As part of the project, users can specify routes they wish to avoid, prompting the system to dynamically delete these nodes from the graph. This deletion process is efficiently managed using Red-Black Trees, which facilitate quick updates and maintain the graph's structure as nodes are removed. This feature not only personalizes the routing experience but also helps in optimizing the pathfinding process by excluding less desirable or congested routes. By actively engaging users in the decision-making process, the system can better align with real-time traffic conditions and preferences, ensuring more efficient and relevant pathfinding outcomes.

1.5. Computational Efficiency

This combination of techniques not only addresses the need for accurate pathfinding but also prioritizes computational efficiency in complex scenarios. The design allows for future enhancements, including the integration of improved heuristics, bidirectional search strategies, and clustering techniques. By incorporating these elements, the project aspires to provide a robust solution for effectively navigating intricate directed graphs while adapting dynamically to user requirements.

1.6. Implementation Details

The project is implemented in C, with a focus on optimizing memory and time efficiency. Data structures such as adjacency lists and priority queue support quick access and updates to nodes and edges, while specialized functions handle dynamic weight adjustments.
Data Structures: Adjacency lists represent the graph structure, while priority queue support efficient edge selection during traversal. Weight Update Mechanism: Functions handle dynamic adjustments, recalculating paths as weights change. Optimization Techniques: Heuristics guide path selection.

1.7. Experimental Results and Evaluation

The solution was tested on various graph sizes and configurations, with results demonstrating effective pathfinding within constraints. Performance metrics include the number of operations, time complexity, and path efficiency, showing an overall balance between computational cost and path quality.
Operation Count: Reduction in operations due to heuristic-based pruning. Execution Time: Achieves suboptimal paths within practical timeframes.

2. Equations

$$\begin{cases} & h(n) = \sqrt{(x_{\text{goal}} - x_n)^2 + (y_{\text{goal}} - y_n)^2}, & \text{(1a)} \\ & f(n) = g(n) + h(n), & \text{(1b)} \\ \text{where} & f(n) \text{ is the total estimated cost of the cheapest solution through node } n. & \text{(1c)} \end{cases}$$

In detail:

- $f(n)$ is defined as the total estimated cost of the cheapest solution that passes through node n. This means it combines:
  - $g(n)$, which is the known cost to reach the node n from the start node.
  - $h(n)$, the estimated cost from node n to the goal.

3. Figures and Algorithms

3.1. Figures

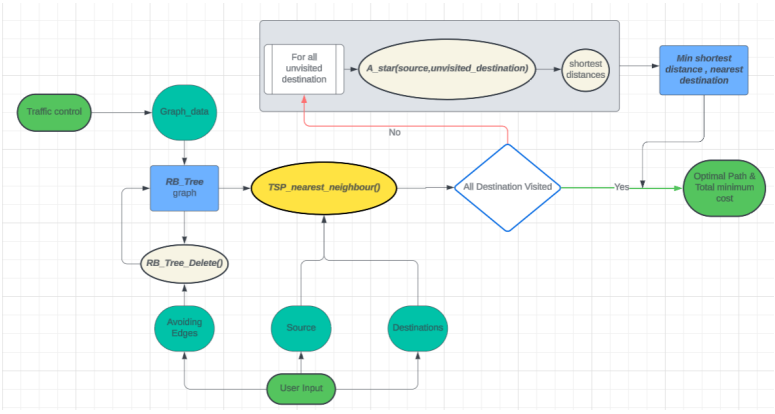

Figure 2: Flowchart.

Our project focuses on identifying the nearest destination from the current node by utilizing the A* search algorithm. This process is repeated iteratively until all destination nodes have been visited. The flowchart below provides a clearer and more representation of this approach, illustrating the inputs and outputs involved in the overall process.

3.2. Algorithms

->**A* Algorithm Explanation:** The algorithm initializes all nodes with infinite distance, marking them as unvisited, while setting the source node's distance to zero. Using a priority queue, nodes are processed based on an f-cost, which combines the actual path cost (g-cost) and a heuristic estimating the distance to the target. For each node u, neighboring nodes v are evaluated by calculating a tentative g-cost. If this new path offers a shorter route to v, the distance and backtracking reference for v are updated, and v is reprioritized in the queue based on its updated f-cost. This process repeats until the target is reached, allowing for early exit and efficient route selection. In the context of real-time traffic management, this algorithm adapts quickly to traffic conditions by dynamically updating edge weights, ensuring the selection of optimal paths that minimize travel time across complex, strongly connected road networks.

