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

Contents

[Cover Page 1](#_Toc172163180)

[**1.** **Introduction** 4](#_Toc172163181)

[**2.** **A\* Search Alogorithm** 5](#_Toc172163182)

[**2.1. Algorithms** 6](#_Toc172163183)

[**2.2. Calculating heuristic values** 7](#_Toc172163184)

[*\*\* Exact Heuristics* 7](#_Toc172163185)

[*\*\* Approximation Heuristics* 7](#_Toc172163186)

[**2.3. Implementation** 9](#_Toc172163187)

[**2.4. Benefits and drawbacks** 11](#_Toc172163188)

[**3.** **Path Caching** 11](#_Toc172163189)

[**4.** **Bidirectional Dijkstra Algorithm** 12](#_Toc172163190)

[**5.** **Contraction Hierarchy** 12](#_Toc172163191)

[**6.** **Conclusion** 12](#_Toc172163192)

[**7.** **Reference** 12](#_Toc172163193)

## **Introduction**

The shortest path problem is a persistent challenge in the realm of computer science and practical applications such as navigation and network routing. In previous discussions, we have explored several advanced algorithms designed to improve the process of finding the shortest path, which are particularly relevant for applications like Google Maps. These algorithms include A\* Search, Path Caching, and Contraction Hierarchy.

In this task, I aim to apply these sophisticated algorithms to a specific graph presented in the CS162 course, aiming to solve the shortest path problem more efficiently and conveniently compared to the traditional Dijkstra’s algorithm.

### 1.1. Recall the Graph

The graph in CS162 course represents a comprehensive bus map of Ho Chi Minh City, featuring 4,347 bus stops and approximately 10,000 edges connecting these stops. This intricate network is characterized by a multitude of routes and route variations (routeVar), each with its own path. The paths for each route are stored in the "path.json" file, while the stops included in these routes are listed in the "stops.json" file. Additionally, general information about all the routes is available in the "vars.json" file.

### 1.2. Algorithms for Speed and Efficiency

- **A\* Search Algorithm**: This algorithm enhances the classic Dijkstra's algorithm by incorporating heuristics to guide the search process, significantly reducing the number of explored nodes and thus speeding up the search. By using an admissible heuristic, A\* ensures that the shortest path is found efficiently.

**- Path Caching**: Path caching stores previously computed paths to reuse in future queries, dramatically reducing the need for recalculations. This method is particularly useful in a dynamic environment like a bus network, where the same paths might be queried repeatedly. The challenge lies in managing the cache to ensure it remains valid as the graph evolves.

**- Bidirectional Dijkstra**: This algorithm runs two simultaneous searches—one forward from the start node and one backward from the goal node. By meeting in the middle, Bidirectional Dijkstra can significantly cut down the search space and time compared to the unidirectional approach.

**- Contraction Hierarchy**: Contraction Hierarchies preprocess the graph to create a hierarchy of nodes by iteratively contracting nodes and adding shortcut edges. This preprocessing step allows for extremely fast query times during actual path searches, as the algorithm can skip over large sections of the graph using these shortcuts.

### 1.3. Implementation and Testing

To evaluate the effectiveness of these algorithms, we implemented them on the Ho Chi Minh City bus map graph. The datasets from "path.json", "stops.json", and "vars.json" provided a robust foundation for creating a realistic and complex testing environment. Through various test cases, each algorithm demonstrated substantial improvements in both speed and efficiency over the traditional Dijkstra’s algorithm. The A\* Search algorithm leveraged heuristics to minimize the search space, while Path Caching reduced redundant computations. Bidirectional Dijkstra cut down the search time by converging searches from both directions, and Contraction Hierarchy offered rapid query responses due to its preprocessing steps.

