## A blue and white cover with a map Description automatically generatedCover Page

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**

Shortest path problem is always a concerning problem for us to deal with.

In the previous task, we have gone through the algorithms that speed up the process of searching the shortest path which can be used in Google Maps.

Those algorithms are:

- A\* Search algorithm

- Path Caching

- Bidirectional Dijkstra

- Contraction Hierarchy

In this task, I am trying to apply those algorithms to the graph presented in course CS162 to solve the shortest path problem in a fast, convenient way compared to Dijkstra’s algorithm.

\* Recall about the graph:

This graph represents a bus map in Ho Chi Minh City with 4347 stops along with approximately 10000 edges and a plenty of route and routeVar. Each route has its own path stored in “path.json”, and the list of the stops it go through in “stops.json”. Moreover, all the general information of all the route are also stored in “vars.json”.

Through various efforts and test cases, those advanced shortest path algorithms have shown significant improvement and applications in querying a shortest path between two stops.

## **A\* Search Alogorithm**

Developed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael, the [A\* algorithm](https://www.geeksforgeeks.org/a-is-admissible/)was designed as an extension and improvement of Dijkstra’s algorithm, which is also known for finding the shortest path between nodes in a graph. Unlike Dijkstra’s algorithm, which uniformly explores all directions around the starting node, A\* uses heuristics to estimate the cost from a node to the goal, thereby optimizing the search process and reducing the computational load.

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 below.

### **2.1. 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, including 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  
 else add the node to the open list  
 end (for loop)  
   
 e) push q on the closed list  
 end (while loop)

*(source: in [1])*

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.



### **2.2. Calculating heuristic values**

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.

*\*\* 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)

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

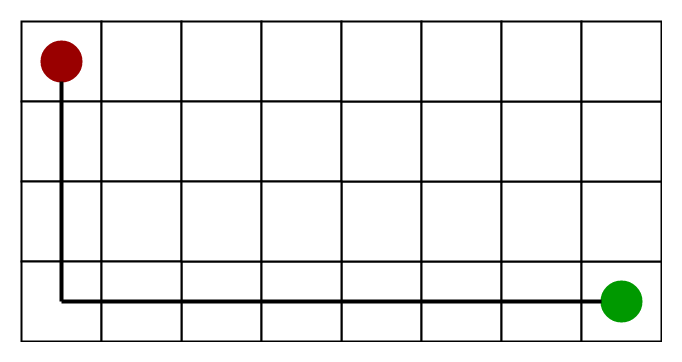
**2.2.1. Manhattan Distance**

* 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).



**2.2.2 Diagonal Distance**

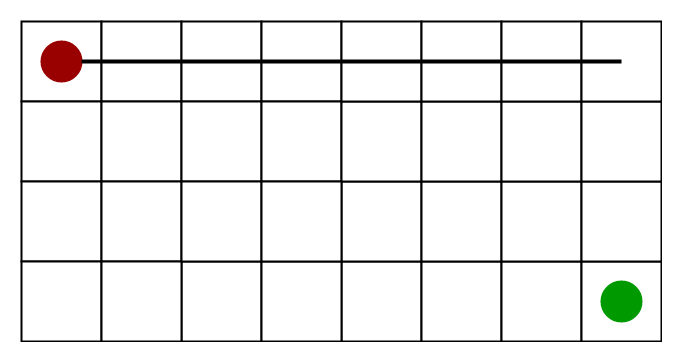
* It is nothing but the maximum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively, i.e.,

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).



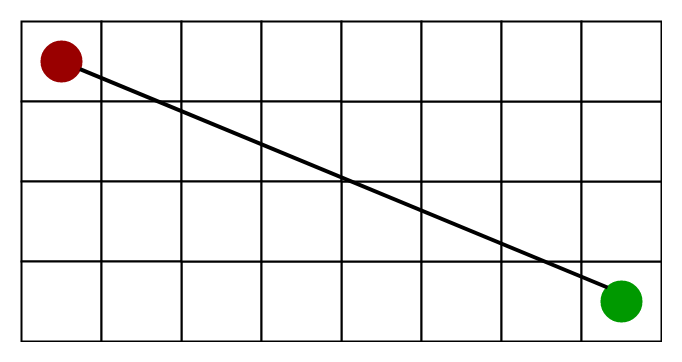
**2.2.3 Euclidean Distance-**

* 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).



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

### **2.3. Implementation**

In this task, I use heuristic function as time prediction from the current stop to the end stop.

# function to calculate h (time estimate)

h = lambda stopId1, stopId2, route, routeVar: sqrt((stopPos[stopId1][0] - stopPos[stopId2][0])\*\*2 + (stopPos[stopId1][1] - stopPos[stopId2][1])\*\*2)/vel[(route, routeVar)]

(h = distance from that stop to the end stop / velocity of that route)

And this is the main algorithm: (pq is the open list)

A computer screen with many white and green text

Description automatically generated

In short, it is similar to Dijkstra’s algorithm, but what we have to push in the queue here is the value of f. Moreover, when we meet the end stop in the queue, we end the search.

Output with 7269 is the start stop and 695 is the end stop:

Precompute (with ignorance to the StopQuery and RouteVarQuery): 0.0000012341

A\* search time: 0.07787561416625976562

[7269, 7273, 7274, 7275, 35, 89, 90, 1409, 1413, 1416, 1891, 388, 390, 569, 573, 433, 434, 728, 115, 117, 116, 725, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 3422, 3430, 641, 643, 642, 644, 645, 646, 647, 648, 649, 650, 651, 3433, 7220, 3434, 695]

Time from 7269 to 695: 116.50851797806007

Traceback time: 0.000001245

Total time: 0.07886672019958496094

----------------------------

2.4. Benefits and drawbacks

The A\* algorithm offers several advantages. Firstly, it guarantees finding the optimal path when used with appropriate heuristics. Secondly, it is efficient and can handle large search spaces by effectively pruning unpromising paths. Thirdly, it can be easily tailored to accommodate different problem domains and heuristics. Fourthly, A\* is flexible and adaptable to varying terrain costs or constraints. Additionally, it is widely implemented and has a vast amount of resources and support available. Therefore, all of them make it a popular choice for solving pathfinding and optimization problems.

While the A\* algorithm has numerous advantages, it also has some limitations. One disadvantage is that A\* can be computationally expensive in certain scenarios, especially when the search space is extensive and the number of possible paths is large. The algorithm may consume significant memory and processing resources. Another limitation is that A\* heavily relies on the quality of the heuristic function. If the heuristic is poorly designed or does not accurately estimate the distance to the goal, the algorithm's performance and optimality may be compromised. Additionally, A\* may struggle with certain types of graphs or search spaces that exhibit irregular or unpredictable structures.

It is important to mention that this algorithm is no more than a “smart guess”.

## **Path Caching**

## **Bidirectional Dijkstra Algorithm**

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.

## **Contraction Hierarchy**

## **Conclusion**

## **Reference**

1. [A\* Search Algorithm - GeeksforGeeks](https://www.geeksforgeeks.org/a-search-algorithm/)
2. [Contraction Hierarchies Guide (jlazarsfeld.github.io)](https://jlazarsfeld.github.io/ch.150.project/)
3. [Euclidean distance - Wikipedia](https://en.wikipedia.org/wiki/Euclidean_distance)
4. [A\* Algorithm in Artificial Intelligence You Must Know in 2024 | Simplilearn](https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm)