07/2024

BIG GRAPH PROCESSING

Data Structures - CS163

Task 02: Advanced Shortest Path

Prepared By
Le Tien Dat

Student ID
23125028

Class
23APCS2

www.facebook.com/ltd.sword
ltdat23@apcs.fitus.edu.vn
University of Science, VNU-HCM

Contents

1. 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.

2. A* Search Alogorithm

Developed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael, the A* algorithm 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.

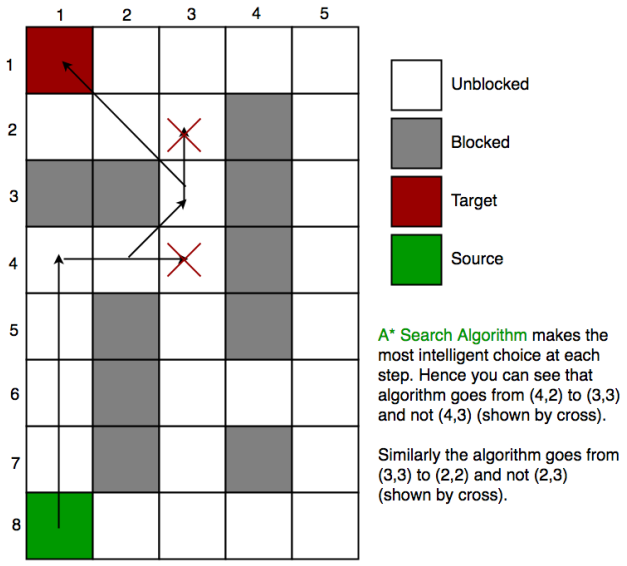
2.1. Algorithms

```
// 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

*** Approximation Heuristics*

There are generally three approximation heuristics to calculate h:

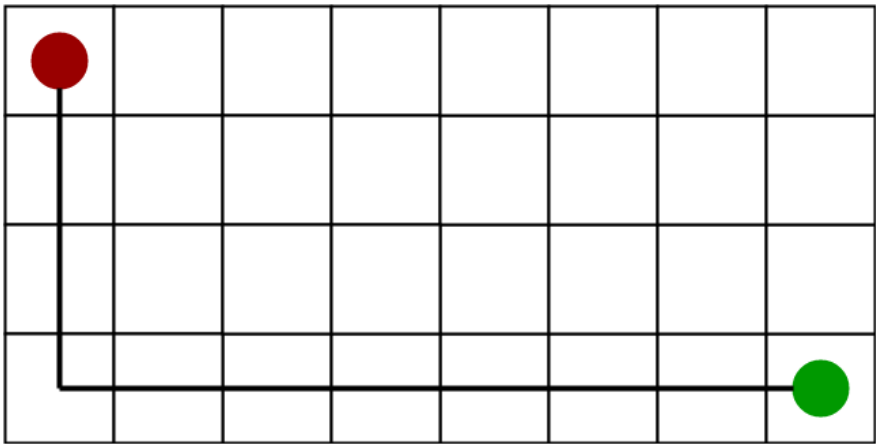
2.2.1. Manhattan Distance

- 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

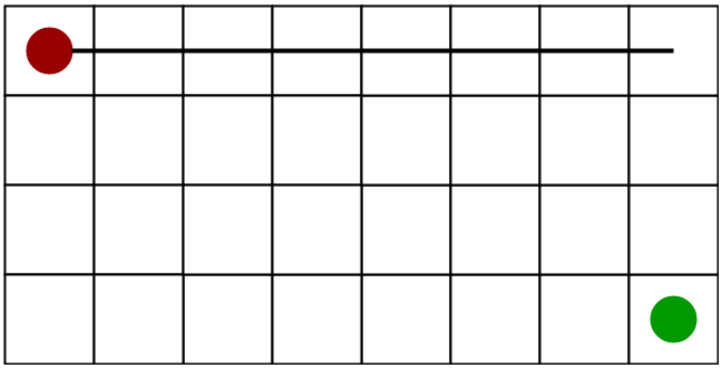
- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

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).



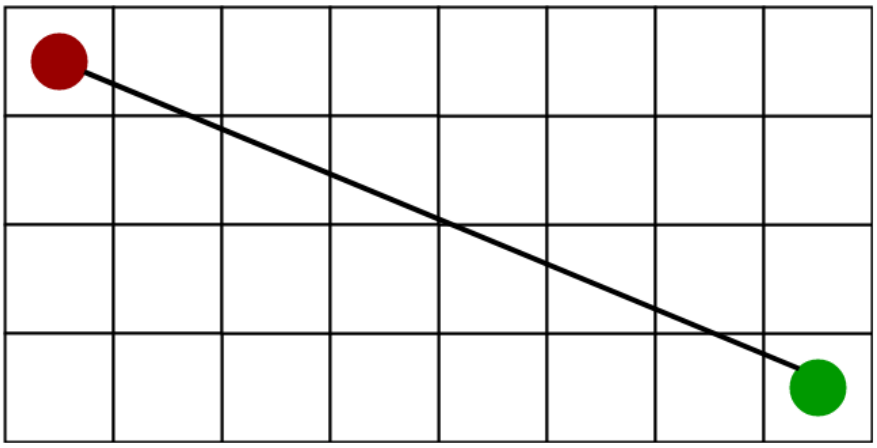
2.2.3 Euclidean Distance-

- 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)

```
while (not pq.empty()):
    f_u, u = pq.get()
    if (u == end):
        break
    for x in adj[u]:
        #if (trace[x[0]] == [0, 0, 0]):
            v = x[0]
            # if (v == end):
            #     flag = True
            time = x[1]
            route = x[2]
            routeVar = x[3]
            tentative_time = d[u] + time
            if (tentative_time < d[v]):
                trace[v] = [u, route, routeVar]
                d[v] = tentative_time
                #print(d[v])
                ftmp = tentative_time + h(v, end, route, routeVar)
                if (v in closed and ftmp >= closed[v]): continue
                else:
                    pq.put((ftmp, v))
                    if (v in closed): del closed[v]
    closed[u] = f_u
    # if (flag): break
```

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.

3. 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.