## **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:

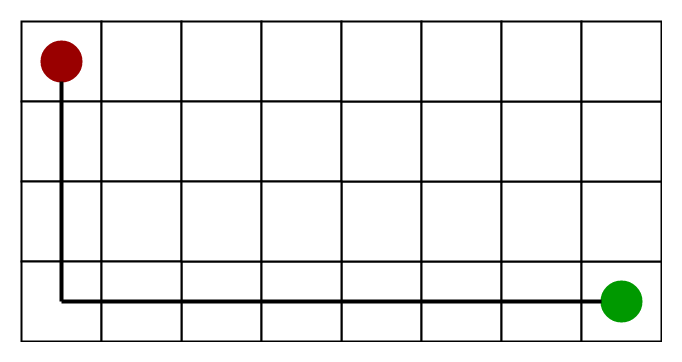
**2.2.1. Manhattan Distance**

* 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**

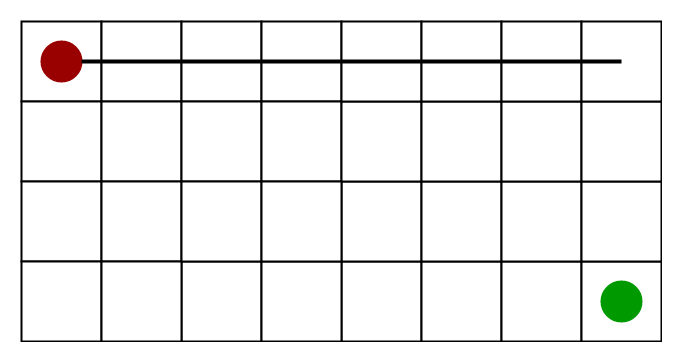
* It is nothing but the maximum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively, i.e.,

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).



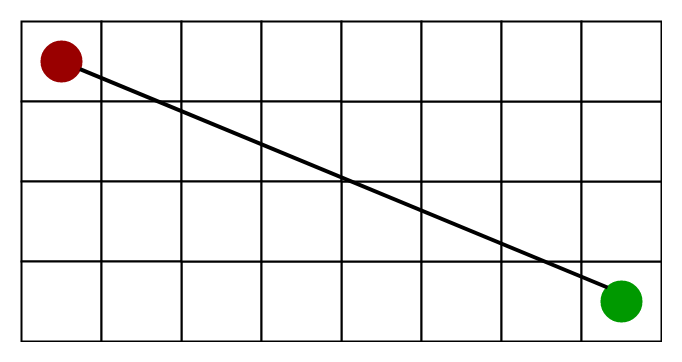
**2.2.3 Euclidean Distance-**

* 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:

A\* search time: 0.03341657948126125

[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: 0.0000012407

Total time: 0.03354907035827636719

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

Compare to the output of Dijsktra’s algorithm with the same input:

Dijkstra search time: 0.05217005729675292969

[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: 0.00100517272949218750

Total time: 0.05317523002624511719

For easier to display, I just print out the stop IDs of the stops that lie on this shortest path. The Route, RouteVar and all other information are stored in a dictionary named “stops” in order to extract information easily.

### 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.

## **Path Caching**

Path caching is a technique used to enhance the efficiency of algorithms tasked with finding the shortest paths in graphs. This method involves storing previously computed paths so they can be quickly retrieved when the same paths are needed again, rather than recalculating them from scratch. Path caching is particularly advantageous in applications with high-frequency, repetitive path queries, such as in navigation systems, network routing, and various optimization problems.

### 3.1. Algorithm

Since the stops in the bus map have the attributes “Zone”, and my observations indicate that the “Zone” in each stops is always a string (the zone is never empty) as well as that string is a district in Ho Chi Minh City. So, what I am trying to do here is to find the common shortest path in all pairs of districts, and these shortest path comes from my all-pair Dijkstra’s algorithm search.

For example, my search is coming to the stops having their stopID is 35 (source) and 3550 (end) and have got the shortest path through Dijsktra’s algorithm.

The algorithm will search for their zones in “stops” dictionary (which is ‘Quận 1’ and ‘Huyện Củ Chi’), then check if the cache has stored the path from these zones or not. If not, simply create a new key to store the list of resulting path as well as the time taken to get to 3550 from 35. Otherwise, I will try to get the common path between the newly created list path and the available path in the cache and store it along with the cost into the cache again.

Therefore, the basic algorithm is:

- For each stop u, run the Dijkstra’s algorithm to get the shortest path from u to all the other stops.

- Then, for each stop, we will get the resulting path and time consuming to get to the destination along with their zones.

- After that, we will look for the key in the cache for checking if the cache has stored the path from these zones or not.

+ If not, create a new key between the zones and store the list path into it.

+ Otherwise, get the common path bet between the newly created list path and the available path in the cache and store it along with the cost into the cache again.

When the searching ends, we store the cache that we have updated into a .JSON file named “cache.json”. So now, we have a file storing the shortest path between two zones and the time taken to go through it as well. In the Query section, we can reuse it by simply read it and apply the combined search to enhance the efficiency and time of the query.

