A blue and white cover with a map

Description automatically generated

## **Problem Statement**

**Task 01 - How does Google Maps handle shortest path calculations on a vast graph representing the global traffic network?**

In the previous solo project, students were asked to work with a small graph of bus traffic in Ho Chi Minh City, where each node represents a bus stop. This graph contains about 4,500 nodes.

Now, imagine a much larger graph of the global traffic network, spanning multiple cities, countries, and continents, and incorporating various modes of transportation such as planes, buses, trains, taxis, motorcycles, and bicycles, all with varying timestamps.

How can Google Maps manage this enormous graph? Would they simply run a Dijkstra algorithm on billions of nodes and edges and return the answer in just a second?

For this task, students are required to research this problem and write a report on it. Using this knowledge, we will later delve deeper into implementing the code.

Contents

[**Problem Statement** 2](#_Toc169273555)

[**1.** **Introduction** 4](#_Toc169273556)

[**2.** **Some techniques used in Google Maps** 5](#_Toc169273557)

[*2.1. Hierarchical Graphs* 5](#_Toc169273558)

[*2.2. Heuristic Search Algorithms* 6](#_Toc169273559)

[*2.3. Real-time data integration with dynamic graph updates* 8](#_Toc169273560)

[*2.4. Precomputation and Caching* 8](#_Toc169273561)

[**3.** **Heuristic functions**  9](#_Toc169273562)

[*3.1. Exact Heuristics* 9](#_Toc169273563)

[*3.2. Approximation Heuristics* 9](#_Toc169273564)

[**4.** **Conclusion** 12](#_Toc169273565)

[**5.** **References** 12](#_Toc169273566)

## **Introduction**

Google Maps is a unique web-based mapping service brought to you by the tech giant, Google. It offers satellite imagery, aerial photography, street maps, 360° panoramic views of streets, real-time traffic conditions, and route planning for traveling by foot, car, bicycle, or public transportation.

If you have driven on unknown roads, or even known ones, you must surely have depended on some kind of navigation system. Long gone are the days when you would have to memorize street maps before traveling to a new area. In this age of technology-driven navigation software, Google Maps has become explosively popular.

It is a great help for navigating new roads and also predicting the traffic density of an area as well as finding the best alternative route for your destination. The offline feature even helps you in areas with poor or no connectivity.

The Google Maps’s network encompasses multiple cities, countries, and continents, integrating various modes of transportation such as airplanes, buses, trains, taxis, motorcycles, and bicycles. The underlying complexity of this problem is magnified by the sheer scale of the graph representing this network, which consists of billions of nodes and edges, each with its own dynamic attributes such as timestamps, traffic conditions, and transportation schedules.

To understand how Google Maps handles the shortest path calculations in such an expansive and intricate graph, it is essential to explore the strategies and algorithms employed. Unlike a simple application of Dijkstra's algorithm on a small graph, the global traffic network requires advanced techniques to deliver results swiftly and accurate.

## **Some techniques used in Google Maps**

Google Maps employs a combination of advanced algorithms, hierarchical data structures, real-time data integration, and distributed computing to handle the shortest path calculations on a vast global traffic network:

