OPTIMIZED HIERARCHY BASED SHORTEST PATH ALGORITHM FOR ROAD NETWORK GRAPHS

Jagreet Das Gupta

Institute of Engineering & Management, Salt Lake Electronics Complex,
Kolkata-700091. India.
Email: jagreetdg@gmail.com

Abstract:

Efficiently determining Shortest Paths on Road Maps has been a heavily engineered function for many routing apps like Google Maps & Yandex Maps for years now. Dutch computer scientist, *Edsger W. Dijkstra* set the stepping stone by formulating the now famous, Dijkstra's algorithm for shortest paths. Later it was improved upon by newer algorithms like A*. We introduce a way to implement modern algorithms such as Contraction Hierarchy, Highway Hierarchy and PHAST Algorithm to find optimal shortest paths in real life scenario road maps. It bases heavily on constructing a virtual "highway" which the shortest path should pass through theoretically as they are heavily traversed in real life generally. It considers parameters such as Road Distance between nodes(Edge Weight), Importance Factor during Contraction Hierarchy Phase, Cartesian Distance From Target, External Real Time Factors like Weather and Traffic. We determine a Heuristic Function using the above parameters and then use that to run a Bi-Directional PHAST based A* from both the source and the target node and then determine the shortest path once a common highway exists in both directional search's settled vector or a fallback stage is reached.

Keywords: *graphs, shortest-paths, goal-directed-search, road-networks, dijkstra,a*-algorithm , contraction-hierarchy, highway-hierarchy, phast-algorithm*

INTRODUCTION:

The Algorithm can be broken down into 3 Phases :

1) Pre-processing or Preparation Phase

2) Highway-Orienting Phase

3) Fallback Phase

Heuristic Function Parameters :

1) Road Distance or Edge Cost (c)

2) Importance Function or CH Parameter (i)

3) Cartesian Distance from Target (d)

4) Real-Time Weather (w)

5) Average Traffic (t)

Each parameter is assigned a significant index and coefficient based on its affect on the choice of the shortest path. Each factor would be discussed in detail in the following sections.

Preparation Phase is for gathering statistical intel like weather forecast, prior traffic on the edges for a stipulated number of days and running the Contraction Hierarchy Algorithm on the graph to obtain an augmented graph for the future phases. After that we apply a PHAST inspired pre-processing which is discussed later. After this part completes we would have successfully segregated the "highways" from the "roads" and memoized the relative highway path distances for efficient lookup later.In the Highway-Orienting Phase, we apply A* with custom heuristics also using the memoized highway distances calculated during PHAST to yield significantly low search times. We also introduce a threshold limit of traffic on these "highways" such that if the traffic on them crosses the limit, we disable highway segregation and transition over to Fallback Stage until the network traffic stabilizes.

In the Falback Stage, we completely ignore the PHAST Data and run a pure Bi-Directional A* from the source and target node using the custom heuristics function until the traffic in the network stabilizes after which again it transitions over to Highway-Orienting Phase.Fallback stage of this algorithm should rarely be run if the algorithm doesn't collapse or bottleneck under real traffic. It has not been bench-marked yet, so it's position among the stalwart algorithms is undetermined for now.

CONSTRAINTS AND IDEAL APPLICATION SCENARIO:

Before diving deep into the working of the algorithm, it is necessary to state the constraints based on which the algorithm will work and the ideal scenario its application is suited to.

Similar to Dijkstra's algorithm, this approach does not work on graphs with negative edge weights or self-cycles. Also ideally, this approach is more tailored for a large surface area road-map usually comprising a state or two. Enlarging or Reducing the working area of the graph may produce unsatisfactory results.

RELATED WORK:

The heart of this algorithm is Dijkstra's Algorithm[5] and its derivative A* Algorithm[6]. Highway-Oriented Phase takes heavy inspiration from the works of Sanders and Schultes on Highway Hierarchies[2] & Transit Node Routing[3] and Preparation Phase relies heavily on Contraction Hierarchy[4] by Geisberger, Sanders, Schultes and Delling and takes inspiration from the PHAST Algorithm[6] by Delling, Goldberg, Nowatzyk and Werneck.

DIJKSTRA'S ALGORITHM:

The origin of all shortest path algorithms, Dijkstra's algorithm uses a greedy approach to quickly select the nodes closest to the source and relax the edges greedily maintaining a priority-queue to store the shortest distances to each node explored till the iteration.