```
def caching(self, filename = ""): # filename to save the cache
    cur = times.time()
    # we run all-pair dijkstra to find the shortest path between all pairs of stops
    self.graph.importGraph("Questions/vars.json", "Questions/stops.json", "Questions/paths.json")
    rvq = RouteVarQuery()
    self.vel = rvq.loadDistTime("Questions/vars.json")
    self.adj = self.graph.getAdjacent()
    self.stops = self.graph.getVertices()
    # get the position for each stop
    # convert from lat lng to x y
    target_crs = CRS.from_epsg(3405)
    source_crs = CRS.from_epsg(4326)
    proj = Transformer.from_crs(source_crs, target_crs)
    trans = lambda x,y: proj.transform(y, x)

    for key in self.stops:
        self.stopPos[key] = trans(self.stops[key][0].getLng(), self.stops[key][0].getLat())

    #f = open("test.txt", "w")
    for start in self.stops:
        self.dijkstra1Point(start)
    #f.close()

    print(f"Total time (all pairs): {times.time() - cur:.20f}")

    # now we've found the cache for all pairs of stops

    # modify the structure of the cache to be fit with the json format
    #if (35 in self.cache): del self.cache[35]

    with open(filename, "w", encoding='utf8') as outfile:
        json.dump(self.cache, outfile, ensure_ascii=False)
```

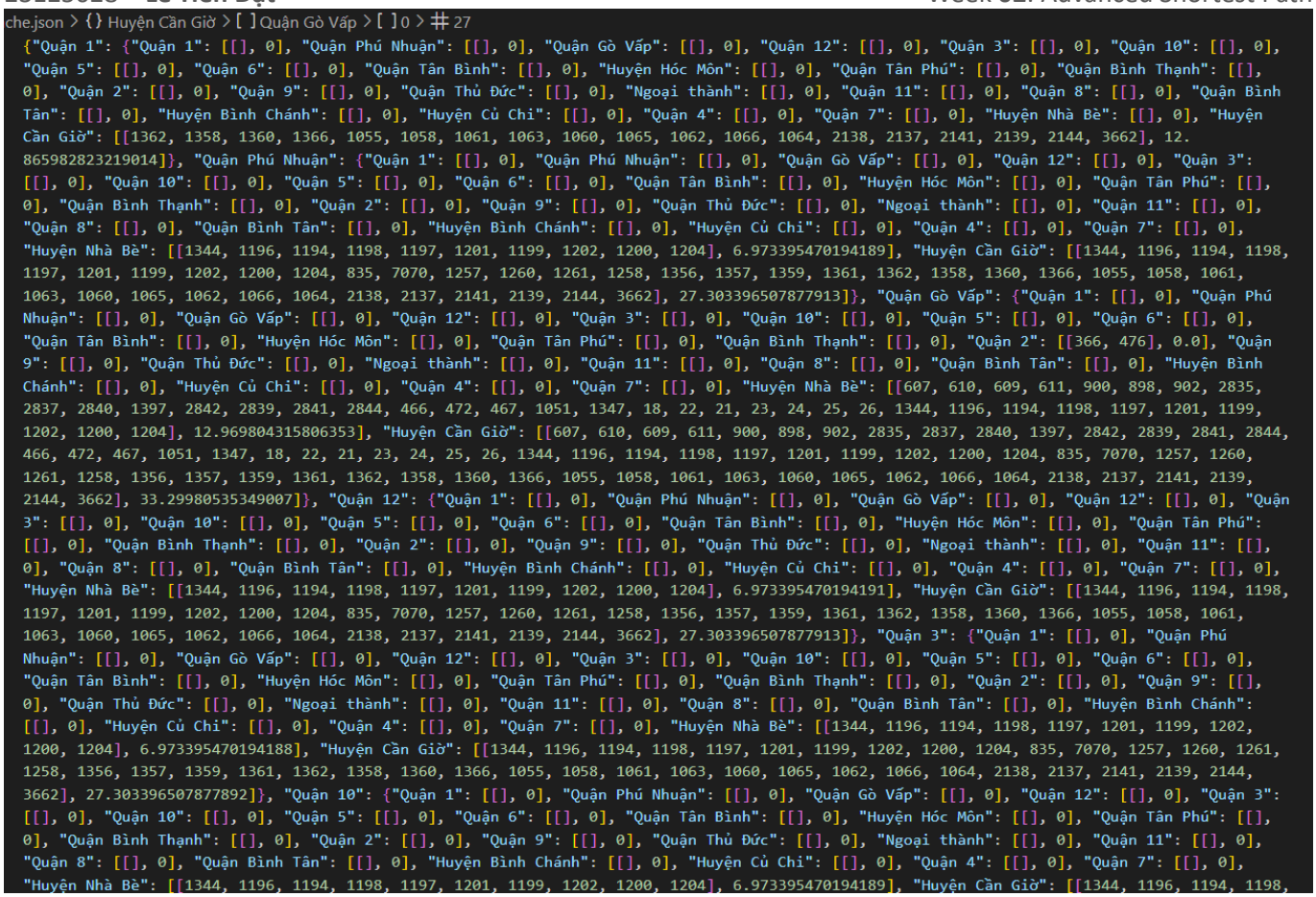
Here is the breakdown of the *dijkstra1Point()* function:

```
def dijkstra1Point(self, start):
    zoneStart = str(self.stops[start][0].getZone())
    # print((zoneStart))
    pq = PriorityQueue()
    d = defaultdict(lambda: 1e7)
    trace = defaultdict(lambda: [0,0,0])
    d[start] = 0
    pq.put((d[start], start))
    trace[start] = [-1,-1,-1]
    while (not pq.empty()):
        u = pq.get()[1]
        for x in adj[u]:
            v = x[0]
            time = x[1]
            route = x[2]
            routeVar = x[3]
            if (d[v] > d[u] + time):
                d[v] = d[u] + time
                trace[v] = [u, route, routeVar]
                pq.put((d[v], v))
    # trace back
    for v in self.stops:
        end = v
        if (v != start and d[v] != 1e7):
            zone = str(self.stops[v][0].getZone())
            res = []
            res.append(v)
            while (trace[v] != [-1,-1,-1]):
                res.append(trace[v][0])
                v = trace[v][0]
            res.reverse()
            if zone not in self.cache[zoneStart]:
                self.cache[zoneStart][zone] = [res, d[end]]
                #f.write(f"{res}\n")
            else:
                #f.write(f"{res} {self.cache[zoneStart][zone]}\n")
                self.cache[zoneStart][zone][0], self.cache[zoneStart][zone][1] = findCommon(self.cache[zoneStart][zone][0], res, d)
            #print(f"{self.cache[(zoneStart, zone)]} {zoneStart} {zone}")
```

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 $O(V.ElogV)$, since we have to perform V seaching functions, which have complexity of $O(ElogV)$.

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.



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.