### *2.1. Hierarchical Graphs*

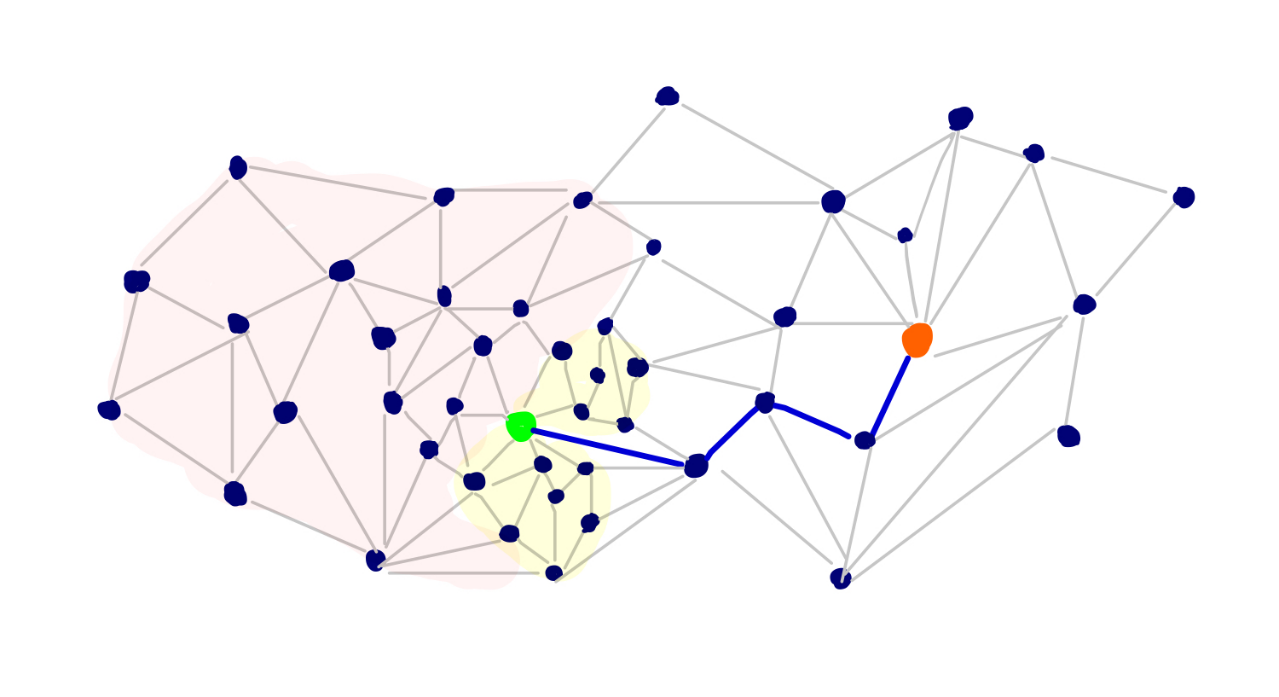
Google Maps uses hierarchical representations of the road network to simplify the search space. The idea is to categorize roads into different levels of hierarchy, such as local roads, arterial roads, and highways. Shortest path calculations can then be performed more efficiently by:

* **Contracting** the lower levels of the hierarchy, which involves simplifying or abstracting less important roads.
* **Navigating** primarily on higher-level roads for long-distance travel and only delving into lower-level roads near the start and end points of a journey.

In Google Maps, they may utilize a preprocessing technique called **Contraction Hierarchies** (CH). It involves preprocessing the graph by adding **shortcuts** that summarize paths through lower-level nodes. During query time, the algorithm can skip over large parts of the graph using these shortcuts, significantly speeding up the search.

#### For example:

Consider the example graph below, with our source at the green node and our destination the orange node:



If we ran Dijkstra’s algorithm from our source, we would hope to avoid settling nodes and relaxing edges in the red highlighted region given the location of our target node. But we’d also want to avoid unnecessarily searching the yellow shaded region. Here, we probably have many vertices with low distances that are in the general direction of our target, but they are inconsequential to the actual shortest path.

Knowing the shortest path between two nodes makes it easy to point out unnecessarily searched portions of the graph, but the natural intuitions about the importance of certain edges over others are important for building faster algorithms.

#### How Much Speedup is Possible?

A series of shortest path queries were made using various algorithms, and the average number of vertices scanned (on the receiving end of an edge relaxation) and the average time per query were recorded. The tests were run on a Western Europe road network graph:

| *Algorithm* | *# of Scanned Vertices* | *Query Time (microseconds)* |
| --- | --- | --- |
| Dijkstra | 9326696 | 2195080 (~ 2 seconds) |
| Contraction Hierarchies | 280 | 110 |

Clearly, it is possible to engineer techniques that only need to process a tiny fraction of the original graph during queries.