Pseudo-code :

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0
    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

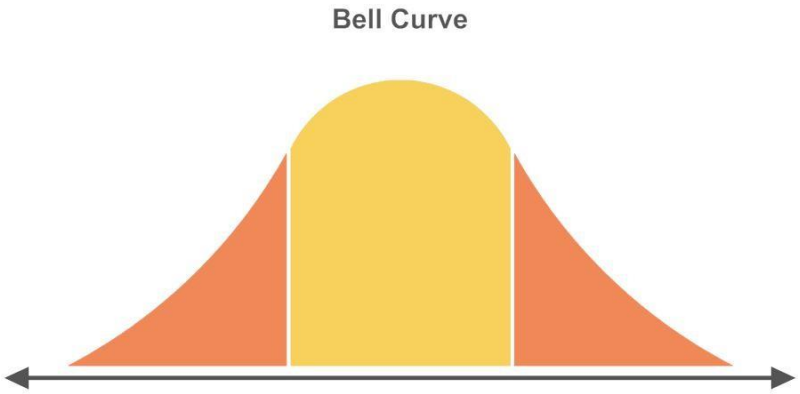
A* ALGORITHM :

A* Algorithm is an improvement upon Dijkstra's Algorithm as it takes into consideration a heuristic value which indicates how far a given node is from the target node. The structure is similar to Dijkstra's Algorithm:

```
function A*(start, goal)
  open_list = set containing start
  closed_list = empty set
  start.g = 0
  start.f = start.g + heuristic(start, goal)
  while open_list is not empty
    current = open_list element with lowest f cost
    if current = goal
      return construct_path(goal) // path found
    remove current from open_list
    add current to closed_list
    for each neighbor in neighbors(current)
      if neighbor not in closed_list
        neighbor.f = neighbor.g + heuristic(neighbor, goal)
        if neighbor is not in open_list
          add neighbor to open_list
        else
          openneighbor = neighbor in open_list
          if neighbor.g < openneighbor.g
            openneighbor.g = neighbor.g
            openneighbor.parent = neighbor.parent
  return false // no path exists
function neighbors(node)
  neighbors = set of valid neighbors to node // check for obstacles here
  for each neighbor in neighbors
    if neighbor is diagonal
      neighbor.g = node.g + diagonal_cost // eg. 1.414 (pythagoras)
    else
      neighbor.g = node.g + normal_cost // eg. 1
    neighbor.parent = node
  return neighbors
function construct_path(node)
  path = set containing node
  while node.parent exists
    node = node.parent
    add node to path
  return path
```

HIGHWAY HIERARCHY :

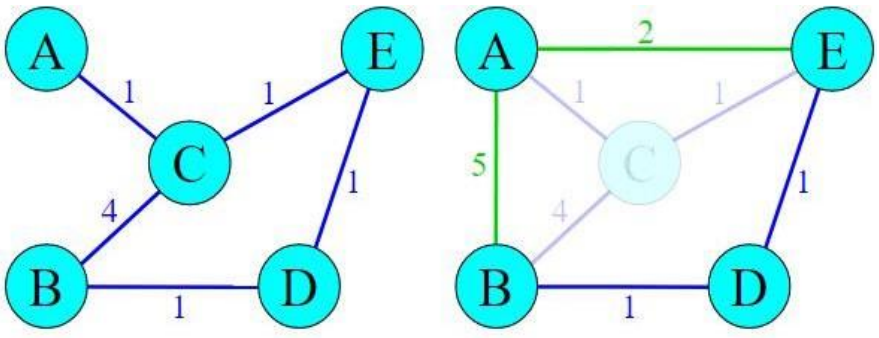
The idea of Highway Hierarchy is that most long-distance trips go through highways or technically speaking, follows a bell-curve similar to this:



Where the Y axis represents the "importance" of the road traversed and the X axis represents the part of journey covered. On inspecting the curve we realize, people tend to follow a symmetric travel pattern on long-distance trips usually opting for a more "highway" style road for the middle bulk of their journey. This gives us an intuition to implement the same as a shortest path algorithm when we are covering long journeys. Thus the algorithm follows the idea of first merging into a highway, then into a bigger highway, then exit via a smaller highway, then exit to a street. Less important roads merge into more important roads during the first part of the journey and vice-versa in the later part.

