UNIVERSITY OF SCIENCE,

HO CHI MINH

GOOGLE MAP'S ALGORITHMS FOR SEARCHING IN VAST GRAPH

TECHNICAL REPORT
W01

submitted for the Solo Project in Semester 3 – First Year

IT DEPARTMENT

Computer Science

by

Phan Tuan Kiet

Full name: Phan Tuan Kiet

ID: 23125062

Class: 23APCS2

Tasks achieved: 01/01

Lecturer:
Professor Dinh Ba Tien (PhD)
Teaching Assistants:
Mr. Ho Tuan Thanh (MSc)
Mr. Nguyen Le Hoang Dung (MSc)

2024

Contents

I) Graph Partitioning

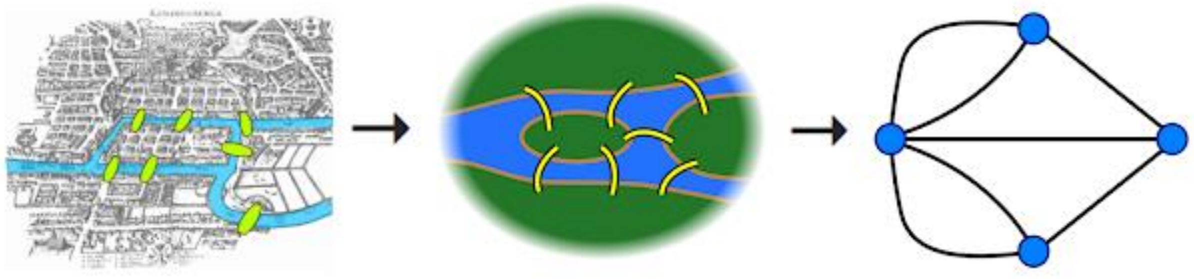
- Dijkstra's algorithm is frequently employed to determine routes in graphs, yet its computational demands can escalate rapidly, exceeding practical limits for anything larger than a small town.

- Therefore, by 'partitioning' a road network, the efficiency of such algorithms can be significantly enhanced. This technique reduces the portion of the graph that needs to be searched, thus speeding up computation.

- Google Maps contains large amounts of data, nodes, and edges. Using graph partitioning is a way to reduce time complexity and enhance efficiency.

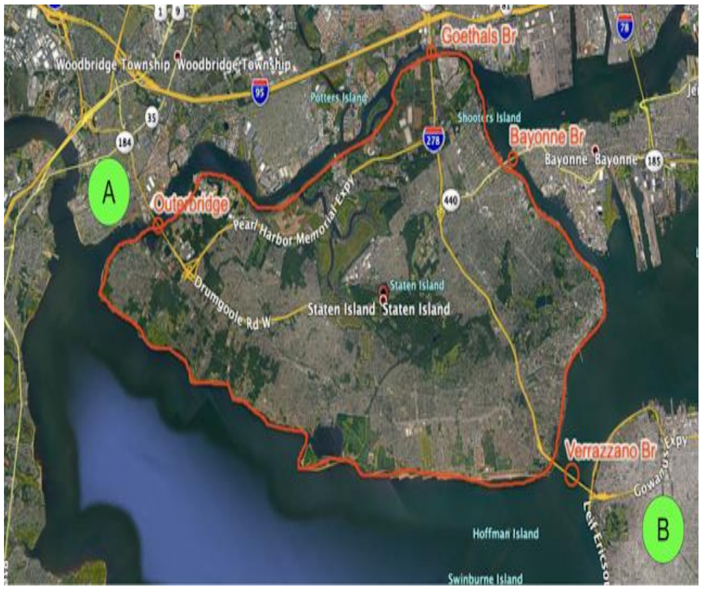
1) Graph – Model Road Networks

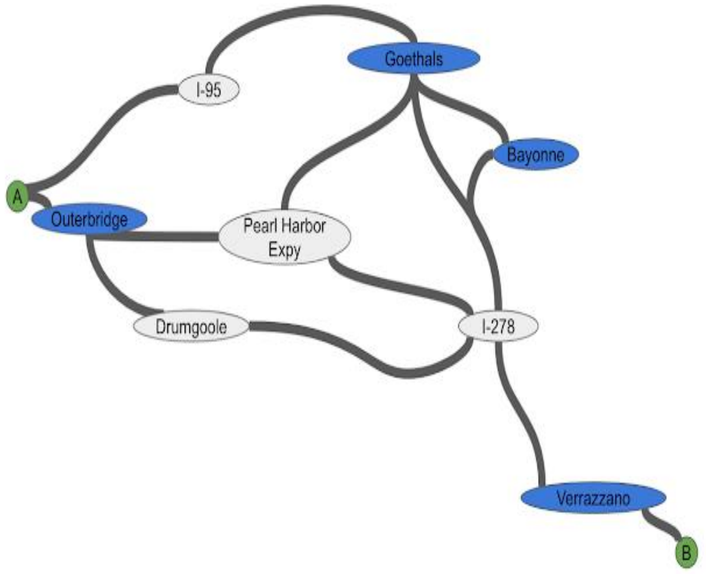
- Road networks can be effectively represented as graphs, where intersections are depicted as nodes and roads as edges.



- As we considered in the last project, let's look at the most well-known solution for finding the fastest route: the Dijkstra algorithm, which works in a best-first search manner. The Dijkstra algorithm performs an exhaustive search starting from the source until it finds the destination. Because of this, as the distance between the source and the destination increases, the computation can become significantly slower.

- The key aspect of graph partitioning is to identify regions that have more internal connections and fewer connections to the outside.