### *2.2. Heuristic Search Algorithms*

Google Maps may use A\* search algorithm to speed up the searching process rather than Dijkstra’s algorithms.

***So, what is A\* Search Algorithm?***   
A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.  
Informally speaking, A\* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.   
And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently.

*Now, we consider an example of using A\* search algorithms:*

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A\* Search Algorithm comes to the rescue.  
What A\* Search Algorithm does is that at each step it picks the node according to a value-‘**f**’ which is a parameter equal to the sum of two other parameters – ‘**g**’ and ‘**h**’. At each step it picks the node/cell having the lowest ‘**f**’, and process that node/cell.  
We define ‘**g**’ and ‘**h**’ as simply as possible below  
**g** = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.   
**h** = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in section **3**.

Algorithms:

// A\* Search Algorithm  
1. Initialize the open list   
2. Initialize the closed list  
 put the starting node on the open   
 list (you can leave its **f** at zero)

🡪 similar to Dijkstra’s Algorithm.  
3. while the open list is not empty  
 a) find the node with the least **f** on   
 the open list, call it "q"  
 b) pop q off the open list  
   
 c) generate q's successors and set their   
 parents to q  
   
 d) for each successor  
 i) if successor is the goal, stop search  
   
 ii) else, compute both **g** and **h** for successor  
 successor.**g** = q.**g** + distance between   
 successor and q  
 successor.**h** = distance from goal to   
 successor (This can be done using many   
 ways, we will discuss three heuristics-   
 Manhattan, Diagonal and Euclidean   
 Heuristics)  
   
 successor.**f** = successor.**g** + successor.**h**  
 iii) if a node with the same position as   
 successor is in the OPEN list which has a   
 lower **f** than successor, skip this successor  
 iV) if a node with the same position as   
 successor is in the CLOSED list which has  
 a lower **f** than successor, skip this successor  
 otherwise, add the node to the open list  
 end (for loop)  
   
 e) push q on the closed list  
 end (while loop)

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A\* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.



### *2.3. Real-time data integration with dynamic graph updates*

Google Maps continuously integrates real-time traffic data, collected from various sources such as GPS devices, mobile phone movements, and user-reported incidents. This data is used to update edge weights in the graph dynamically, reflecting current traffic conditions.

Techniques such as **Dynamic Shortest Path (DSP)** algorithms are employed to handle real-time updates efficiently. These algorithms can adjust precomputed paths and graph structures quickly in response to changes in traffic conditions.

### *2.4. Precomputation and Caching*

For frequently queried routes, Google Maps precomputes the shortest paths and stores them in a cache. This allows for instant retrieval of route information without having to recompute the path. They may use Landmark-Based Routing for identifying a set of key landmarks and precomputing the shortest paths between all pairs of landmarks. During a query, the algorithm can quickly find the shortest path by routing through these landmarks.

## **Heuristic functions**

In section 2, we discuss about h – a heuristic function that estimated movement cost to move from that given square on the grid to the final destination. *So, how to calculate h?*

We can do: either calculate the exact value of h (which is certainly time consuming) or approximate the value of h using some heuristics (less time consuming).  
We will discuss both of the methods.

*3.1. Exact Heuristics*We can find exact values of h, but that is generally very time consuming.  
Below are some of the methods to calculate the exact value of h.  
- Pre-compute the distance between each pair of cells before running the A\* Search Algorithm.  
- If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the [Euclidean Distance](https://en.wikipedia.org/wiki/Euclidean_distance)

*3.2. Approximation Heuristics*  
There are generally three approximation heuristics to calculate h:

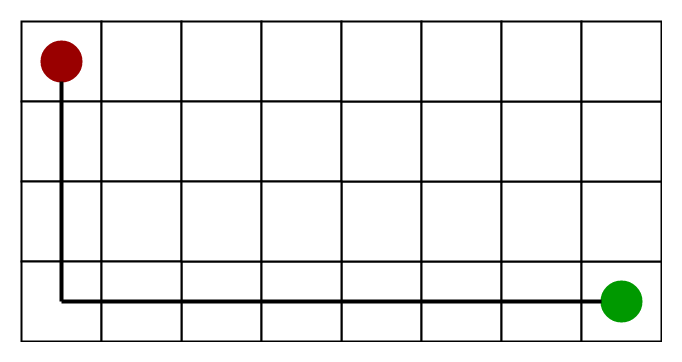
**3.2.1. Manhattan Distance**

* It is nothing but the sum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively, i.e.,

