Design and Solution Overview Traffic Optimization System

Elbek Tursunov

Latvijas Universitate
Datoriku Fakultate

CID: DatZ3168 DSA
Instructor: Professor Guntis Arnicans
SID: et19042@students.lu.lv
Fall 7th, 4th year

January 16, 2024

Abstract

Pathfinding algorithms play a crucial role in various applications, ranging from robotics and artificial intelligence to network routing and video games. In the realm of traffic optimization systems, the significance of pathfinding algorithms cannot be overstated. This paper explores the design and solution overview for "Traffic Optimization System - Boss" study project.
The system utilizes real-time data collected by public services, encompassing critical parameters such as maximum car capacities, current traffic levels, peak traffic constraints, delays, and minimum travel times for specific street segments.

Requirements

Key Objectives

1. **Optimize Traffic Routes:** Develop a sophisticated system to optimize traffic routes for "The Boss" in Riga, Latvia, ensuring efficient and timely travel.
2. **Real-time Data Utilization:** Leverage real-time data from public services to obtain crucial information about street segments, including maximum car capacity, current traffic, peak capacity, and additional delay.
3. **Continuous Route Planning:** Implement a responsive algorithm capable of continuous route planning, dynamically adjusting to changing traffic conditions.

Assumptions

1. The Boss's vehicle is equipped with a GPS system that consistently transmits coordinates to a central server.
2. Public services reliably provide detailed data on street segments.
3. The central server can receive destination information and promptly return an optimized route based on real-time data.
4. Intersections do not contribute to time or delay.
5. Street segments and their properties can change dynamically.

Input File

Structure

1. **Line 0 [Optional]:** Optional flag which indicates update request on segments' data. If flag is set then following after flag segments need to be updated with new values (UPD: [Optional]).
2. **Line 1:** Number of street segments (N).
3. **Next N lines:**
   - Segment ID (1 to N).
   - Maximum number of cars (Ci).
   - Current number of cars (Ni).
   - Maximum capacity during peak traffic (Mi).
   - Delay time for each car above Ci (Di).
   - Minimum time without obstacles (Ti).
   - List of Segment IDs that are directly connected to the current segment (ConSeg).

Example

```
4
1 10 5 2 15 3 2 3
2 8 7 1 12 2 4
3 5 4 3 8 1 2
4 12 6 4 18 5 1
4 1
```

Output File

Structure

1. **Line 1:** A single value representing the estimated arrival time at the destination.
2. **Line 2:** Segment ID and direction for each segment in the optimized route.

Example

```
1
4 -> 1
```

Theory Overview

1. Greedy Algorithm

- **Idea:** Always choose the segment with the least time to minimize overall travel time.
- **Advantages:** Simplicity and quick implementation.
- **Disadvantages:** Potential oversight of long-term delays and dynamic changes in traffic.
- **Time Complexity:** The Greedy Algorithm exhibits a time complexity of $O(V^2)$ per iteration, where V signifies the number of street segments. This quadratic complexity stems from the algorithm's iterative selection of the subsequent segment with the minimum travel time.
- **Space Complexity:** The space complexity is $O(V)$, necessitated by the storage of distances associated with each street segment. The linear space complexity aligns with the algorithm's basic storage requirements.

2. Dijkstra's Algorithm

- **Idea:** Find the shortest path based on the sum of travel times.
- **Advantages:** Considers the entire route, adaptable to dynamic changes.

- **Disadvantages:** Computationally expensive for large networks.
- **Time Complexity:** Dijkstra's Algorithm demonstrates a time complexity of $O((V + E)\log V)$, where V and E denote the number of street segments and edges, respectively. The logarithmic factor is introduced by the priority queue operations during exploration.
- **Space Complexity:** The space complexity amounts to $O(V)$, accounting for the storage of distances and the utilization of a priority queue. This linear space complexity reflects the algorithm's memory requirements.