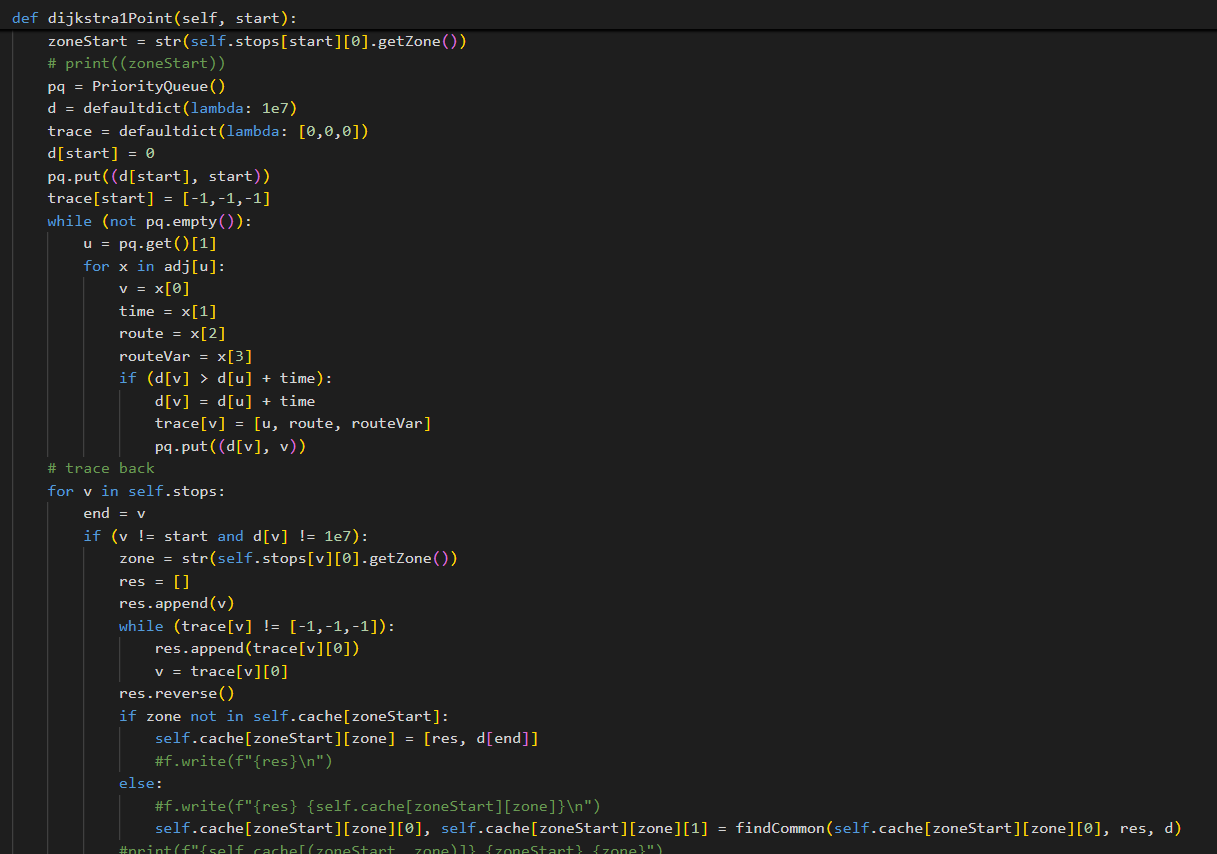
### 3.2. Implementation

Below picture is the implementation of the *caching()* function. It takes in the graph that we have done in CS162 (the *importGraph* function) and output a file with the details of the cache.

A screen shot of a computer program

Description automatically generated

Here is the breakdown of the *dijkstra1Point()* function:



It is simply Dijkstra’s algorithm combined with checking the zones and appending (or modifying) the key in the cache to store the shortest path between the districts.

The caching function has a complexity of , since we have to perform V seaching functions, which have complexity of .

The result of *caching* function is shown below. (due to the size of the file, I will just show a viewport of the file). If no common path in the two zones was found, the resulting cache will be [] and 0.

A screen shot of a computer screen

Description automatically generated

### 3.3. Query combined with A\* Search Algorithm

After creating a cache file for storing the shortest path between the districts in Ho Chi Minh City, we come to the query section for printing out the shortest path between two stops.

To do this, we firstly find the zones of the starting stop u and the ending stop v, then check if the shortest path between two zones is empty.

If this shortest path is empty, we just simply find the regular shortest path between two stops.

If not, we name the shortest path between two zones ‘path’. We will find the shortest path from u to path[0] as well as the shortest path from path[-1] to v, then simply add all the path that we found to the ‘path’ list.

So, what algorithm should we utilize to find the shortest path from u to path[0] and from path[-1] to v?