```
def query(self, start = 0, end = 0):
    cur = times.time()
    zoneStart = str(self.stops[start][0].getZone())
    zoneEnd = str(self.stops[end][0].getZone())
    if (zoneEnd in self.cache[zoneStart] and self.cache[zoneStart][zoneEnd][1] != 0):
        lst = self.cache[zoneStart][zoneEnd]
        mid1 = lst[0][0]
        mid2 = lst[0][-1]
        time = lst[1]
        path1 = self.aStarSearch(start, mid1)
        path2 = self.aStarSearch(mid2, end)
        res = path1[0] + lst[0][1:-1] + path2[0]
        time += path1[1] + path2[1]
        print(res)
        print(f"Time: {time}")
    else:
        path = self.aStarSearch(start, end)
        print(path[0])
        print(f"Time(no caching): {path[1]}")
    print(f"Query time: {times.time() - cur:.20f}")
```

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.

4. 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 *(this section also take a lot of consideration from that guide)*.

The concept of CH is simple. It uses a process of ***node contraction*** to create a hierarchy in which every node belongs to a unique level (in other words, an ordering of nodes based on importance).

During the contraction process, a set of shortcut edges is added to the graph that preserves shortest paths. The bidirectional search then only considers the subset of edges that lead from less important to more important nodes, and these edges might include the shortcuts we added. The result is faster querying with only a modest increasing in preprocessing compared to Highway Node Routing.

So we can present a full overview of the CH algorithm as a reference, and the sections that follow give an analysis of each core component in greater detail.

The High-Level CH Algorithm:

Preprocess*:*

- *Order nodes based on some computed level of importance.*
- *Contract each node in order of least important to most important.*

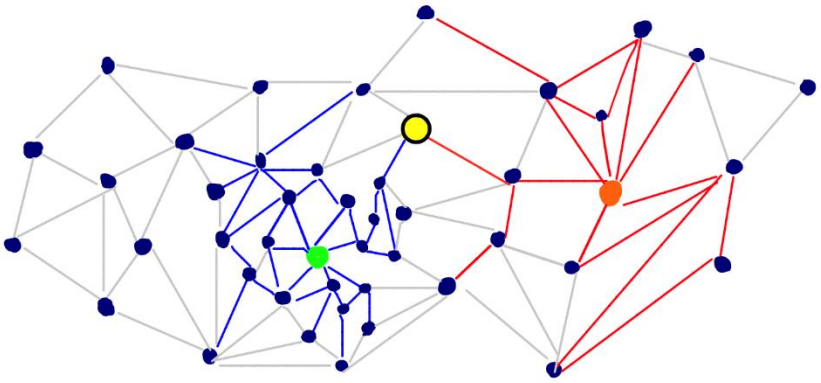
Query*:*

- *Run a bidirectional Dijkstra search, but only consider edges leading to nodes with a higher importance.*

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

Bidirectional 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.

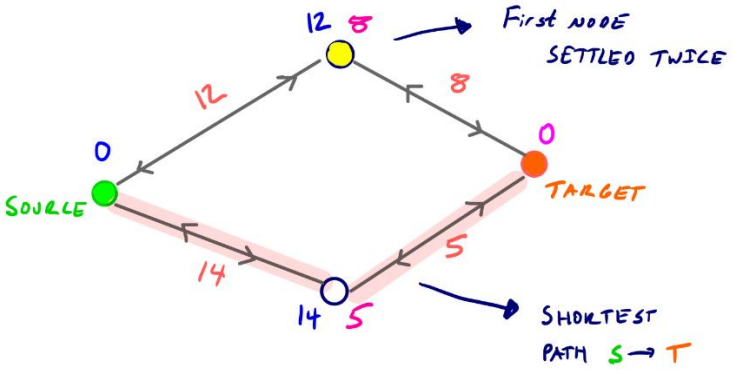
Note that the backward search starting at the target node considers the transpose of the graph, where all edge directions are flipped.

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.

What’s the Shortest Path?

Is the node settled by both searches on the shortest path? It could be, but not always. Consider this counter example:



At each iteration we settle the smaller of the minimum nodes in the two priority queues. So we first settled the yellow node in the reverse search, then the bottom node in the reverse search, and then the yellow node again in the forward search.

So our search would halt since we've settled a node in both searches now, but the actual shortest path goes along the pink shaded edges.

The actual shortest path, then, is defined as:

*For all **u** reached by both searches, dist(s, t) = **min{dist(s,u) + dist(t,u)}***

This means that we look at all vertices with finite shortest path scores from both *s* and *t*, and take the minimum sum of the two scores. So in the example above, the shortest path goes along the pink edges because the sum of the two shortest path scores for the lower vertex is 19, which is less than the sum for the vertex settled twice.

Using bidirectional search and the hierarchy structure, we can get the effective algorithm.

4.2. Preprocess Phase

The preprocess phase consists of two main processes: choose a node order and node contracting. Now, we'll go deeper into these processes and their way to implement the code.

4.2.1. Choosing a Node Order