- Every fastest route problem can usually be divided into three smaller chunks: to the entrance, to the exit, and then to the destination. Hence, the problems now become finding the fastest route in each of these chunks, which are called beacons.

- Note that beacons are only useful as long as there are not too many of them. The fewer beacons there are, the fewer shortcuts need to be added, the smaller the search space, and the faster the computation. Therefore, a good partitioning should have relatively fewer beacons for the number of components (i.e., a particular area of a road network).

2) Google Map's algorithm

- There are numerous partitioning schemes, such as METIS (for general networks) and PUNCH. Google Maps' solution is based on the inertial-flow algorithm.

2.1) Problem Statement

- Given a graph G = (V,E), a Graph Partition divides the vertex set V into disjoint subsets V0, V1, ..., Vk-1. These subsets, or regions, (Vi, Ei) are roughly equal in size, and the edges between different regions Vi and Vj (denoted as E(Vi, Vj)) are minimized. The main challenge is balancing the cut size and maintaining equal partition sizes. Variants include balanced k partitioning, ensuring partitions meet an imbalance parameter, or a relaxed variant with a region size constraint. This partition can be achieved by recursively cutting G into two pieces until the subgraphs are sufficiently small, with various objective functions guiding the bisections.

Definition 1 (Cuts and Balanced Cuts). Given a graph G = (V, E), a cut is a partition of V into two disjoint subsets V0, V1. A b-balanced cut (for any $0 < b \leq 1/2$) is a cut such that $|Vi| \geq \lceil b \cdot |V| \rceil$ for both $i \in \{0, 1\}$.

- At each level of the recursion, the objective is to find a b-balanced cut (for some b, say b = 1/4) that minimizes the number of cut edges, i.e., min |E(V0, V1)|, where E(V0, V1) := {(u, v) ∈ E : u ∈ V0, v ∈ V1}.

2.2) Inertial Flow

- The main method of choosing b-balanced cuts in road networks is actually effiicent heuristic. Assump that the undirected graph G = (V, E) is connected, as typically partitioning algorithms are applied to each connected component independently. The method applies two standard primitives: sorting and maximum flow.

1) Pick a line l in R^2 and orthogonally project V onto l (more precisely, for each vertex v, project its point in the embedding f(v) onto l).

2) Sort V by order of appearance on l (ties broken arbitrarily but consistently).

3) Let the first [b.|V|] vertices (in projection order) be the source s, and let the last [b.|V|] vertices be the sink t.

4) Compute a maximum flow between source s and sink t.

5) Return a corresponding minimum st cut.

Key properties

- By choice of s and t, all minimum st cuts are b-balanced.

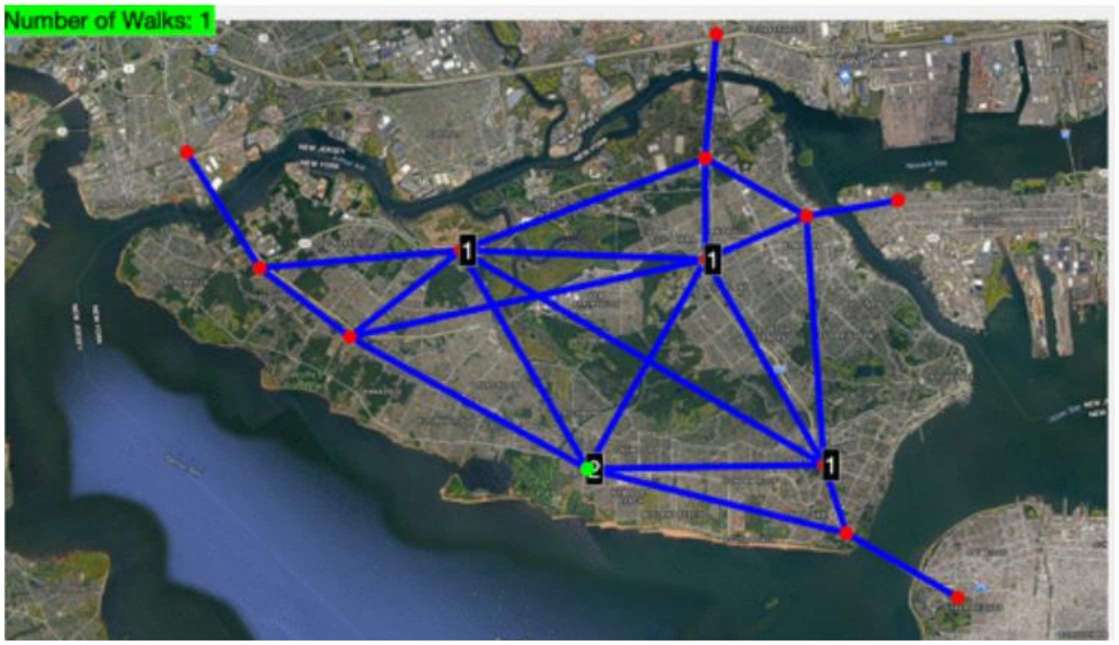
- The running time is bounded by the time required to sort V plus the time required to compute one maximum flow in G. Computing the entire separator tree (recursive bisection) requires time proportional to sort plus $\log_{1/(1-b)} |V|$.

- A basic implementation using standard libraries is straightforward.

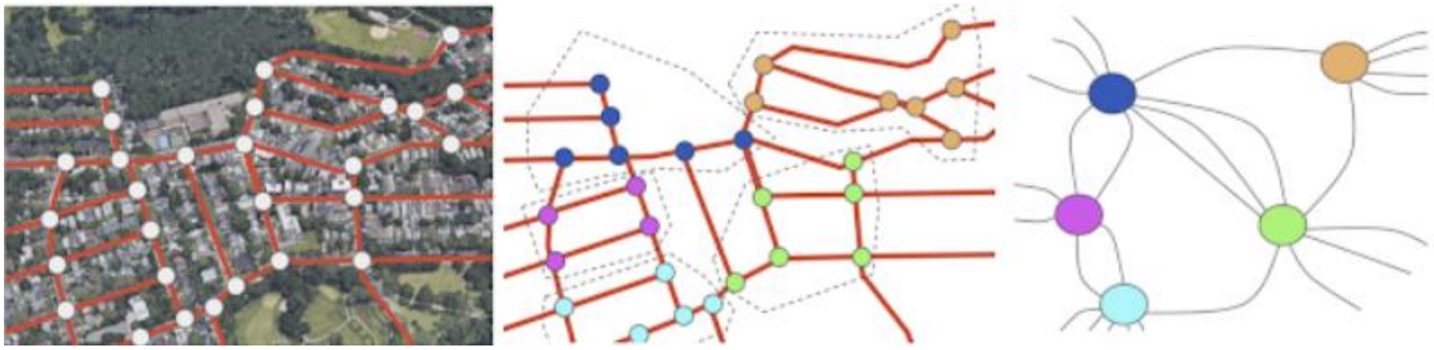
- The quality of the cut depends on the line $l \in R^2$ chosen in the first step of the algorithm.

3) Balanced Partitioning and Random Walk

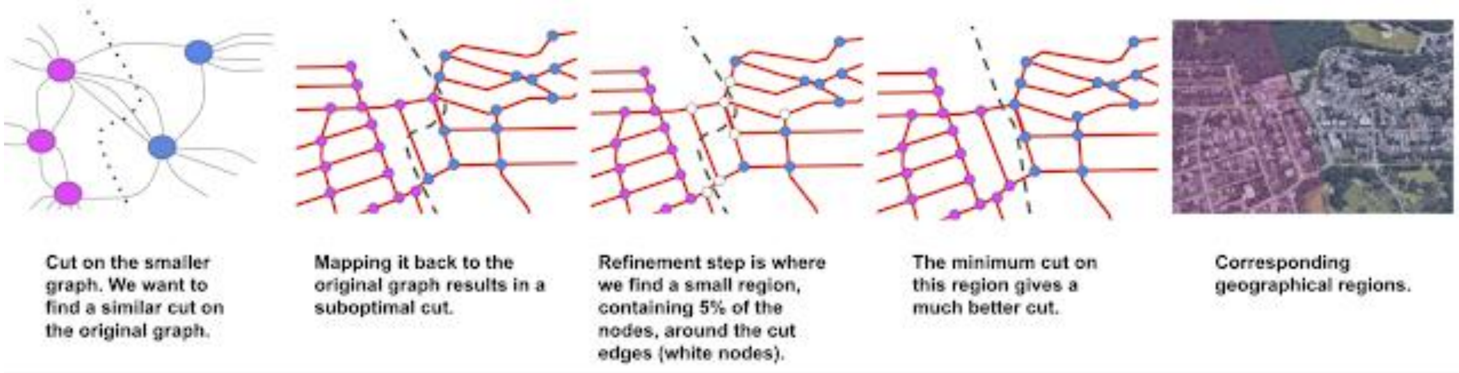
- To partition a road network graph into balanced components, let's begin by aggregating closely connected nodes to streamline subsequent partitioning. Random walks are then employed, leveraging their tendency to remain within well-connected areas while having limited connections to the outside.



- After identifying these tightly connected components, the algorithm condenses each group into a single new node.



- Having found a cut on the small graph, the algorithm performs a refinement step to project the cut back to the original graph of the road network.


Cut on the smaller graph. We want to find a similar cut on the original graph.
Mapping it back to the original graph results in a suboptimal cut.
Refinement step is where we find a small region, containing 5% of the nodes, around the cut edges (white nodes).
The minimum cut on this region gives a much better cut.
Corresponding geographical regions.

→ Break down a large scale graph problem into smaller subproblems to be solved independently and in parallel.

II) Multi-Level Dijkstra and A* Search

1) Multi-Level Dijkstra

- Multi-Level Dijkstra is a hierarchical approach to Dijkstra's algorithm, where the graph is processed at multiple levels of detail or abstraction. The key idea is to reduce the computational complexity of finding shortest paths by working with simplified versions of the graph first and then refining the path on more detailed levels as necessary. The search graph is the objectvie of this MLD algorithm, further enhancing MLD graphs and optimization will be described below.

1.1) Enhancing Multi-Level Graphs.

- A multi-level graph M extends a weighted digraph G = (V,E) through multiple levels of edges, depending on a sequence of vertex subsets. For a pair of vertices s,t ∈ V, a subgraph Mst of M, called search graph, with the same s-t distance as in G can be determined efficiently. As the search graph is substantially smaller than G, it allows for answering the given query much faster.

HIGH-PERFORMANCE MULTI-LEVEL ROUTING

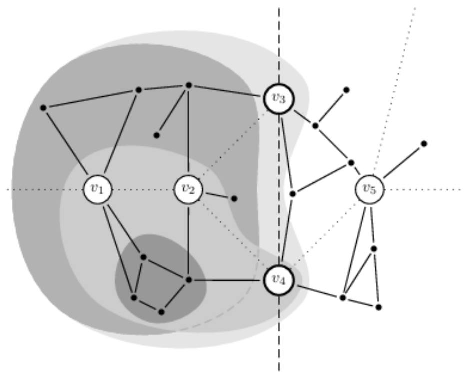


FIGURE 1. Hierarchy due to graph decomposition: components (darker shades) with belonging wrapped components (lighter shades) at levels 1 (smaller components) and 2 (larger components).

D. DELLING, M. HOLZER, K. MÜLLER, F. SCHULZ, AND D. WAGNER

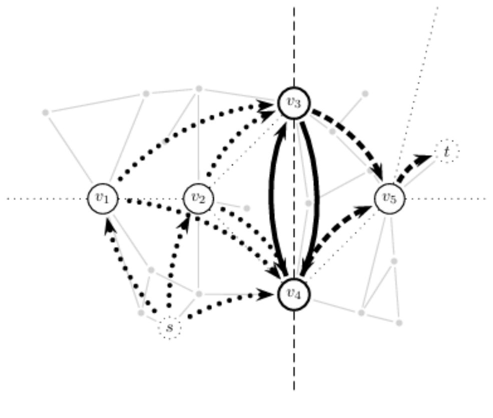


FIGURE 2. The search graph $\mathcal{M}_{st}$: edges from $\mathscr{L}$, $\mathscr{U}_i$, and $\mathscr{D}_i$ are shown as thick lines with solid, dotted, and dashed styles, respectively.

<u>1.2) Optimizing Partial Graphs.</u>

- In contrast to the classic variant, where a multi-level graph is stored as a whole, let's spread it over a large number of partial graphs (as seen before, any search graph can be constructed through the union of a number of appropriate partial graphs). The foremost advantage is that each of them can be optimized individually using two different techniques: pruning of edges that cannot contribute to a shortest path and conversion of a partial graph into an equivalent one with more vertices but fewer edges.

HIGH-PERFORMANCE MULTI-LEVEL ROUTING

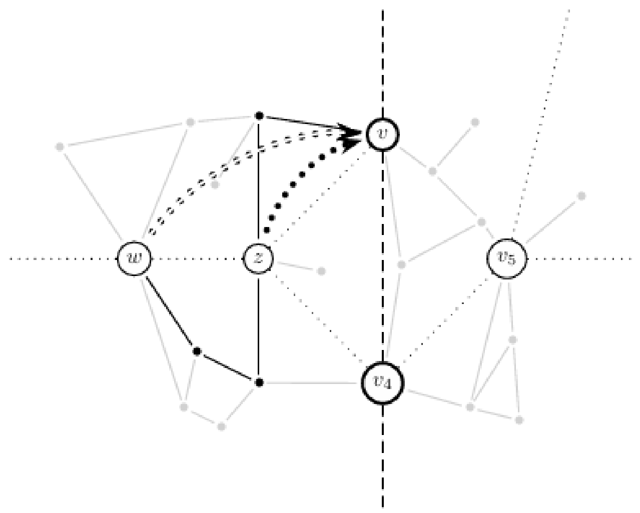


FIGURE 4. Superseded edges. Edge (w, v) (double-dotted line) is superseded by (z, v) (dotted line), because there is a shortest w-v path via z in the input graph (highlighted thin lines).

- A further optimization technique is based on the following idea. The number of edges of a partial graph can be reduced by introducing auxiliary vertices and replacing many original edges with few edges through the new vertices such that distances are preserved.

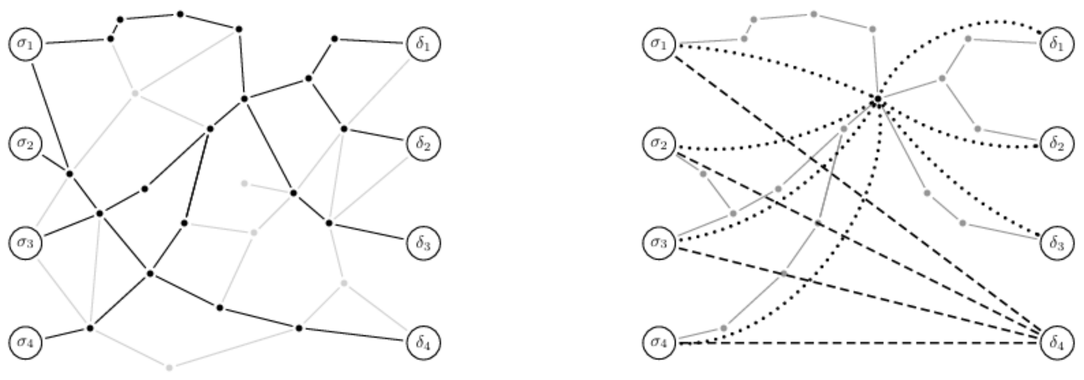


FIGURE 5. Constructing equivalent graphs. Left: sample graph; highlighted edges are contained in a shortest σ-δ path. Right: belonging search graph with edge compression applied; dotted and dashed edges are contained in the graph.

2) A* algorithm

- A* algorithm is a flexible, more efficient, and cutting-edge algorithm currently utilized by Google Maps to find the shortest path between a given source and destination. It can handle larger graphs, making it preferable over Dijkstra's algorithm in a wider range of contexts. A* operates similarly to greedy Best-First-Search. The algorithm is widely used because it combines the aspects of Dijkstra's algorithm (favoring vertices close to the source) and Greedy Best-First-Search (choosing nodes close to the goal).

- One of the difference between A* and Dijkstra's algorithm is that A* algorithm, being a heuristic algorithm, has two cost functions :

+ g(x): The actual cost to reach a vertex (same applies to Dijkstra).

+ h(x): Approximate cost to reach goal node from node n. (Heuristic part of the cost function).

+ f(x): Total estimated cost of each node.

- The cost calculated by A* algorithm shouldn't be overestimated, since it is a heuristic algorithm. Hence, the actual cost to reach the destination node, from any given node n should be either greater than or equal to h(x). This is called Admissible Heuristic.