3. A* Algorithm

- **Idea:** Combines aspects of Dijkstra's and Greedy algorithms using heuristic information.
- **Advantages:** More efficient in finding the optimal route by considering both cost and heuristic.
- **Disadvantages:** Requires a well-defined heuristic function.
- **Time Complexity:** Similar to Dijkstra's Algorithm, the A* Algorithm exhibits a time complexity of $O((V + E)\log V)$. The logarithmic factor arises from priority queue operations during segment exploration, considering both cost and heuristic.
- **Space Complexity:** The space complexity is $O(V)$, encompassing the storage of distances, heuristics, and the use of a priority queue. This linear space complexity aligns with the algorithm's storage demands.

4. Bellman-Ford Algorithm

- **Idea:** Find the shortest paths from a single source to all other vertices, even with negative edge weights.
- **Advantages:** Handles negative weights, adaptable to various scenarios.
- **Disadvantages:** Inefficient for dense graphs.
- **Time Complexity:** The Bellman-Ford Algorithm manifests a time complexity of $O(V \cdot E)$, where V and E represent the number of street segments and edges, respectively. The quadratic factor emerges from the multiple relaxation steps, iterating through all segments and edges.
- **Space Complexity:** The space complexity is $O(V)$, accounting for the storage of distances. This linear space complexity encapsulates the algorithm's memory utilization.

5. Floyd-Warshall Algorithm

- **Idea:** Find the shortest paths between all pairs of vertices in a weighted graph.
- **Advantages:** Handles negative weights, computes all-pair shortest paths.
- **Disadvantages:** Inefficient for large graphs due to its cubic time complexity.
- **Time Complexity:** The Floyd-Warshall Algorithm displays a cubic time complexity of $O(V^3)$, where V denotes the number of street segments. The algorithm's nested loops contribute to the computation of shortest paths between all pairs of segments.
- **Space Complexity:** The space complexity amounts to $O(V^2)$, incorporating the storage of distances for each pair of segments. This quadratic space complexity aligns with the algorithm's memory requirements.

6. Bidirectional Search

- **Idea:** Simultaneously perform a forward and backward search to find a meeting point.
- **Advantages:** Reduces time complexity for certain scenarios, particularly in unweighted graphs.
- **Disadvantages:** Requires knowledge of the goal vertex in advance.
- **Time Complexity:** The Bidirectional Search introduces a time complexity of $O(b^{d/2})$, where b represents the branching factor and d signifies the distance from the source to the destination. This complexity reflects the reduction in the effective search space achieved by simultaneously exploring segments from both directions.
- **Space Complexity:** The space complexity is $O(V)$, encompassing the storage of distances for each street segment. This linear space complexity aligns with the algorithm's memory utilization.

Selected Approach

Approach: A* Algorithm

The algorithm is an extension of Dijkstra's algorithm and introduces a heuristic to guide the search efficiently. A* uses a combination of the actual cost to reach a node from the start node (known as ″g-cost″) and an estimated cost from the current node to the goal node (known as ″h-cost″ or heuristic cost). The total cost of a node is the sum of these two costs: f(n)=g(n)+h(n).

A* maintains two lists: an open list and a closed list. The open list contains nodes that are candidates for exploration, while the closed list includes nodes that have already been considered. The algorithm repeatedly selects the node with the lowest total cost from the open list, explores its neighbors, and updates their costs. This process continues until the goal node is reached or the open list becomes empty.

The heuristic function is crucial in A* as it guides the algorithm toward the goal efficiently. It should be admissible (never overestimating the true cost to reach the goal) to guarantee the optimality of the solution.

The A* algorithm is known for its effectiveness in finding optimal paths while being computationally efficient, especially when a good heuristic is available.