Our query is correct no matter the order in which nodes were contracted, but a good node ordering has major implications for the performance of the queries (see the proof in [5]). The order in which nodes are contracted affects the shortcut edges that do or don't get added in our graph. And too many shortcut edges means a too dense graph and slower queries as a result.

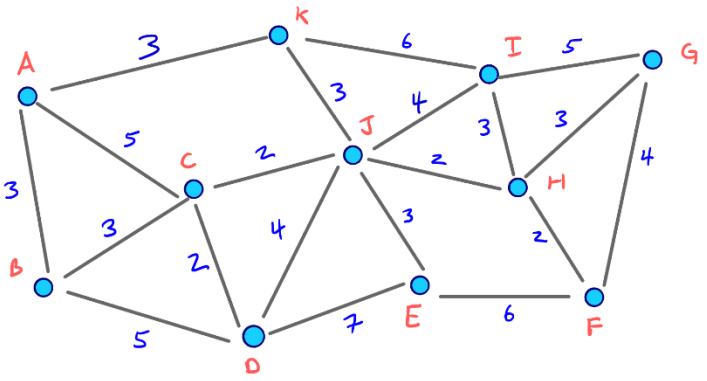
So a primary concern with contraction hierarchies is determining a good node ordering during the preprocess step. In turns out that finding an optimal node ordering is hard (see [6]). A lot of continued research and developments to contraction hierarchies deals with finding better heuristics for good node ordering. However, the approaches given by [7] do perform very well in practice.

In short, there are many ways to choose a node order, from simple to complicated. Here I introduce two popular solutions for this choosing problem:

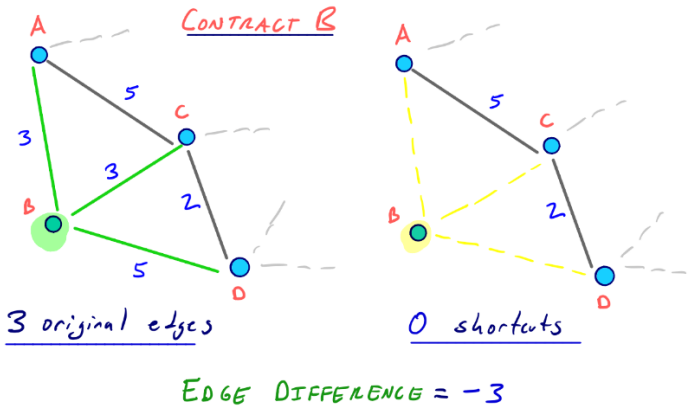
* **Degree-based order**: Nodes are ordered based on their degree (number of edges), which means that the nodes with fewer edges are contracted first. This is one of the simplest solutions to heuristic that reduces the number of shortcuts. However, we can go further than this way, I mean, make it more efficient in the number of shortcuts and number of ordering using **Edge Difference**.

* **Edge Difference** (ED): In this way, nodes are contracted based on the edge difference between the number of original edges and the shortcut edges if the node is contracted. The nodes which have fewest ED values will be contracted first.

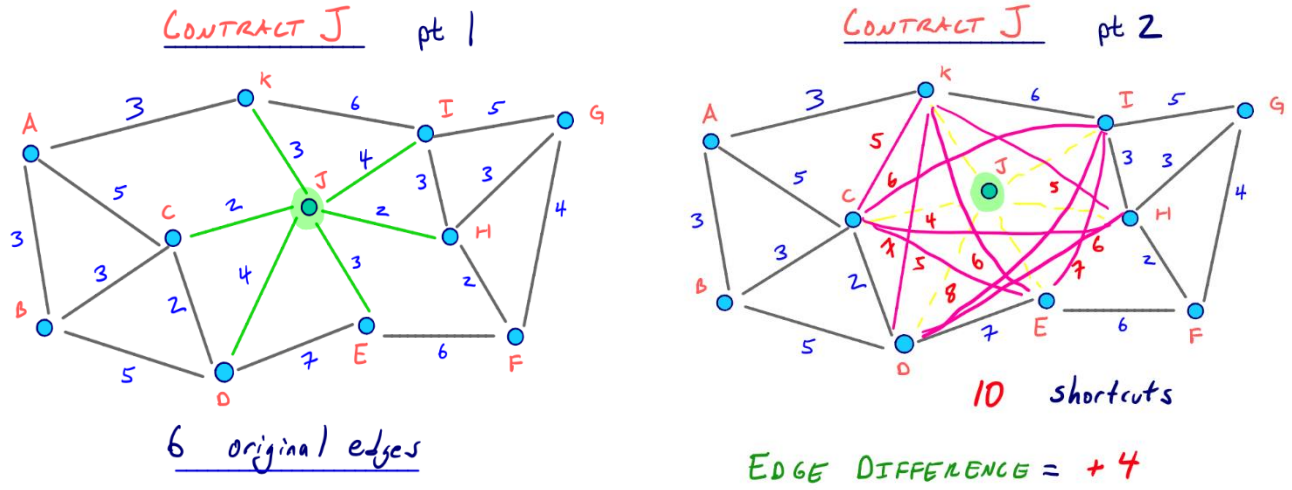
Let's consider this example:



If we try to contract B (the contract method will be proposed in the next section), the edge difference is -3 as the figure below.



But if we try to contract J, the edge difference will be +4:



So clearly *J* has a much greater edge difference than *B*, and so *J* is less attractive to contract early on. Nodes like *J* are hubs that connect many of its neighbors with low costing edges. Its adjacent edges are like highways that should be used in the middle of long trips, so it would make sense that a node like *J* is *important* and shouldn't be contracted early on if we want the best possible query performance.

Then a primary method for determining a node order and subsequently contracting all nodes in that order involves maintaining a priority queue based on some cost function, which in our case is simply the *edge difference* of each node.

So first, to load the priority queue, we can compute the edge difference of each node by *simulating* its contraction. In the next section about the node contraction, we'll know that we're doing a constant amount of work per node, which initially gives us a linear amount of work for this step.

But consider that after we begin contracting nodes, the edge difference for other nodes can be affected. If we wanted to strictly adhere to an ordering based on edge difference, we would need to recompute the edge difference for every remaining node in the graph after each contraction. Of course, this would end up taking quadratic time, so it isn't feasible. Instead, we can use the ***lazy update*** heuristic.

* **Lazy updating**: Consider that we've already computed our initial node ordering. Now we're in the process of actually contracting each node in order, by extracting the min from our priority queue.

Before contracting the next minimum node, we recompute its edge difference. If it's still the smallest in the priority queue, we can go ahead and actually contract it. If its edge difference is no longer the min, then we update its cost and rebalance our PQ. We then check the next minimum node and continue this process. [8] and [9] find that performing these lazy updates results in improved query times because of a better, updated contraction order.