→ $f(x) = h(x) + g(x)$.

- A node is expanded by A* search, only if it is considered to be promising. This algorithm focuses solely on reaching the goal node from the node which is currently under consideration. It does not try to reach other nodes

→ Single shortest path problem, not all shortest paths like the previous project we dived into.

- Thus, if the heuristic function taken into consideration approximates the future cost accurately, then one needs to explore considerably less nodes as compared to Dijkstra's algorithm.

- A* algorithm terminates when the destination node has been reached ensuring that the total cost of reaching the destination node is the minimum compared to all the other paths. It can also terminate if the destination node has not been reached despite all the nodes being explored. Unlike, Dijkstra's algorithm, it does not explore all nodes possible, thus A* proves to be more proficient.

- Owing to the high accuracy, flexibilty, ability to deal with mammoth graphs and its stark advantages over Dijkstra's algorithm, Google Maps has shifted to deploying A* algorithm for obtaining the shortest path between the source and destination.

2.1) Algorithm

1) Initialise two lists:

    a) Open_list

    b) Closed_list

2) Insert the source node into the open_list

3) Keep the total cost of the source node as zero

4) While open_list is not NULL:

    a) Check the node with minimum total cost (least f(x)) present in the open_list

    b) Rename the above node as X

    c) Pop X out of the open_list

    d) Generate the successors to X

```
        e) Set X as the parent node of all these successors
5)      for each successor:
            a) if successor is same as the destination:stop
            b) else:
                successor.g(x) = X.g(x) + distance between successor and X
                successor.h(x) = distance from destination [goal] to successor (using heuristics)
                successor.f(X) = successor.g(x) + suucessor.h(x)
            c) if a node with same position as successor is present in open_list, but having a lower cost [f(x)]
            than successor, then skip the successor
            d) if a node with the same position as the successor is present in the closed_list , but having a lower
            cost [f(x)], then skip this successor, else insert the node into the open_list
        end for loop
6) Add X into the closed list
7) End the while loop
8) The route is obtained by the closed_list
9) Stop.
```

2.2) PSEUDO CODE

```
function A_star_algorithm (start, goal)
    open_list = Initialised set contains start //Initialisation of the open list closed_list = NULL set //Initialisation of the
    closed list
    start.g = 0
    start.f = start.g + h(start, goal) // f(x)= g(x) + h(x) where h is the heuristic function
    while open_list is not empty:
        node = element in open list with lowest cost //f(x) should be minimum
        if node = goal// when destination is reached, current node = goal return route(goal) // route found
        remove node from open_list
        insert node into closed_list
        for neighbour in neighbours(node):
            if neighbour not in closed_list:
                neighbour.f = neighbour.g + h(neighbour, goal) //f(x)= g(x) + h(x) where h is the heuristic
        function
            if neighbour is not in open_list:
                insert neighbour into open_list
            else:
                x = neighbour node already present in open_list
        if neighbour.g < x.g:
            x.g = neighbour.g
            x.parent = neighbour.parent
    return False // no path exists
function neighbours(node)
    neighbours = set of valid neighbours to node // check for obstacles here
    for neighbour in neighbours:
        if neighbour is diagonal:
            neighbour.g = node.g + diagonal_cost // Pythagoras Theorem eg. 5^2 = 3^2 + 4^2 → here 5 is the
    diagonal
        else:
            neighbour.g = node.g + normal_cost // eg. 3
            neighbour.parent = node
    return neighbours
function route(node) / Displays the route
    path = the set which contains the current node
    while node.parent exists:
        node = node.parent
        add node to path
    return path
```

III) Real-Time Updates Algorithm

- With GPS accuracy improving to within a few meters, GPS navigation systems have become widely popular, replacing paper maps for travel directions. These systems allow real-time route planning using electronic maps, making travel more efficient. However, outdated maps or real-time traffic incidents can lead to incorrect routing.

- Methods like the Radio Data System-Traffic Message Channel (RDS-TMC) transmit real-time traffic data via FM channels, but this is mostly limited to urban areas and highways. RDS-TMC's low bit rate also restricts the amount of traffic data transmitted.

- Modern GPS navigation apps, such as Garmin StreetPilot Onboard and PaPaGo! Mobile, pre-load offline maps and can access real-time traffic information online to optimize routes. Nevertheless, these apps face similar issues as RDS-TMC, with incomplete real-time data coverage and the need for an active internet connection.

- How does Google Maps resolve this issue?

→ The Google Maps app tracks the location and speed of users who have opted in to share their location data. This data is anonymized and aggregated to provide real-time traffic information.

→ Google also employ sophisticated machine learning algorithms to analyze the data collected from various sources and predict traffic conditions. These algorithms can identify patterns and trends in the data to provide accurate real-time traffic updates.