Data Structures

1. Priority Queue for efficient extraction of the segment with the minimum f-value.

2. Distance array to store the current distance from the starting segment to each segment.

3. Heuristic array to store estimated remaining distance to the destination for each segment.

Algorithm 1 A* Algorithm

```
1:  procedure A_STAR(start, goal)
2:      $openSet \leftarrow$ Priority Queue()
3:      $closedSet \leftarrow$ Set()
4:      $startNode \leftarrow$ Node($start$)
5:      $goalNode \leftarrow$ Node($goal$)
6:      $openSet$.add($startNode$)
7:      while $openSet$ is not empty do
8:          $currentNode \leftarrow openSet$.pop()        ▷ Get the node with the lowest $f$-score
9:          if $currentNode == goalNode$ then
10:             return RECONSTRUCTPATH($currentNode$)
11:         end if
12:         $closedSet$.add($currentNode$)
13:         for each $neighbor$ in $currentNode$.neighbors do
14:             if $neighbor$ in $closedSet$ then
15:                 continue
16:             end if
17:             $tentative\_gScore \leftarrow currentNode.gScore +$ Distance($currentNode, neighbor$)
18:             if $neighbor$ not in $openSet$ or $tentative\_gScore < neighbor.gScore$ then
19:                 $neighbor$.cameFrom $\leftarrow currentNode$
20:                 $neighbor.gScore \leftarrow tentative\_gScore$
21:                 $neighbor.fScore \leftarrow neighbor.gScore +$ Heuristic($neighbor, goalNode$)
22:                 if $neighbor$ not in $openSet$ then
23:                     $openSet$.add($neighbor$)
24:                 end if
25:             end if
26:         end for
27:     end while
28:     return FAILURE
29: end procedure
30: procedure RECONSTRUCTPATH(node)
31:     $path \leftarrow$ []
32:     while $node$ is not null do
33:         $path$.add($node$)
34:         $node \leftarrow node$.cameFrom
35:     end while
36:     return REVERSE($path$)
37: end procedure
```

Initial Assessment of the Implementation

Challenges

- Dynamic updates of traffic data.
- Real-time coordination between the central server and public services.

Ease of Implementation

Moderate difficulty due to the need for dynamic updates and real-time data.

Hardware Requirements

Standard hardware with an internet connection for real-time updates.

Response Time

Expected to be responsive, taking into account real-time traffic updates.

Conclusion

The analysis of pathfinding algorithms reveals a nuanced landscape where the selection of the most suitable algorithm depends on the specific requirements and characteristics of the problem at hand. Dijkstra's Algorithm stands out for its optimality and efficiency in non-negative weighted graphs, but its limitations in the face of negative weights prompt consideration of alternatives. A* Algorithm, with its combination of Dijkstra's principles and heuristic guidance, proves to be a versatile choice across diverse applications, yet its efficacy is contingent upon the quality of the heuristic employed.

Breadth-First Search (BFS) excels in unweighted graphs, guaranteeing the shortest path, but its applicability diminishes in scenarios with weighted edges. Depth-First Search (DFS), while memory-efficient and useful for certain tasks like cycle detection, lacks the optimality guarantees essential for pathfinding applications. Bidirectional Search emerges as an intriguing solution, leveraging the efficiency gained by exploring from both ends simultaneously, yet it demands an accurate heuristic for optimal performance. The Bellman-Ford Algorithm, designed to handle negative weights and detect negative cycles, provides a crucial tool for scenarios where such elements are present. However, its inherent inefficiency in comparison to Dijkstra's Algorithm in non-negative graphs necessitates careful consideration.

In practice, the choice of algorithm hinges on the nature of the graph, computational resources, and the necessity for optimality. As technology advances and problem domains evolve, ongoing research into enhancing existing algorithms and developing novel approaches will further refine the field of pathfinding, addressing the ever-growing complexity of modern applications.

But at momentum, In conclusion, the chosen A* algorithm presents a balanced and efficient solution for optimizing traffic routes for "The Boss" in Riga. The incorporation of heuristic information enhances overall performance, and while implementation poses moderate challenges, it is anticipated to yield optimal results.

Source Code

The assumption is that an update event will activate another interface, and this triggered interface will subsequently invoke an algorithm to recalculate the optimal path based on the updated data. Currently, the repository contains only a basic implementation of the A* algorithm, lacking functionality for updating segment data.

Repository link: https://github.com/kelbudiul/datz3168practicaltasks/

References

R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968.

Wikipedia. A* search algorithm, 2023a. URL `https://en.wikipedia.org/wiki/A*_search_algorithm`.

Wikipedia. Bellman–ford algorithm, 2023b. URL `https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm`.

Wikipedia. Floyd–warshall algorithm, 2023c. URL `https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm`.