My answer is, A\* Search Algorithm. Compared with other search algorithms like Dijkstra’s algorithm or Floyd-Warshall algorithm, which 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.

Below figure is the implementation of the query combined with A\* Search Algorithm.

A screen shot of a computer program

Description automatically generated

This is the output when testing 3550 as the starting stop and 3683 as ending stop:

Path caching (combined with A\*):

[3550, 3552, 3553, 3555, 3554, 3556, 3557, 3558, 3559, 3560, 3561, 3563, 3562, 3564, 3565, 3566, 1185, 1206, 1205, 1207, 1208, 1211, 1209, 1212, 1210, 1214, 1216, 1213, 1218, 3232, 7608, 1215, 1222, 1217, 1219, 1224, 1225, 1165, 1221, 1223, 4746, 1227, 4747, 1228, 1230, 1232, 4588, 1231, 1235, 1234, 1393, 1239, 167, 172, 169, 174, 607, 610, 609, 611, 900, 898, 902, 2835, 2837, 2840, 1397, 2842, 2839, 2841, 2844, 466, 472, 467, 1051, 1347, 18, 22, 21, 23, 24, 25, 26, 1344, 1196, 1194, 1198, 1197, 1201, 1199, 1202, 1200, 1204, 835, 7070, 1257, 1260, 1261, 1258, 1356, 1357, 1359, 1361, 1362, 1358, 1360, 1366, 1055, 1058, 1061, 1063, 1060, 1065, 1062, 1066, 1064, 2138, 2137, 2141, 2139, 2144, 3662, 3664, 3663, 3666, 3665, 3667, 3669, 3670, 3671, 3683]

Time: 122.66109437702957

Query time: 0.00281405448913574219

Compared to regular A\* Search algorithm and Dijkstra’s algorithm, the query time has improved significantly since the cache worked effectively on the far points (the two zones are ‘Huyện Củ Chi’ and ‘Huyện Cần Giờ’, which is about 75 kilometers apart).

### 3.4. Benefits and drawbacks

In applications where queries are repeated, such network routing or guidance systems, path caching greatly improves the efficiency of algorithms for determining the shortest path. Path caching minimizes computing time and resource consumption by saving previously computed paths for easy retrieval and reuse. Better performance and quicker reaction times result from this, particularly in dynamic contexts where certain pathways are frequently queried.   
However, because storing pathways can be resource-intensive, path caching has disadvantages such as higher memory utilization and more complexity in maintaining cache validity. It can be difficult to guarantee that cached pathways stay current when the graph changes; complex methods are needed to prevent giving out-of-date or inaccurate paths. Additionally, the initial computation to populate the cache can introduce overhead, and integrating path caching into existing systems can add implementation complexity. Balancing these advantages and challenges is crucial for effectively utilizing path caching in practical applications.

## **Contraction Hierarchy**

The **Contraction Hierarchies (CH)** algorithm is a unique approach to computing shortest paths in large road network graphs. The work was originally presented in 2008 by Geisberger, Sanders, Schultes, and Delling and has since served as a springboard for other advanced route-planning algorithms. [2] shows an illustrative guide of the nature of Contraction Hierarchy, along with the examples and guides for implementation as well.

The concept of CH is simple. We can consider categorizing every edge, or node by its level of importance. Highways for example, might be considered more important than surface roads.

Then, when running Dijkstra’s algorithm from source to target, we could heuristically choose to settle only nodes that are endpoints of an edge with a higher level of importance than the edge we previously traveled on.

In doing so, we would hope to exclude a large number of nodes from ever being settled and effectively reduce our search space.

4.1. Bidiretional Dijkstra

A close-up of a network

Description automatically generatedBidirectional Dijkstra is an advanced variant of the traditional Dijkstra's algorithm, designed to enhance efficiency in finding the shortest path between two nodes in a graph. Unlike the standard approach, which expands nodes from the starting point until the destination is reached, Bidirectional Dijkstra simultaneously runs two searches: one forward from the start node and one backward from the goal node. These searches meet in the middle, effectively reducing the number of nodes that need to be explored and thus speeding up the search process.

To run two “simultaneous” rounds of Dijkstra’s algorithm, we maintain two priority queues and at each iteration either settle a node in the forward search, or settle a node in the backward search.

*Note that the backward search starting at the target node considers the transpose of the graph, where all edge directions are flipped.*

At each iteration, we compare the minimum node of the two priority queues, and we proceed with a round of Dijkstra in the priority queue with the smaller node.

We mark nodes that have already been settled by one of the searches, and when we settle a node that’s already been marked, our algorithm halts.

## **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)