Highway Dimension, Shortest Paths, and Provably Efficient Algorithms

Mathis Först

November 2, 2018

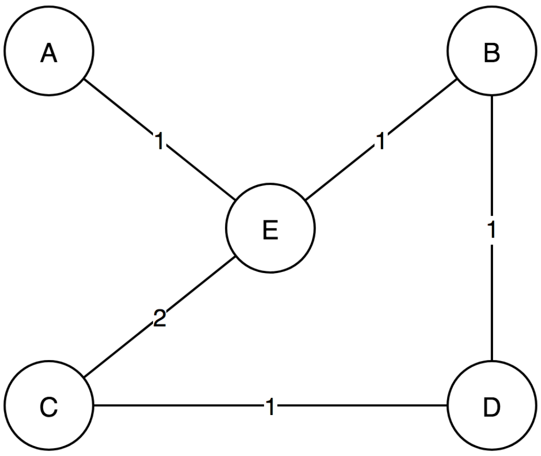
1 Introduction

Finding a shortest path between two nodes in a graph $G(V, E)$ is a common problem. The common way to solve this is Dijkstras Algorithm which has a running time of $O(|V| \log |V| + |E|)$. If we think of navigation in road networks this is too slow since the road networks of Europe or North America each consist of tens of millions of intersections (which are nodes in the corresponding graph).
To make navigation fast even on slow mobile devices we can exploit the fact that road networks do not change very often. Therefore we can spend some time (in the magnitude of hours) to perform preprocessing operations on the graph. The additional information will help to perform path queries much faster.
We will see the method of contraction hierarchies to speed up path queries. Then we will introduce the property 'highway dimension' of a graph. Finally we will use these highway dimensions to prove the performance of a slightly adapted variant of contraction hierarchies.

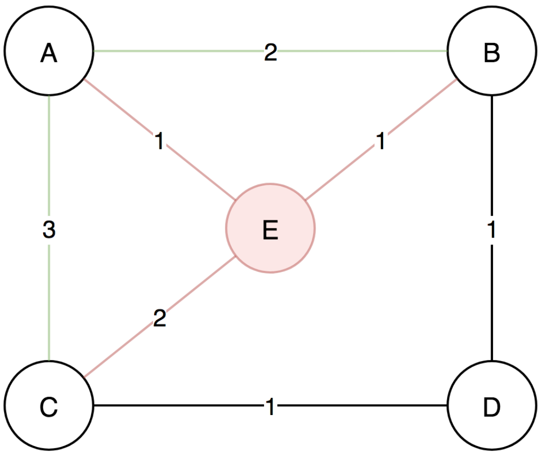
2 Basic Definitions

We use the following notations:
For a graph $G = (V, E)$

- $n = |V|$
- $m = |E|$
- $w(e)$ the weight of an edge $e \in E$
- $P(u, v)$ is a shortest path between the nodes u and v
- $D = max_{u,v \in V}(P(u, v))$ is the diameter of the graph, the longest shortest path
- $B_{u,r}$ is a ball with radius r around the node u, the set of nodes $S = \{v \in V \mid |P(u, v)| \leq r\}$
- Δ is the maximum degree of a node of a graph



(a) Original Graph G



(b) G after shortcutting node E

Figure 1: Shortcuts

2.1 Shortcuts

The idea behind a shortcut is to remove a node v from the graph without destroying shortest paths going through v.

Definition 1 (Shortcut)**.** *The shortcut operation deletes a vertex v from G. For every pair of neighbors of v $\{u, w\}$ where $(u, v) \cdot (v, w)$ is the shortest path between u and w, the edge (u, w) with the weight $l(u, w) = l(u, v) + l(v, w)$ is added to the graph.*

We have a short look at the example in Figure 1. We want to shortcut node E. Since the shortest paths $P(A, C)$ and $P(A, B)$ go through E we add the corresponding shortcut edges. The shortest path $P(B, C)$ does not go through E. Therefore no shortcut edge is necessary.

3 Contraction Hirarchies (CH)

The basic idea behind contraction hierarchies is pretty simple. We will use a shortest path algorithm ehich is very similar to Dijkstras algorithm. To speed up the query we do not visit all neighbours of a node when we scan the node but only specific ones. Of course we can not just ignore random nodes since this could end in a wrong result. Therefore use the following preprocessing and pathfinding strategies.