CONTRACTION HIERARCHY :

The basic idea of Contraction Hierarchy is to contract or reduce the graph into an augmented graph with lesser nodes/edges/easier to process but also preserving the relative distance between any two individual nodes. This version of CH differs slightly from the one given by Geisberger, Sanders, Schultes and Delling, to suit our constraints better. Contracting a node means removing an unimportant node from the graph and adding shortcuts between all pairs of nodes(u,v) such that the path between (u,v) that passed through the given unimportant node is represented with a single edge in the augmented graph rather than two edges and a node previously.



We choose the order in which the node is contracted such that :

1) The no of added shortcuts is minimized.

2) The important nodes(to be defined in the following section) is spread across the graph

3) Minimize the no of edges in the shortest paths in the augmented graph

In order to exhaustively choose this order, we first define some importance factor associated with each node which depends on the following factors :

1) The overall edge difference(Σ(added shortcuts) - Σ(in-degree of contracted nodes) - Σ(out-degree of contracted nodes)) is minimized in the augmented graph

2) No of Contracted Neighbors of any contracted node is minimized

3) Upper Bound on the Maximum Node Level in any shortest path passing through this node between any pair of vertices(u,v) is minimized

4) Nodes which have less average No of Traversals through it in Real Life are contracted first

We also define a probability function of a given node as to how likely a query may be made such that the given node is the source or target node. We will use this parameter later in the contraction process.

Once we establish a definite importance parameter, the contraction runs as follows :

1) Keep all nodes whose probability function is less than 0.3 in a min priority queue by decreasing importance

2) Extract least important node on each iteration

3) Recompute its importance as it may change in the newly augmented graph

4) If it is still minimal in the priority-queue, contract(remove) it from the augmented graph and add shortcut edges as necessary, else insert it back into priority-queue

5) Repeat from Step 1 until Priority-Queue is empty

Proof that atleast one node will be contracted every |V| (No of Vertices) Iterations :

1) If we don’t contract a node, we update its importance

2) After |V| attempts, all nodes will have updated importance

3) The node with the minimum updated importance will be checked after that but its importance will turn out the same because it has already been recomputed and hence it gets contracted.

4) In Real Life, it has been seen the average no of iterations to contract a node is

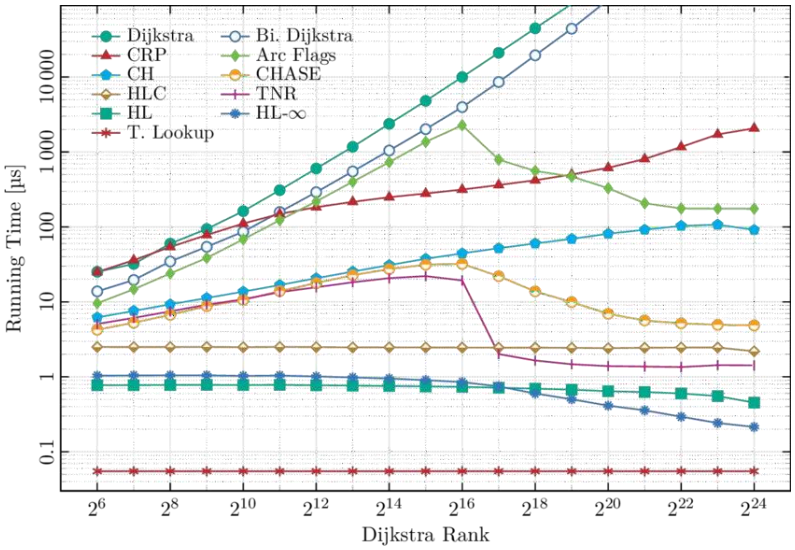
maximum one or two

Comparision between Dijkstra and CH :

On a graph of Europe with 18 million nodes, random pairs of vertices were chosen, Dijkstra had an avg run-time of 4.365 seconds while CH had an avg run-time of 0.18 milliseconds.

Almost 25000 times faster !