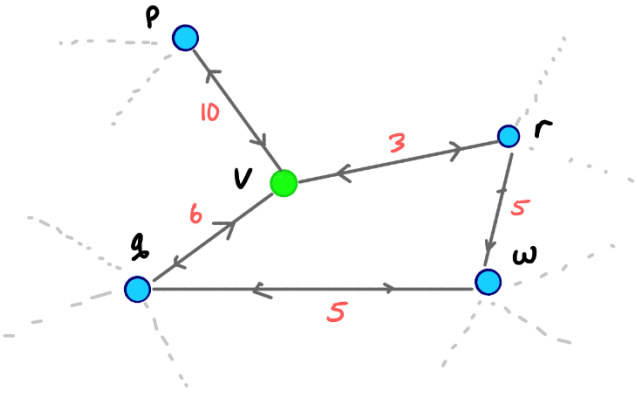
Other heuristics used to update the cost values in the priority queue involve only recomputing edge differences for *neighbors* of the most recently contracted node. We could also decide to do a full re-computation of edge differences for all remaining nodes if too many lazy updates occur in successive order.

Next, we move to the most important part: contract a node.

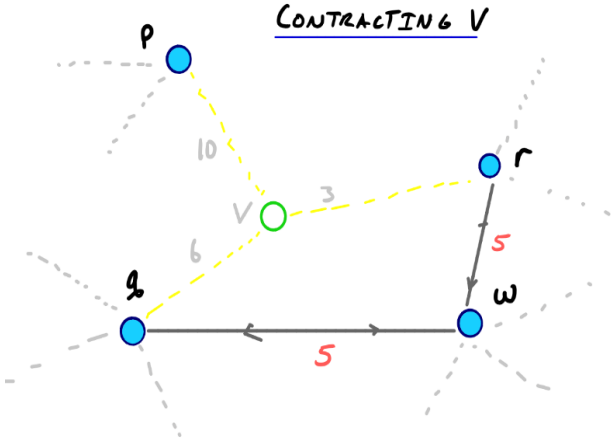
4.2.2. Node contraction

When we *contract* a node, we first remove it from the graph. If the contracted node existed on the shortest path between two of its neighbors before contraction, we add a shortcut edge between the two neighbors such that all shortest path lengths are still preserved.

Consider this small example, where we contract node *v*. We can assume that no other edges exist between *v*, *p*, *q*, and *r*, other than the ones shown.



First *v* and its adjacent edges are dropped from the graph. Then we check if any shortest paths between two neighbors of *v* actually went through *v*. In this case, the shortest paths between *p* and *r*, *q* and *r*, and *p* and *q* all used *v*, so we add the shortcut edges *pr*, *qr*, and *pq* with corresponding weights.



Notice that even though a path from *q* to *r* still existed after removing *v* and its incident edges, it wasn't the shortest path from *q* to *r*, so our shortcut edge *qr* is needed.

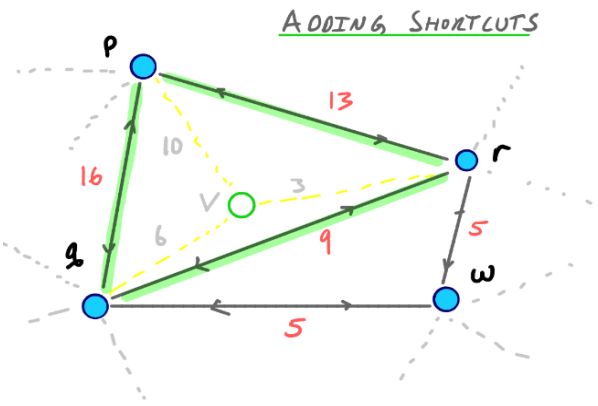
Formal Definition

Consider a node *v* and the following contract process:

We denote the set of all vertices with edges incoming to *v* as *U*, and the set of all vertices with incoming edges from *v* as *W*.

Then for each pair of vertices *(u, w)*, for *u* in *U* and *w* in *W*:

If the path <uvw> is the unique shortest path from u to w, we add a shortcut edge uw to the graph with weight w(u, v) + w(v, w).



Then, we remove *v* and all of its adjacent edges from the graph.

We're now done contracting *v*.

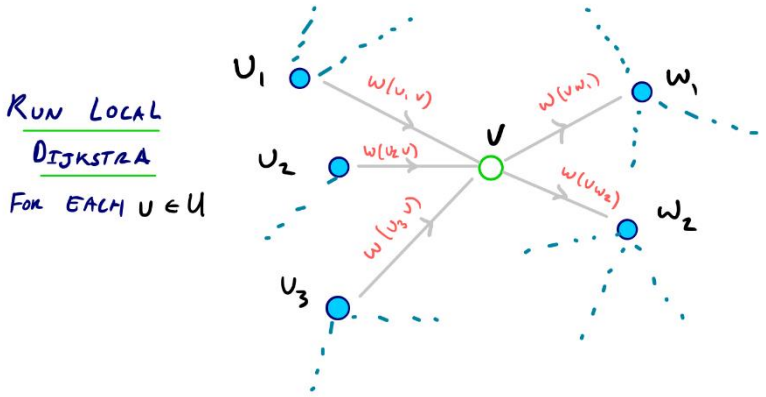
After contracting node v_n we consider the overlay graph ***G**** that contains all original nodes and edges and all shortcut edges added during the contraction process.

We use *G** for our bidirectional Dijkstra searches.

Then, we come to the approaches to add a shortcut between two nodes u and v.

A straightforward way to determine which shortcuts are needed is to run a series of local shortest-path searches from every node *u* in *U* with *v* excluded. If we reach a node *w* in *W* with a distance less than *w(u, v) + w(v, w)*, then we've just found a path *witnessing* that <*uvw*> is not the best way to get from *u* to *w*.

On the other hand, if <*uvw*> ***is*** the unique shortest path from *u* to *w* and moreover is the the *only* path from *u* to *w*, our local search won't terminate since we're considering the subgraph that excludes *v*.



So we can do the following **for every *u* in *U***:

1. *For every node w in W, compute P_w as the cost from u to w through v, which is the sum of the edge weights w(u, v) + w(v, w).*
2. *Then P_{max} is the maximum P_w over all w in W.*
3. *Perform a standard Dijkstra's shortest-path search from u on the subgraph excluding v.*
4. *Once a node is settled with a shortest-path score greater than P_{max}, we stop.*

Then for each *w*, if *dist(u, w) > P_w* we add a shortcut edge *uw* with weight P_w.

If this condition doesn't hold, no shortcut is added.

4.2.3. Implementation

Below figues are the implementation of preprocessing phase for both the degree-based approach and edge difference approach.