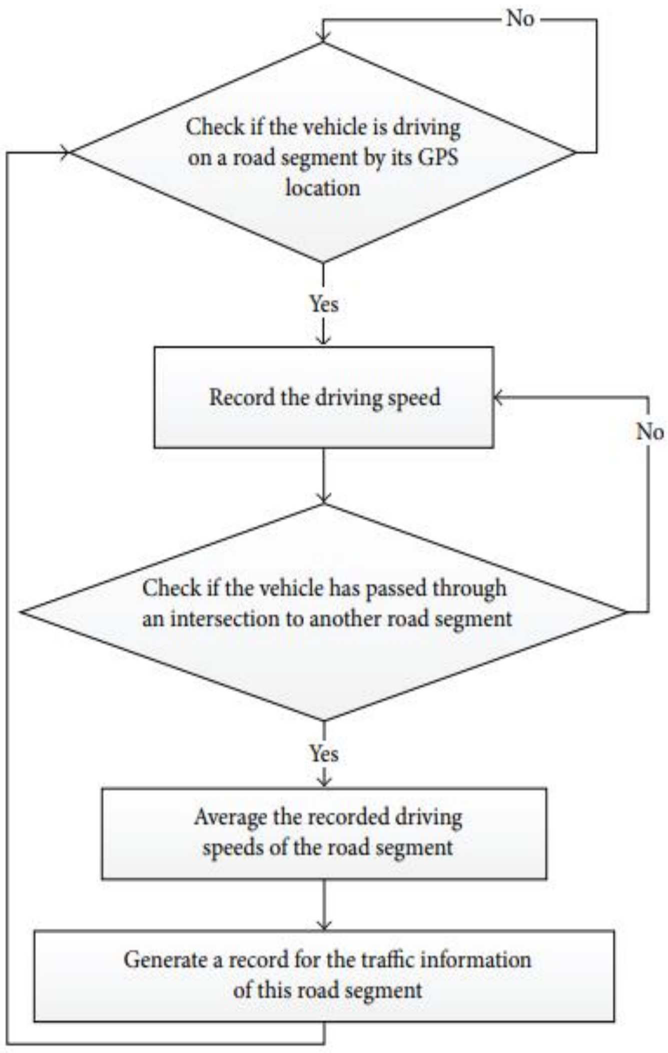

No
Check if the vehicle is driving on a road segment by its GPS location
Yes
Record the driving speed
No
Check if the vehicle has passed through an intersection to another road segment
Yes
Average the recorded driving speeds of the road segment
Generate a record for the traffic information of this road segment

FIGURE 3: The traffic information recording flow of the vehicle.

TABLE 1: The three-field format for traffic information of a road segment.

FIGURE 4: Real-time traffic information on Google Maps.

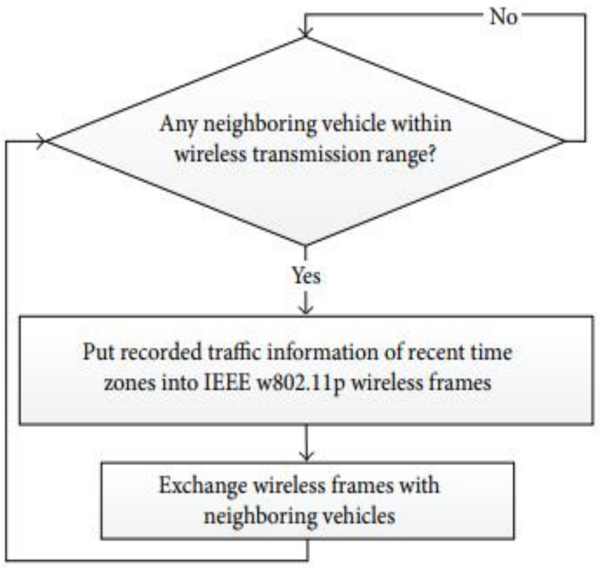

No
Any neighboring vehicle within wireless transmission range?
Yes
Put recorded traffic information of recent time zones into IEEE w802.11p wireless frames
Exchange wireless frames with neighboring vehicles

FIGURE 5: The traffic information distribution flow.

- Why is this important? Because relying solely on algorithms isn't enough! To generate accurate and useful information for applying A* Search or Dijkstra's Algorithm to determine the best route, we need to continuously justify and update the current situation in real time. This involves accounting for every second of changes in traffic conditions, road closures, and other dynamic factors that can affect route planning.

IV) Google Maps' Graph Enhanced Optimization

1) Overview

- In the context of *the Google Maps* app, graphs are used to represent a network of roads with vertices representing important points such as intersections or locations, and sides representing the roads between those points. Each side has a weight that indicates the length or travel time between two related points.

- Google Maps uses a number of data sources to obtain information about traffic and road availability, including user reports, traffic sensors, and historical traffic data. This information is then integrated into the graph representation of the road system, where each road is represented as a node and the connections between the roads as edges. Using algorithms such as A* or Dijkstra, Google Maps can perform calculations to determine the best route based on a combination of physical distance and estimated travel time affected by traffic.

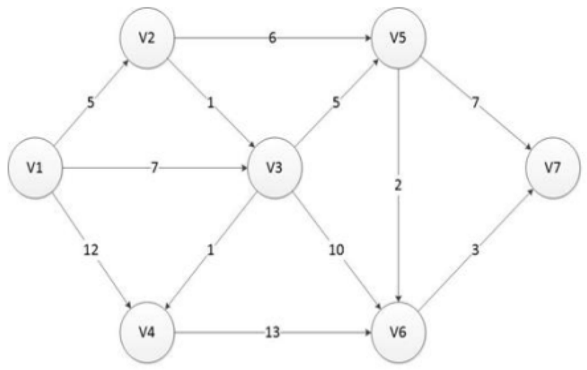


Figure 1. (a)

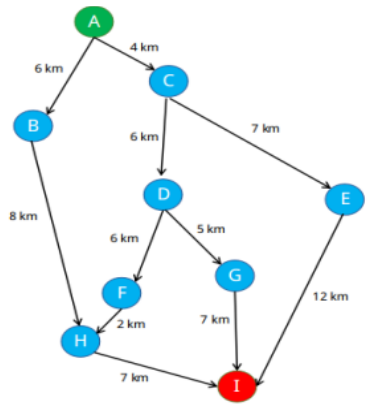


Figure 1. (b)

Figure 1. (a) is an example of the Dijkstra Algorithm while Figure 1. (b) is an example of the A* Algorithm. In the context of *the Google Maps* App, the Dijkstra Algorithm is used to determine the shortest route between two locations desired by the user. The road network graph is represented by vertices representing locations and sides representing roads, while the weights on each side reflect the length of the road or travel time between two locations. By applying the Dijkstra