Here is a more exhaustive statistical comparison of running times between popular shortest path algorithms :



PHAST ALGORITHM :

PHAST Algorithm or Table Lookup is a variation of Contraction Hierarchy for one to many queries. It is similar to the Dynamic Programming technique as it involves trading off running time for heavy memory usage. In the original PHAST, the pre-processing search begins with a forward upward search from from the source node and the backward search from the target node determines distance to all nodes. It does so by relaxing all edges(u,v) in the backward graph such that rank(u)>rank(v). We extend this idea in the forward search as well thereby obtaining a comparitively linear graph traversal. This idea directly integrates with the highway hierarchy approach as we are travelling from less important to more important nodes in the forward search and more important to less important nodes in the backward search. We memoize these results for efficient lookup during later stages.

Here is a comparison of running times of various versions of PHAST :

	PREPRO		CUSTOM		SEQ. QUERIES [MS]		PAR. QUERIES [MS]	
	time	space	time	space	limit [min]		limit [min]	
algorithm	[s]	[MiB]	[s]	[MiB]	$x = 100$	$x = 500$	$x = 100$	$x = 500$
RangeDijkstra	–	645	–	–	59.00	966.79	–	–
isoCRP (1-phase)	28.39	762	**1.50**	138	26.20	114.24	–	–
isoCRP	28.39	762	**1.50**	138	17.58	75.22	3.02	9.08
isoGRASP	28.39	762	2.38	1 093	10.73	43.23	2.43	6.20
ES+PHAST (cd)	36.93	767	–	–	**6.50**	35.17	**1.57**	8.33
ES+PHAST (cp)	1 336.70	766	–	–	12.74	29.30	4.09	7.50
ES+PHAST (do)	1 265.59	880	–	–	22.21	46.59	4.33	8.10
VS+PHAST	868.52	1 530	–	–	10.24	**27.77**	2.13	**4.09**

PREPARATION PHASE :

First we make an exhaustive list of all the parameters involved and then form an efficient equation for the heuristic function of A*.

1) Weather (w)

2) Traffic (t)

3) Edge Cost / Road Distance (c)

4) Cartesian Distance from Target Node (d)

5) Importance (i)

All Parameters lie in the tentative range [0-100]. Some parameters may exceed or fall below this range in extreme cases.

Weather :

We divide the entire graph into a four fold grid-square system, determine the average weather in each square grid and denote the weather parameter in all nodes in that grid by the same. Weather function of an edge is the mean of the two nodes it connects. It has an index of 1 and a co-efficient 2 in the heuristic function. A value of 0 indicates minimal weather involvement while a value of 100 indicates extreme weather interference.

Traffic :

It is the mean traffic across the nodes in the previous few days. Traffic function of an edge is the mean of the two nodes it connects. It has an index of 3 and co-efficient 1. A value of 0 indicates no traffic while 100 indicates a Jammed road.

Edge Cost :

It is defined as 100 multiplied by the Path Distance between the two connected nodes divided upon by the Direct Cartesian Distance between the source and the target node. Mathematically, it is given by :

$$\frac{(100*P(u,v))}{D(s,t)}$$

Where P(u,v) is the edge distance between the two concerned nodes connected by the edge, s is the source node, t is the target node and D() is the Direct Cartesian Distance, which is defined as the coordinate distance between two nodes on a Cartesian Plane. It has an index of 2 and co-efficient 1. The lower its value, the shorter the path is.

Cartesian Distance from Target Node :

Cartesian Distance from Target Node for a given node is defined as 100 multiplied by the Direct Cartesian Distance between the given node and the target node divided by the Direct Cartesian Distance between the source node and the target node. Mathematically it is given by :

$$\frac{(100*D(u,t))}{D(s,t)}$$

Where D() is the Direct Cartesian Distance function, u the given node, s the source node and t the target node. For an edge, it is defined as the mean between the values of its connecting nodes. It has an index of 4 and co-efficient 1. The lower its value, the closer the node/edge is to the target node.

Importance:

The Importance parameter depends on several factors and is given by the equation:

$$\frac{\big((in + out - ea) - (cn + l) + rt\big) * 100}{N}$$

in : The Summation of in-degrees of the contracted nodes in CH

out : The Summation of out-degrees of the contracted nodes in CH

ea : Edges Added during the shortcut creation part of CH

cn : Number of Contracted Neighbors of the given Node

rt : Average probability of this node being traversed in a shortest path

l : Upper Bound on the node level in any shortest path passing through it

It has an index of e and co-efficient 3. A value of 0 indicates least importance and 100 indicates maximum importance.

Heuristic Function :

Finally our Heuristic Function turns out to be:

$$h = 3i^e + c^2 + d^4 + 2w + t^3$$

We will use this heuristic function to direct our A* search in later stages of the algorithm.

The next part of the Preparation Phase is the Contraction and PHAST Phase. Contraction Heuristic Algorithm contracts the graph into an augmented graph as explained previously using another parameter called the probability function and then PHAST is run on the augmented graph to memoize the highway path for later lookup. This terminates the preparation phase.

HIGHWAY-ORIENTING PHASE:

This is the core phase of the algorithm and generally yields the fastest query times. Initially it sets a live traffic variable for each highway edge and a suitable threshold traffic is selected.

If during any point of time during this phase, the live traffic variable on any highway edge exceeds the threshold, the algorithm automatically transitions over to Fallback Phase. The A* Directed search during this phase is as follows :

```
function V*(source,target)
    a_open = empty set
    b_open = empty set
    a_closed = empty set
    b_closed = empty set
    for each highway in phast(source)
        add highway and heuristic to a_open
    for each highway in phast(target)
        add highway and heuristic to b_open
    while a_open is not empty or b_open is not empty
        if a_open is not empty
            current = a_open element with lowest cost
            remove current from a_open
            add current to a_closed
            for each child in neighbors(current)
                if child not in a_closed
                    child.cost =
min(child.cost,current.cost+edge(current,child)+heuristic(child,target))
                if child not in a_open
                    add child to a_open
                    update child.parent to current
                else
                    if child.cost < child.cost in a_open
                        update child.cost in a_open
                        update child.parent in a_open to current
        if b_open is not empty
            current = b_open element with lowest cost
            remove current from b_open
            add current to b_closed
            for each child in neighbors(current)
                if child not in b_closed
                    child.cost =
min(child.cost,current.cost+edge(current,child)+heuristic(child,source))
                if child not in b_open
                    add child to b_open
                    update child.parent to current
                else
                    if child.cost < child.cost in b_open
                        update child.cost in b_open
                        update child.parent in b_open to current

        if common_element exists in a_closed and b_closed
            return trace_path(commmon_element,source,target)
    return no_path_found
```

FALLBACK PHASE :

This phase only runs when the live traffic in any one of the highways crosses the threshold and hence the Highway Hierarchy and PHAST Optimization has to be abandoned until the network traffic stabilizes. Once Fallback Phase is called, it follows this algorithm :

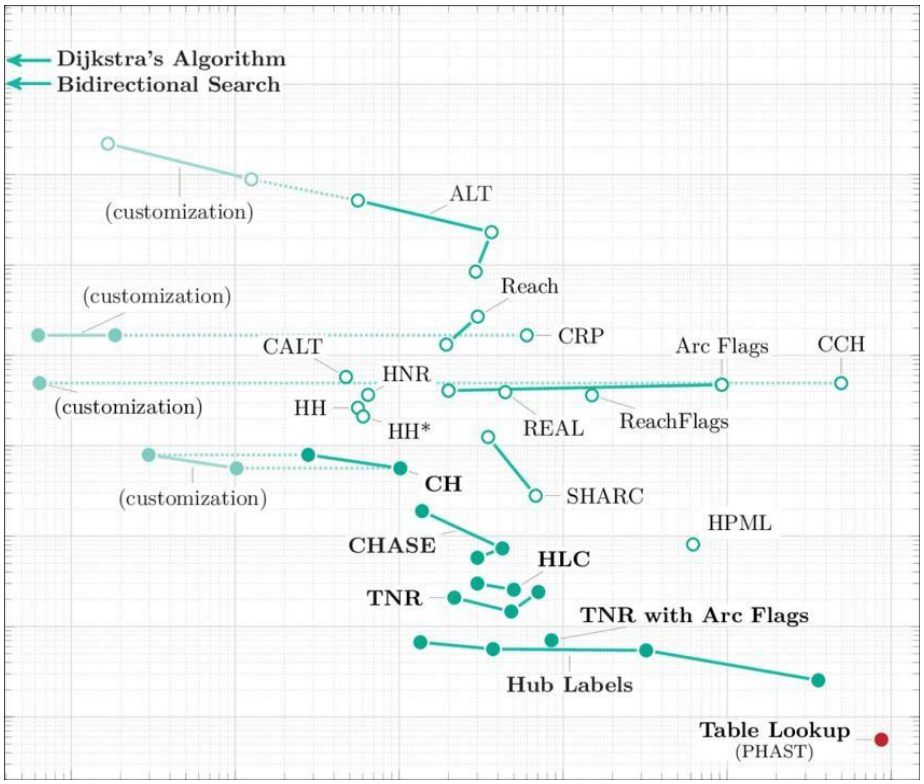
```
function fallback*(source,target)
    a_open = empty set
    b_open = empty set
    a_closed = empty set
    b_closed = empty set
    add source and heuristic to a_open
    add target and heuristic to b_open
    while a_open is not empty or b_open is not empty
        if a_open is not empty
            current = a_open element with lowest cost
            remove current from a_open
            add current to a_closed
            for each child in neighbors(current)
                if child not in a_closed
                    child.cost =
min(child.cost,current.cost+edge(current,child)+heuristic(child,target))
                if child not in a_open
                    add child to a_open
                    update child.parent to current
                else
                    if child.cost < child.cost in a_open
                        update child.cost in a_open
                        update child.parent in a_open to current
        if b_open is not empty
            current = b_open element with lowest cost
            remove current from b_open
            add current to b_closed
            for each child in neighbors(current)
                if child not in b_closed
                    child.cost =
min(child.cost,current.cost+edge(current,child)+heuristic(child,source))
                if child not in b_open
                    add child to b_open
                    update child.parent to current
                else
                    if child.cost < child.cost in b_open
                        update child.cost in b_open
                        update child.parent in b_open to current

        if common_element exists in a_closed and b_closed
            return trace_path(commmon_element,source,target)
    return no_path_found
```

The running time of this phase is significantly lower than Highway-Oriented Phase as it doesn't use the PHAST Memoizations but in turn provides a more traffic friendly routing to prevent deadlocks in highways.

CONCLUSION :

While traditional algorithms like Dijkstra and A* certainly do the job quite efficiently for smaller graphs, they become un-scalable when applied on larger graphs. Modern algorithms like Contraction Hierarchies, CALT, ALT, CRP etc. tend to achieve acceptable numbers, memoization algorithms like PHAST although outperform them. Here is a graphical comparison between the most popular shortest-path algorithms.



But we may not always have the necessary memory power needed for efficient PHAST Execution, as it tends to take up rather ridiculous chunks of memory as illustrated by this memory-time usage table of shortest-path common algorithms :

		DATA STRUCTURES		QUERIES	
algorithm	impl. source	space [GiB]	time [h:m]	scanned vertices	time [µs]
Dijkstra	[75]	0.4	–	9 326 696	2 195 080
Bidir. Dijkstra	[75]	0.4	–	4 914 804	1 205 660
CRP	[77]	0.9	1:00	2 766	1 650
Arc Flags	[75]	0.6	0:20	2 646	408
CH	[77]	0.4	0:05	280	110
CHASE	[75]	0.6	0:30	28	5.76
HLC	[82]	1.8	0:50	–	2.55
TNR	[15]	2.5	0:22	–	2.09
TNR+AF	[40]	5.4	1:24	–	0.70
HL	[82]	18.8	0:37	–	0.56
HL-∞	[5]	17.7	60:00	–	0.25
table lookup	[75]	1 208 358.7	145:30	–	0.06

The proposed approach in this paper presents a hybrid algorithm, which combines the pros of the common algorithms and provides an efficient, scalable solution to the shortest-path problems in road-network graphs.

REFERENCES :

1.PHAST : Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, Renato F. Werneck, Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA https://www.microsoft.com/en-us/research/wp-content/uploads/2010/09/phastTR.pdf
2.Engineering Highway Hierarchies : Peter Sanders and Dominik Schultes, Universit¨at Karlsruhe (TH), 76128 Karlsruhe, Germany http://algo2.iti.kit.edu/documents/routeplanning/esa06HwyHierarchies.pdf
3. Transit Node Routing Reconsidered : Julian Arz, Dennis Luxen, Peter Sanders, Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany http://algo2.iti.kit.edu/documents/chbasedtnr_TR.pdf
4.Contraction Hierarchies : Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling, Universit¨at Karlsruhe (TH), 76128 Karlsruhe, Germany http://algo2.iti.kit.edu/schultes/hwy/contract.pdf
5.Dijkstra : Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1(1):269–271, 1959