Algorithm 1 A* Algorithm

Procedure a_star(source, target, dist, prev)
for $i = 0$ **to** $n - 1$ **do**
    $dist[i] = INF$, $prev[i] = -1$
end for
$dist[source] = 0$
Let pq be a PriorityQueue with size 0
pq_push(pq, heuristic(source, target), 0, source)
while !pq_empty(pq) **do**
    Let $current = pq\_pop(pq)$
    $u = current.city$
    if $u == target$ **then**
        return {Early exit if target is reached}
    end if
    for $i = 0$ **to** $graph[u].size - 1$ **do**
        Let $v = graph[u].edges[i].city$
        Let $weight = graph[u].edges[i].weight$
        $tentative\_g\_cost = dist[u] + weight$
        if $tentative\_g\_cost < dist[v]$ **then**
            $dist[v] = tentative\_g\_cost$
            $prev[v] = u$
            $f\_cost = tentative\_g\_cost + heuristic(v, target)$
            pq_push(pq, f_cost, tentative_g_cost, v)
        end if
    end for
end while

The time complexity of the A* algorithm is $\mathbf{O}(V^2 + E)$.
where

- $V$is no of vertices
- E.is no of edges

The time complexity of O(**V2**+**E**) arises due to the following reasons:
->**Priority Queue Operations**: Each pq pop operation in this code scans the entire queue to find the minimum, taking O(V) per pop. In the worst case, all V nodes are processed, leading to $O(V^2)$ for all pops combined.
->**Exploring Neighbors**: For each node, examining all its edges (neighbors) contributes O(E) to the complexity, as all edges are processed once.
->**Heuristic Computation**: Calculating the heuristic (Euclidean distance) is O(1) per call, so this part doesn't impact the overall complexity significantly.
->**Overall Complexity**: Combining all operations, the algorithm's time complexity becomes O($V^2 + E$).

TSP Using A* Search Algorithm Explanation: The *tsp_a_star_based* procedure computes an optimal route from a source city to multiple destinations, combining principles from the Traveling Salesman Problem (TSP) and A* pathfinding. Starting at the source, it iteratively selects the nearest unvisited destination by using A* to find the shortest path, updates the total path cost, marks destinations as visited, and displays the path. After the first selection, edge weights are randomized to simulate real-time conditions, with updates saved to a file for dynamic route adjustments. This process continues until all destinations are visited, yielding the total minimal path cost. By integrating A* efficiency with adaptive TSP logic, this approach supports real-time, traffic-sensitive navigation.

Algorithm 2 TSP using A* Search

Procedure tsp_a_star_based(source, destinations, num_destinations, filename)
$total\_cost = 0$
$visited[MAX\_CITIES] = \{0\}$
$current\_city = source$
$visited[source] = 1$
print("Path: ", current_city)
$first\_selection\_done = 0$
for $count = 0$ **to** $num\_destinations - 1$ **do**
  $nearest\_city = -1$, $min\_distance = INF$
  $dist[MAX\_CITIES], prev[MAX\_CITIES]$
  for $i = 0$ **to** $num\_destinations - 1$ **do**
    if !visited[destinations[i]] **then**
      a_star(current_city, destinations[i], dist, prev)
      if $dist[destinations[i]] < min\_distance$ **then**
        $min\_distance = dist[destinations[i]]$
        $nearest\_city = destinations[i]$
      end if
    end if
  end for
  if $nearest\_city == -1$ **then**
    print("Error: No unvisited destinations remaining.")
    return
  end if
  if $current\_city \neq nearest\_city$ **then**
    print_path(current_city, nearest_city, prev)
  end if
  $total\_cost+ = min\_distance$
  $visited[nearest\_city] = 1$
  if !first_selection_done **then**
    randomize_edge_weights()
    save_weights_to_file(filename)
    $first\_selection\_done = 1$
  end if
  $current\_city = nearest\_city$
end for
print("Total minimal path cost: ", total_cost)

->The time complexity of *tsp_a_star_based* is as follows:
->As the algorithm focus on finding nearest neighbour from the source city in a greedy approach the function *tsp_a_start_based* run as follows:
->For a source finding the nearest neighbour includes calling the a* shortest path algorithm for all the unvisited destinations.
->In the first iteration of the outer loop (when no destinations have been visited yet), the inner loop will check all D destinations.