```python
def precompute(self, mode = ""):
    if (mode == 'degree'):
        cur = times.time()
        # for simple, we use the degree of the node as the rank of the node, which is the number of neighbors of the node
        # ascending in the deg of nodes.
        for node in self.nodes:
            self.rank[node] = len(self.graph.adj[node]) + len(self.graph.rev[node])
        rank = sorted(self.rank, key = self.rank.get)
        idx = 1
        for node in rank:
            self.rank[node] = idx
            idx += 1
        self.order = sorted(self.rank, key = self.rank.get)

        initadj = self.graph.adj.copy()
        initrev = self.graph.rev.copy()

        # we contract all the node in the order to get the shortcut
        for node in self.order:
            start = times.time()
            self.contract(node)
            print(f"Contract node {node} in {times.time() - start:.20f}")

        self.graph.adj = initadj.copy()
        self.graph.rev = initrev.copy()

        # add the shortcuts to the graphs
        self.addShortcut()

        print(f"Precomputation time: {times.time() - cur:.20f}")
```

This is the self.addShortcut() method:

```python
def addShortcut(self):
    # add the shortcut edges to the initial graph
    for sc in self.shortcut:
        idx = bisect_left(self.graph.adj[sc[0]], (sc[1], self.shortcut[sc][1], 0, 0))
        if (idx < len(self.graph.adj[sc[0]]) and self.graph.adj[sc[0]][idx][0] == sc[1]):
            self.graph.adj[sc[0]][idx] = (sc[1], self.shortcut[sc][1], 0, 0)
        else:
            self.graph.adj[sc[0]].insert(idx, (sc[1], self.shortcut[sc][1], 0, 0))

        idx = bisect_left(self.graph.rev[sc[1]], (sc[0], self.shortcut[sc][1], 0, 0))
        if (idx < len(self.graph.rev[sc[1]]) and self.graph.rev[sc[1]][idx][0] == sc[0]):
            self.graph.rev[sc[1]][idx] = (sc[0], self.shortcut[sc][1], 0, 0)
        else:
            self.graph.rev[sc[1]].insert(idx, (sc[0], self.shortcut[sc][1], 0, 0))
```

In order to make the graph less dense, I have sorted all the stops in the adjacency list and utilize binary search to this list. If existing an edge that the shortcut is more optimal, I will replace that edge by the shortcut; otherwise, just simply insert that shortcut into the graph.

```
elif (mode == 'edgeDiff'):
    pqNode = PriorityQueue()
    cur = times.time()
    # we compute the edge difference of the node
    for node in self.nodes:
        edgeDiff = self.computeEdgeDiff(node)
        pqNode.put((edgeDiff, node))
        self.countDiff[node] = edgeDiff

    # now we have the nodes in the order of edge difference
    # we combine contracting the node along with lazy updating of the edge difference

    initadj = self.graph.adj.copy()
    initrev = self.graph.rev.copy()

    idx = 1
    while (not pqNode.empty()):
        start = times.time()
        tmp = pqNode.get()
        node = tmp[1]
        ed = tmp[0]
        edgeDiff = self.computeEdgeDiff(node)
        #edgeDiff = self.countDiff[node]
        if (pqNode.empty() or edgeDiff <= pqNode.queue[0][0]):
            self.order.append(node)
            self.rank[node] = idx
            idx += 1
            self.contract(node)
            print(f"Contract node {node} in {times.time() - start:.20f}")
        else:
            pqNode.put((edgeDiff, node))

        if (len(pqNode.queue) <= 2):
            print(len(pqNode.queue))
```

```
    self.graph.adj = initadj.copy()
    self.graph.rev = initrev.copy()

    # add the shortcuts to the graphs
    self.addShortcut()
    print(f"Number of shortcuts: {len(self.shortcut)}")
    print(f"Precomputation time: {times.time() - cur:.20f}")

else:
    # default mode is degree
    self.precompute(mode = 'degree')
```

And the most important method, the contract() method. Below pictures are the construction of the main Dijkstra's algorithm of checking the shortest path from u and adding new shortcut to the adjacency list:

```
for u in backward:  # u = (stopId, time, route, routeVar)
    self.countDiff[u[0]] -= 1
    time = times.time()
    maxP = 0
    d = [1e9]*9000
    count = [0]*9000
    for v in forward:
        count[v[0]] += 1
    d[u[0]] = 0
    d[node] = self.graph.getTime(u[0], node)
    for v in forward: # v = (stopId, time, route, routeVar)
        maxP = max(maxP, self.graph.getTime(u[0], node) + self.graph.getTime(node, v[0]))

    # perform a standard dijkstra from u to the subgraph excluding the node
    pq = PriorityQueue()
    pq.put((d[u[0]], u[0]))
    while not pq.empty():
        point = pq.get()[1]
        for v in self.graph.adj[point]:
            count[v[0]] += 1
            if (v[0] == node):
                continue
            if d[v[0]] > d[point] + v[1]:
                d[v[0]] = d[point] + v[1]
                pq.put((d[v[0]], v[0]))

            # if a node is settled, we check the value of the node with maxP
            # if the value of the node is greater than maxP, we can break the loop
            if count[v[0]] == len(self.graph.rev[v[0]]):
                if d[v[0]] > maxP:
                    break
```

Adding new shortcut to the adjacency list:

```
for v in forward: # v = (stopId, time, route, routeVar)
    self.countDiff[v[0]] -= 1
    if u[0] == v[0]:
        continue

    # add a shortcut from u to v
    if (d[v[0]] > self.graph.getTime(u[0], node) + self.graph.getTime(node, v[0])): # if the shortcut is better than the path
        # check if there is a shortcut from u to v
        # if we modify the shortcut, we need to store the new shortcut along with the point that the shortcut is from
        # for example, A -> B -> C, if we want to add a shortcut from A to C, we need to store B in the shortcut
        # so, the shortcut structure is shortcut[(A,C)] = [B, time]
        if (u[0], v[0]) in self.shortcut:
            shortcut = self.shortcut[(u[0], v[0])]
            if shortcut[1] > u[1] + v[1]:
                # update the shortcut
                shortcut[0] = node
                shortcut[1] = u[1] + v[1]
                if ((u[0], node) in self.shortcut): # from u to node is already a shortcut
                    self.shcutRoute[(node, v[0])] = (v[2], v[3]) # we just need to update the route and routeVar from node to v
                                                                 # since the route and routeVar from u to node has been stored in
                                                                 the shchtRoute
                elif ((node, v[0]) in self.shortcut):
                    self.shcutRoute[(u[0], node)] = (u[2], u[3])
                else:  # if there is no shortcut from u to node and from node to v
                    self.shcutRoute[(u[0], node)] = (u[2], u[3])
                    self.shcutRoute[(node, v[0])] = (v[2], v[3])
                self.shortcut[(u[0], v[0])] = shortcut

                # add newly added shortcut to the graphs
                # just need to update the list, not add a new one
                idx = bisect_left(self.graph.adj[u[0]], (v[0], u[1] + v[1], 0, 0))
                if (idx < len(self.graph.adj[u[0]])):
                    self.graph.adj[u[0]][idx] = (v[0], u[1] + v[1], 0, 0)
                    self.graph.time[(u[0], v[0])] = u[1] + v[1]
```

```
            idx = bisect_left(self.graph.rev[v[0]], (u[0], u[1] + v[1], 0, 0))
            if (idx < len(self.graph.rev[v[0]])):
                self.graph.rev[v[0]][idx] = (u[0], u[1] + v[1], 0, 0)
        else:
            self.shortcut[(u[0], v[0])] = [node, u[1] + v[1]]
            self.countDiff[v[0]] += 1
            self.countDiff[u[0]] += 1
            if ((u[0], node) in self.shortcut): # from u to node is already a shortcut
                self.shcutRoute[(node, v[0])] = (v[2], v[3]) # we just need to update the route and routeVar from node to v
                                                             # since the route and routeVar from u to node has been stored in the
                                                             shchtRoute
            elif ((node, v[0]) in self.shortcut):
                self.shcutRoute[(u[0], node)] = (u[2], u[3])
            else:  # if there is no shortcut from u to node and from node to v
                self.shcutRoute[(u[0], node)] = (u[2], u[3])
                self.shcutRoute[(node, v[0])] = (v[2], v[3])

            # add newly added shortcut to the graphs
            idx = bisect_left(self.graph.adj[u[0]], (v[0], u[1] + v[1], 0, 0))
            if (idx < len(self.graph.adj[u[0]]) and self.graph.adj[u[0]][idx][0] == v[0]):
                self.graph.adj[u[0]][idx] = (v[0], u[1] + v[1], 0, 0)
            else:
                self.graph.adj[u[0]].insert(idx, (v[0], u[1] + v[1], 0, 0))
            self.graph.time[(u[0], v[0])] = u[1] + v[1]
            idx = bisect_left(self.graph.rev[v[0]], (u[0], u[1] + v[1], 0, 0))
            if (idx < len(self.graph.rev[v[0]]) and self.graph.rev[v[0]][idx][0] == u[0]):
                self.graph.rev[v[0]][idx] = (u[0], u[1] + v[1], 0, 0)
            else:
                self.graph.rev[v[0]].insert(idx, (u[0], u[1] + v[1], 0, 0))
```

After having found the shortcuts in the preprocess phase, we can also save the shortcuts and other attributes to a .JSON file for easier to handle. Next time, we can just take out the shortcuts from file and use the query without consuming time in doing the preprocessing time, which turns out to be a little bit long.

For example, this is the figure about the preprocessing time:

```
Query time: 0.00200748443603515625
Total time: 53.560655355453491210 94
```

Visit my repository for more information about implementation in [8].

4.3. Query Phase

So consider now that during our preprocessing stage we:

1. Ordered all nodes
2. Contracted each node in order
3. Have an overlay graph G* containing all original nodes and edges, and also all shortcut edges added during the contraction phase.

Now we want to query our graph to find the shortest path between a source *s* and a target *t*.

The CH query uses a modified version of the bidirectional Dijkstra search we introduced earlier, but it considers only smaller subgraphs of G* in both directions.

Given our order of nodes, we denote for nodes *v* and *w* that ***v > w*** if *v* was contracted **after** *w*. And we say that ***v < w*** if *v* was contracted **before** *w*.

Then we can define the upward and downward graphs:

*The **upward graph** G^*_U only contains edges from v to w where v > w.*

*The **downward graph** G^*_D only contains edges from v to w where v < w.*

Because every node has a unique position in the ordering, every edge is either in the upward or downward graph.

The Bidirectional Search Spaces

Recall that in the original bidirectional search method, we run a forward search from the source and a backward search from the target. For our CH query from *s* to *t*, we run a forward search from *s* on the G^*_U graph, and a backward search from *t* on the G^*_D graph.

However, for symmetric graphs like the ones we are considering, a backward search (reversing edge directions) on G^*_D is the same as an upward search on G^*_U. **So from both our source and**

target nodes, we can perform a standard Dijkstra search on the G^*_U graph, which means that we can just go on the higher priority nodes in both sources.

Finding the Shortest Path Length

In any case, after running two complete Dijkstra searches on G^*_U from both the source and target, we have a set of nodes that are settled in both searches. We denote this set as *L*.

For every node *v* in *L*, we sum the shortest path scores of *v* (one score from *s*, one from *t*). The shortest path distance then is the minimum sum over all *v* in *L*.

In other words:

dist(s, t) *= min{ dist(s, v) + dist(v, t)} over all v in L*

One more thing is, can we get the actual arcs, or edges of the shortest path?

We're able to compute the *length* of the shortest path, but we still need to unpack the actual arcs used on that path. We can handle this during the stage of contraction when shortcut edges are added. When we add a shortcut edge from *u* to *w* during the contraction of *v*, we store a shortcut pointer to *v*.

We then begin unpacking the shortest path on the G^*_U graph from the highest order node, in both directions. We back-trace parent edge pointers as in a regular Dijkstra algorithm, and if the parent edge has a shortcut pointer, we replace the parent edge with the shortcut edge, and continue to recursively unpack the full path.

In fact, the attribute self.shortcuts has the structure:

`shortcut = {(node 1, node 2): [mid node, time]}`

which stores the middle node to unpack the shortest path easily. We can also store the route and routeVar attributes into a dictionary for easier to recall them in needing to print out.

This is the implementation of unpack() method using recursion:

```python
def unpack(self, u, v, mode = "") -> list:
    # unpack the path from u to v
    if (mode == "adj"):
    # return a reverse path from u to v for easier to append to the result
        if (u, v) in self.shortcut:
            shortcut = self.shortcut[(u, v)]
            return self.unpack(shortcut[0], v, mode) + self.unpack(u, shortcut[0], mode)
        else:
            return [u]
    elif mode == "rev":
        if (u, v) in self.shortcut:
            shortcut = self.shortcut[(u, v)]
            return self.unpack(u, shortcut[0], mode) + self.unpack(shortcut[0], v, mode)
        else:
            return [v]
    else: return self.unpack(u, v, mode="adj")
```

Combining all the things that are mentioned above, we come to the implementation of the query in CH:

```
def compareQueue(a, b):
    if (len(a.queue) == 0):
        return 'rev'
    elif (len(b.queue) == 0):
        return 'adj'
    else:
        return 'adj' if a.queue[0][0] < b.queue[0][0] else 'rev'
```

```
while (not pqStart.empty() or not pqStop.empty()):
    if (compareQueue(pqStart, pqStop) == 'adj'):
        d_u, u = pqStart.get()
        if (u in visitedStop):
            intersect = u
            for node in visitedStop:
                if (dStart[intersect] != 1e8 and dStart[intersect] + dStop[intersect] > dStart[node] + dStop[node]):
                    intersect = node
            for node in visitedStart:
                if (dStop[intersect] != 1e8 and dStart[intersect] + dStop[intersect] > dStart[node] + dStop[node]):
                    intersect = node
            break
        for v in self.graph.adj[u]:
            if dStart[v[0]] > d_u + v[1] and self.rank[u] < self.rank[v[0]]:
                visitedStart.add(v[0])
                dStart[v[0]] = d_u + v[1]
                pqStart.put((dStart[v[0]], v[0]))
                traceStart[v[0]] = [u, v[2], v[3]]
    else:
        d_u, u = pqStop.get()
        if (u in visitedStart):
            intersect = u
            for node in visitedStop:
                if (dStart[intersect] != 1e8 and dStart[intersect] + dStop[intersect] > dStart[node] + dStop[node]):
                    intersect = node
            for node in visitedStart:
                if (dStop[intersect] != 1e8 and dStart[intersect] + dStop[intersect] > dStart[node] + dStop[node]):
                    intersect = node
            break
        for v in self.graph.rev[u]:
            if dStop[v[0]] > d_u + v[1] and self.rank[u] < self.rank[v[0]]:
                visitedStop.add(v[0])
                dStop[v[0]] = d_u + v[1]
                pqStop.put((dStop[v[0]], v[0]))
                traceStop[v[0]] = [u, v[2], v[3]]
```

This is the traceback algorithm:

```
    if (mode == 'display'):
        print(f"Intersect at node {intersect}")
    # trace back to get the path
    res = []
    v = intersect
    if (intersect == 0):
        v = stop
        while (traceStart[v] != [-1,-1,-1] and traceStart[v] != [0,0,0]):
            res.append(traceStart[v][0])
            v = traceStart[v][0]
        res.reverse()
    else:
        res.append(intersect)
        while (traceStart[v][0] != -1):
            # unpacked the path after contracting from v to traceStart[v][0]
            res = res + self.unpack(traceStart[v][0], v, mode="adj")
            v = traceStart[v][0]
        res.reverse()
        v = intersect
        while (traceStop[v] != [-1,-1,-1]):
            res = res + self.unpack(v, traceStop[v][0], mode="rev")
            v = traceStop[v][0]
    if (mode == 'display'):
        print(res)
        print(f"Time: {dStart[intersect] + dStop[intersect]}")
        print(f"Query time: {times.time() - cur:.20f}")
    else:
        return res, dStart[intersect] + dStop[intersect]
```

Test this query with the same test case of section 3 (starting stop ID: 3550, ending stop ID: 3683), we got this output:

```
Intersect at node 1397
[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: 122.66109437702956
Query time: 0.00014281272888183594
```

As we can see, Contraction Hierarchy performs an excellent query time in a long searching path, compared with other search algorithms.

4.4. Benefits and Drawbacks

Contraction Hierarchy is perfect for applications that require real-time replies, like navigation systems, because it reduces the search space and provides incredibly quick query speeds for

pathfinding in big, static graphs. It is especially useful for stable networks with a stable structure because of its scalability and efficiency in managing big networks, like citywide transportation systems. However, these advantages do have some drawbacks such as high preprocessing costs and higher memory consumption as a result of the introduction of shortcuts, which can be resource- and time-intensive. Contraction Hierarchy has additional disadvantages due to the difficulty of creating and maintaining it, which calls for specific knowledge and cautious data structure maintenance. Additionally, because repeated preprocessing is required to counterbalance the benefit of rapid searches, its performance decreases in highly dynamic contexts where frequent updates are required. To properly use Contraction Hierarchy in practical applications, it is necessary to weigh its benefits and drawbacks.

5. Conclusion

Utilizing advanced algorithms like A* Search, Path Caching, and Contraction Hierarchy can significantly enhance the efficiency and performance of solving the shortest path problem in complex graphs, such as those representing urban transportation networks.

The A* Search algorithm, with its heuristic-driven approach, effectively narrows down the search space, leading to faster pathfinding by focusing on the most promising routes. Path Caching further optimizes performance by storing and reusing previously computed paths, reducing redundant calculations and providing rapid responses to repetitive queries. Contraction Hierarchy, through its preprocessing steps, restructures the graph to allow swift traversal by minimizing the number of nodes and edges that need to be considered during queries.

Each of these techniques offers distinct advantages: A* Search is particularly effective for real-time pathfinding with its balance of accuracy and speed; Path Caching excels in environments with high query repetition, significantly cutting down on computational load; and Contraction Hierarchy provides unparalleled query performance after an initial preprocessing phase, making it ideal for large-scale, static graphs. When used in conjunction, these algorithms can address the limitations of traditional methods like Dijkstra's algorithm, providing robust and scalable solutions for modern pathfinding challenges. By leveraging the strengths of A* Search, Path Caching, and Contraction Hierarchy, systems can achieve a high level of efficiency, accuracy, and responsiveness, essential for applications ranging from GPS navigation to network routing.

6. Reference

These are the website that I have looked for information about these algorithms, including:

1. A* Search Algorithm - GeeksforGeeks
2. Contraction Hierarchies Guide (jlazarsfeld.github.io)
3. Euclidean distance - Wikipedia
4. A* Algorithm in Artificial Intelligence You Must Know in 2024 | Simplilearn
5. Core Components of CH | Contraction Hierarchies Guide (jlazarsfeld.github.io)
6. Preprocessing Speed-up Techiques is Hard (kit.edu)
7. **Contraction hierarchies: Faster and simpler hierarchical routing in road networks**

This is my GitHub repository link:

8. GitHub Source Code for CS163 Solo Project (github.com)