**h** = abs (current\_cell.x – goal.x) +   
 abs (current\_cell.y – goal.y)

* When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



**3.2.2 Diagonal Distance**

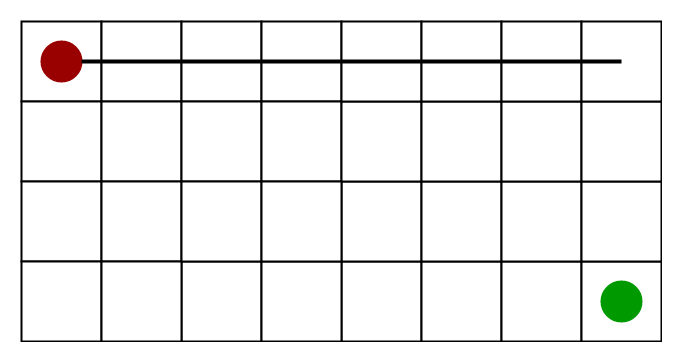
* It is nothing but the maximum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively, i.e.,

dx = abs(current\_cell.x – goal.x);  
dy = abs(current\_cell.y – goal.y);  
**h** = D \* (dx + dy) + (D2 - 2 \* D) \* min(dx, dy);

where D is length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2) ).

* When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



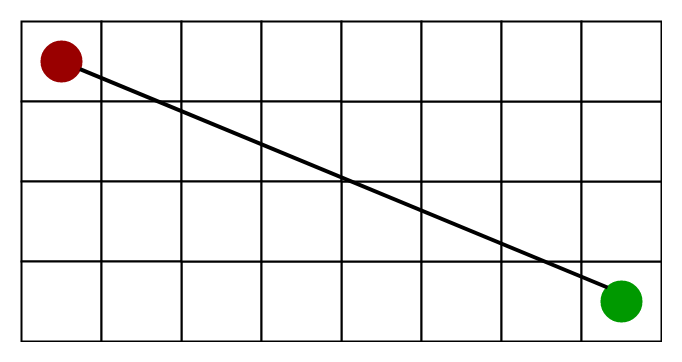
**3.2.3 Euclidean Distance-**

* As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula:

**h** = sqrt ( (current\_cell.x – goal.x)2 + (current\_cell.y – goal.y)2);

* When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



**3.2.4. Relation (Similarity and Differences) with other algorithms:**  
Dijkstra is a special case of A\* Search Algorithm, where h = 0 for all nodes.

## **Conclusion**

Google Maps' approach to shortest path calculations on a global scale leverages a mix of advanced algorithms, real-time data integration, precompution and caching. By breaking down the problem into manageable parts and using sophisticated techniques like hierarchical graphs, heuristic search, precomputation, and dynamic updates, Google Maps can deliver fast and accurate routing information. This multi-faceted strategy ensures that users receive reliable and efficient navigation, tailored to their preferences and real-time conditions.

## **References**

1. [This Is How Google Maps Exactly Works And The Algorithm Behind It (postoast.com)](https://www.postoast.com/how-google-maps-work-and-its-algorithm/)
2. [How Does Google Map Works? - GeeksforGeeks](https://www.geeksforgeeks.org/how-does-google-map-works/)
3. [A\* Search Algorithm - GeeksforGeeks](https://www.geeksforgeeks.org/a-search-algorithm/)
4. [Contraction Hierarchies Guide (jlazarsfeld.github.io)](https://jlazarsfeld.github.io/ch.150.project/)
5. [Euclidean distance - Wikipedia](https://en.wikipedia.org/wiki/Euclidean_distance)