-> In the second iteration, there will be $D-1$ destinations left, and so forth, until in the last iteration, only 1 destination remains.
–>Thus the total number of times function calling the a* is $\frac{D(D+1)}{2}$ which makes the overall complexity $O\left(\frac{D(D+1)}{2} \cdot (V^2+E)\right)$

3.3. Overall Complexity

Operation	**Time Complexity**
Taking Graph Input	$O(V)$
Construction of Red-Black Tree and removing avoided edges	$O(K \log E)$
Finding optimal path through `tsp_a_star_based`	$O\left(\frac{D(D+1)}{2} \cdot (V^2+E)\right)$
Overall Complexity	$O\left(D^2 \cdot (V^2+E)\right)$

Table 1: Time Complexities of Different Operations

where

- K is no of avoided edges
- $Note$: The number of destinations D in real time practices always less than a small constant.
- So considering D^2 as a constant reduces the overall complexity to O(V^2+E)

4. Some further useful suggestions

4.1. Incorporating Real-Time Traffic Data for Edge Weight Adjustments

In addition to the existing features of the Optimal Route Managing System, a further modification to the project involves enhancing the edge weight updating mechanism. Currently, the algorithm updates edge weights randomly to simulate changes in traffic conditions. However, with access to real-time traffic data, the project can transition to updating edge weights based on actual traffic conditions. This enhancement will ensure that the pathfinding algorithm reflects more accurate and relevant data, leading to improved route optimization and responsiveness to current traffic scenarios. By aligning the edge weight updates with real-time traffic information, the system will provide users with more reliable and efficient navigation solutions in dynamic urban environments.

5. Conclusions

This project offers a practical approach to multi-destination pathfinding in directed graphs with dynamic weights. By allowing node revisits and employing flexible optimization strategies, the algorithm successfully balances efficiency and path quality. Future improvements could explore additional heuristics or alternative methods of dynamic weight management to further enhance performance.

6. Bibliography and citations

Acknowledgements

->We extend our sincere gratitude to our course instructor, Dr. Anil Shukla, and our teaching assistant, Monisha Singh, for their invaluable guidance throughout the course of this project. Their support and expertise were instrumental to our progress and learning.

-> We have referenced Euclidean heuristic from an article published in internet archeive by Peter Hart, Nils Nilsson and Bertram Raphael who published a star algorithm.[2]

->We have referenced the Red-Black Tree algorithm from Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein[1]

->Our exploration of advanced routing algorithms led us to discover this particular model, which we learned about through the comprehensive resources provided by Graphhopper.[4]

->All our RedBlack tree algorithms that we have used are from the link in this reference[3]

References

[1] Thomas H. Cormen. *Introduction to algorithms*. Prentice Hall India, 2009.

[2] Peter Hart. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 2005.

[3] Algorithms of Red Black Trees.

[4] Graphhopper. A route planning API.

.1. Appendix A

Manhattan Distance Heuristic: The Manhattan distance heuristic is commonly used in grid-based path-finding, such as in games or robotics. It calculates the sum of the horizontal and vertical distances from the current node to the goal node, assuming only orthogonal movements are allowed.
Diagonal Distance Heuristic: This heuristic is used in grid-based environments where diagonal moves are allowed. It estimates the distance by considering both horizontal and vertical moves, as well as diagonal moves, using a combination of these distances.
Octile Distance Heuristic: The octile distance heuristic is a variation of the diagonal distance heuristic that assumes diagonal moves are more costly. It uses a weighted combination of horizontal, vertical, and diagonal distances, reflecting the higher cost of diagonal moves.

.2. Appendix B

Euclidean Distance Heuristic

The Euclidean distance heuristic is frequently used in pathfinding and optimization problems where the shortest direct path is desired. This heuristic calculates the straight-line distance between two points in Euclidean space, considering their Cartesian coordinates (x, y, and optionally z for 3D space). Euclidean distance is particularly effective for problems that assume a flat plane or space without obstructions or non-linear paths.

Advantages of Euclidean Distance Heuristic:

1. **Precision in Pathfinding:** The Euclidean distance heuristic provides precise estimates of the shortest path in environments where direct line-of-sight paths are possible, making it ideal for robotics and indoor navigation.
2. **Simplicity and Efficiency:** The simplicity of Euclidean distance calculation makes it computationally efficient, reducing overhead in real-time applications such as game AI and autonomous vehicles.
3. **Smooth Distance Gradients:** The Euclidean heuristic generates smooth gradients, allowing for more natural and continuous path selection in 2D and 3D spaces, useful for applications that require fine control.
4. **Applicability in Various Fields:** It is widely applicable in fields like computer graphics, robotics, and physics simulations, providing reliable distance measurements in flat or low-curvature environments.
5. **Direct Use in A* Algorithms:** In algorithms like A*, Euclidean distance is a common choice for heuristics when the search area resembles a Euclidean plane, enhancing the accuracy of shortest-path estimations.