- Google Maps takes advantage of the advantages of each of these algorithms. The combination of these two algorithms allows Google Maps to provide optimal, responsive, and reliable route planning services. Using the Dijkstra or A* algorithm, Google Maps can perform calculations to determine the best route based on a combination of physical distance and estimated travel time affected by traffic. When a route is selected, traffic and availability data is continuously monitored in real-time. If there is a change in road conditions, such as a traffic jam or road closure, Google Maps will automatically adjust the route to find a faster or more efficient alternative. The integration of traffic and availability data not only helps users get more accurate travel time estimates, but also allows them to avoid congestion and slow routes. This optimizes the user experience and ensures that they can reach their goals quickly and efficiently using the Google Maps app.

2) General Method

2.1) Road Mapping

- Google Maps has a very detailed road mapping, which includes many road points and segments around the world. Each path is represented in the graph as a series of vertices connected by sides.

2.2) Determining the Starting Point and Destination

- The Dijkstra algorithm then explores the nodes that are neighboring the initial node and updates the shortest distance value from the initial node to those nodes, after which it selects the node that has the shortest distance from the initial node, then marks it as a fixed node and updates the distance value to neighboring nodes.

2.3) Route Determination

- After the route search algorithm has been executed, *Google Maps* determines the optimal route from the starting point to the destination point based on the results of the algorithm. This route is then displayed to the user in the form of complete directions.

- Geographic data includes the specific location of the road, usually in the form of GPS coordinates. Different types of roads are also noted, such as highways, city roads, or country roads, each with different characteristics. In addition, information on road direction (one-way or two-way), number of lanes, and speed limits in each road segment is collected. This information is important to ensure accuracy in route creation and travel time estimates. *Real-time* traffic data is also crucial, covering current traffic conditions such as congestion, accidents, or construction work that can affect travel. This data is frequently updated through traffic sensors, reports from users, and analysis of travel patterns that are collected continuously. This one I have mentioned above.

→ Once all the data has been collected, the next step is to organize it into the appropriate data structure for graph representation.

<u>3) Advanced Improvements (Based on real research)</u>

- *Hierarchical graphs*, which group nodes and sides into hierarchies based on the importance of roads, have also been implemented to reduce computational complexity.

- Another important optimizations in path finding is the use of *preprocessing*, where the path network is pre-processed to speed up path finding during actual execution. *Preprocessing* involves the initial analysis and simplification of road network data, so that when the path finding algorithm is applied, the process becomes much faster and more efficient. This technique is particularly beneficial in handling large and complex road graphs, such as those used in modern navigation applications.

- The *contraction hierarchies* technique is one of the effective *preprocessing* methods . This technique allows for the simplification of the graph by contracting vertices and sides that are considered less important, thereby reducing the size of the graph actually used in the path finding. This contraction process maintains critical paths and eliminates *insignificant nodes* and *edges* without changing the final result of the path look. With a smaller graph size, the computational time to find the shortest path can be significantly reduced.

- Another approach that improves the efficiency of pathfinding is *dynamic adjustment* based on *real-time* traffic conditions. This algorithm allows for the adjustment of the side weights in the graph according to current traffic conditions, such as congestion or road closures. *Real-time* traffic information is obtained from a variety of sources, including road sensors, user reports, and historical data.

→ By dynamically updating the weights, the algorithm can provide more accurate travel time predictions and more efficient routes. By combining various optimizations and heuristics, the Dijkstra algorithm can be adapted and optimized for use in largescale routing systems. For example, the use of *contraction hierarchies* can be combined with dynamic adjustments to maximize efficiency. Additionally, heuristics such as those used in the A* algorithm can help guide pathfinding more efficiently, reducing the number of *nodes* that need to be explored.

- *Dynamic customization* is another important element that allows *Google Maps* to quickly adjust routes based on changes in traffic conditions in *real-time*. The algorithm used can update the recommended route in the event of a significant change in traffic conditions, such as congestion or accidents. This ensures that users always get the most efficient route at that time. The use of real-time traffic data is the key to *Google Maps* ' success in providing accurate travel time estimates. This data is obtained from a variety of sources, including road sensors, GPS data from users, and traffic reports from transportation authorities. Real-time data integration allows applications to quickly update traffic conditions and adjust recommended routes.

- One of other important techniques in route determination optimization is *Contraction Hierarchies (CH).* This method was introduced to speed up routing by reducing the size of the graph through the contraction of less important nodes. This technique has shown significant efficiency in navigation and transportation applications. The basic principle of *Contraction Hierarchies* is to remove or merge insignificant nodes in the path graph, while retaining important path information. By reducing the number of *nodes* that must be processed, the algorithm can find the shortest path faster. This contraction process involves selecting the least important nodes and replacing them with shortcuts that connect neighboring *nodes* directly.

+ *Insignificant nodes* are *those* whose contribution to the entire path is considered minimal. The removal or contraction of these *nodes* is carried out based on the topology analysis of the road graph. *Nodes* that have few connections or that are not widely used in the optimal route are often prime candidates for contraction. Even if *the nodes* are deleted, the path information is still preserved through *shortcuts*. This *shortcut* is a shortcut that connects neighboring *nodes* from deleted nodes, ensuring that the shortest route is not interrupted. In this way, important information regarding the path remains even though some nodes have contracted.

+ One of the main advantages of CH is the significant reduction in compute time. By reducing the number of *nodes* that must be processed, CH reduces the computational load and enables faster route finding. This is especially important in real-time applications such as navigation. CH also improves the scalability of routing algorithms. With the ability to handle very large road graphs, CH enables navigation applications to operate efficiently in large cities with complex road networks. This scalability is critical for applications used by millions of users around the world.

→ Overall, *Contraction Hierarchies* are a very effective technique in routing optimization. By contracting less important nodes and retaining path information through shortcuts, CH reduces computation time and improves algorithm efficiency. The implementation of CH in applications such as *Google Maps* shows how this technique can be used to handle very large and complex road graphs quickly and accurately.

4) Shortest path Integration with Real-time traffic data

- The integration of Dijkstra's algorithm with real-time traffic data is an important step to improve accuracy and efficiency in routing. The Dijkstra algorithm has traditionally been used to find the shortest path in a weighted graph, but in the context of real traffic, road conditions can change rapidly. Therefore, the use of real-time traffic data allows this algorithm to update the side weights in the graph according to current traffic conditions, such as congestion, accidents, or road construction. This integration ensures that the recommended route is always optimal based on the latest information. Research on dynamic shortest lane algorithms, such as *Dynamic* A* or (D*), shows great potential in efficiently adjusting routes based on changing traffic conditions. D* is an extension of the A* algorithm that allows dynamic adjustment to changes in the graph during the execution of a path search. This is especially relevant for real-time navigation applications, where road conditions can change unexpectedly and quickly. The D* algorithm leverages the most recent information to update the path and side weights in the graph, allowing the system to adjust the route without the need to recalculate the entire path from scratch.

→ The ability to dynamically adjust routes is essential to ensure that users get the fastest and most efficient route based on the current traffic conditions. The main advantage of the dynamic shortest lane algorithm is its ability to respond to changes in traffic conditions in real-time. The implementation of this algorithm in navigation applications, such as *Google Maps,* allows users to get an optimal route at all times. Thus, users can save travel time and avoid stress due to unexpected traffic jams. The algorithm can also reduce the traffic load on major roads by distributing vehicles to alternative routes that are less congested.

5) Bidirectional search optimization

- Bidirectional search is an effective method to speed up the process of finding a route by starting the search from the origin and destination points at the same time. This method aims to reduce the number of nodes that need to be explored during the pathfinding process. Two-way search works by initiating two separate searches: one from the origin point and one from the destination. These two quests move towards each other and meet in the middle. In this way, the total number of nodes that need to be explored can be significantly reduced, since each quest only needs to cover half of the total distance between origin and destination → Improve efficiency.

- One of the main advantages of two-way search is the reduction in the total number of *nodes* that need to be explored. In a traditional one-way search, the entire path from the origin point to the destination is explored, which can be very timeconsuming and resource-consuming. With bidirectional search, *node exploration* can be split between two searches, reducing the computational load. Two-way search has been shown to speed up computing time by up to half compared to oneway search. The reduction in the number of *nodes* explored means that the algorithm can find the fastest route faster, which is especially important in applications that require quick responses, such as road navigation.

- By combining two-way search and *Contraction Hierarchies,* the algorithm becomes more efficient at handling large road graphs. CH reduces the size of the graph by contracting less important nodes, while bidirectional search reduces the number of *nodes* that need to be explored. This combination provides a significant performance improvement in routing. As an example of a real application, *Google Maps* uses a combination of these algorithms to calculate the fastest route for its millions of users every day. By using CH and two-way search, *Google Maps* can provide accurate and efficient routes in a very short time, even if it has to deal with a very large and complex road network.

- Two-way search adds efficiency by starting a path search from two points: the starting point and the destination point. These two quests meet in the middle, reducing the total number of *nodes* that need to be explored. This technique not only reduces computing time but also improves the accuracy of search results. *Google Maps* uses this method to ensure that users get the fastest route with minimal computing time. Efficient implementation of algorithms is essential for navigation applications used by millions of users every day. Efficiency in computing allows *Google Maps* to provide fast and accurate route search results, ensuring that users do not have to wait long to get route information. This efficiency also allows the system to handle a large number of requests without experiencing performance degradation.

V) Conclusion

→ Maps and navigation apps like Google Maps have become essential tools in daily life, helping users find the fastest or shortest route between locations. In these apps, roads are represented as nodes, and the distances between them are weights on the edges connecting the nodes. Route search algorithms, such as Dijkstra's and A*, are crucial in Google Maps. These algorithms analyze road graphs, considering the distances and travel costs on each edge to determine the optimal route from the starting point to the destination. By leveraging graph technology and route search algorithms, Google Maps delivers an efficient navigation service.

References

[Inertial Flow] Aaron Schild and Christian Sommer. SEA 2015 – 14th International Symposium on Experimental Algorithms (pp. 286-297).

[Multi-Level Dijkstra] Daniel Delling, Martin Holzer, Kirill M¨ uller, Frank Schulz, and Dorothea Wagner.

[A Star Algorithm] Heeket Mehta - Dept. of Computer Engineering Dwarkadas J Sanghvi college of Engineering, University of Mumbai, India.
Pratik Kanani - Dept. of Computer Engineering Dwarkadas J Sanghvi college of Engineering, University of Mumbai, India.
Priya Lande - Dept. of Computer Engineering Dwarkadas J Sanghvi college of Engineering, University of Mumbai, India.

[Real Time Updates] Ing-Chau Chang, Hung-Ta Tai, Feng-Han Yeh, Dung-Lin Hsieh, and Siao-Hui Chang Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua 500, Taiwan.

[Google Maps Graphs] Della Putri Ristanti[1], Rafiantika Megahnia Prihandini[2], Dinda Alvina Roikhatul Jannah[3] Department of Mathematics Education, University of Jember, Jember, Indonesia