3.1 CH Preprocessing

The basic idea of the preprocessing is building a hierarchy over all nodes and then shortcut them from lower to higher hierarchy levels until only one node is left. All the shortcut edges E^+ which were generated by this process are then added to the original graph so that we perform the path query finally on $G = (V, E \cup E^+)$.
For the first attempt we use the following simple hierarchy on the nodes: Every node v gets a random unique rank $rank(v)$. In the hierarchy a node v has a higher level than a node u iff $rank(v) > rank(u)$.

3.2 CH Pathfinding

To find a shortest path in the augmented graph we use a bidirectional variant of Dijkstras Algorithm. The basic idea is to look for shortest paths from the start node s and the end node t in parallel until the two searches meet somewhere in the middle.
To do this we have to maintain a separate priority queue for both searches to know which node should be scanned next. To decide which node is scanned next, we take the element from queue where the top element has the shorter path length. In that way our search stays balanced. This means the circles of scanned nodes around s and t keep having approximately the same radius while they grow during the search.
The trick in CH is the following: When scanning a vertex v we only consider neighbors w with $rank(w) > rank(v)$. In that way we do not have to scan all vertices and can save computing time. Since we have chosen an arbitrary hierarchy the performance may vary a lot.
The modified Dijkstra-search terminates when both queues are empty. But at this point we do not have our final result.
We define $d_s(u)$ as the shortest path from the startpoint s to u found by the previous search. Similar $d_t(u)$ is the distance from the endpoint t to u.
After the search completed we can or can not have $d_s(u)$ or $d_t(u)$ for each node u.
The shortest path $P(s,t)$ goes through the node u that minimizes the path length $d_s(u) + d_t(u)$ and is given by the concatenation $P(s,u) \cdot P(u,t)$.

Let's do this in the example of Figure 2:
We use the following preprocessed graph. The node hierarchy is given by the alphabetical ordering (e.g $rank(A) < rank(B)$). All edges added during preprocessing are drawn in green. The start and end nodes are marked in yellow. Note that the two green edges on the right were added during the shortcutting of node A and the edge (D,E) was added during the shortcutting of node C.
We want to find a shortest path from B to D. So we start the search at both nodes. We

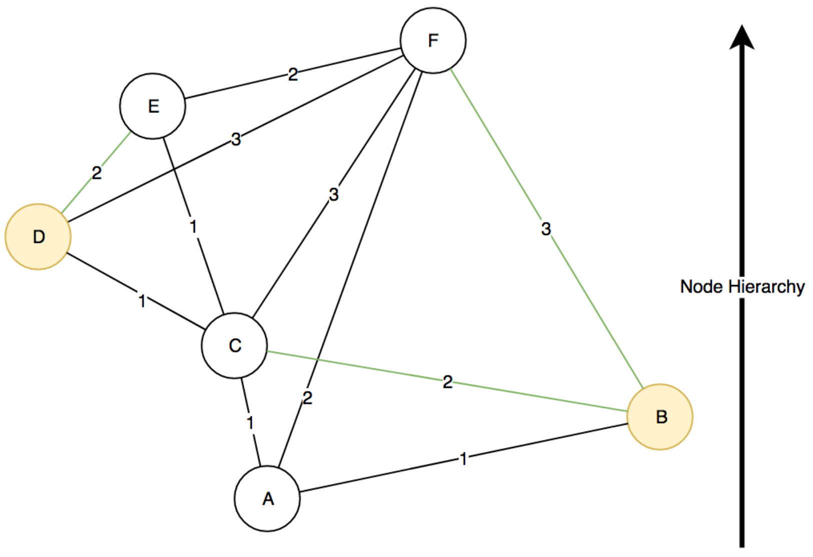


Figure 2: Preprocessed Graph

get the following distances:

$$\begin{aligned} d_D(D) &= 0 \\ d_B(B) &= 0 \\ d_D(E) &= 2 \\ d_B(C) &= 2 \\ d_D(F) &= 3 \\ d_B(D) &= 3 \\ d_B(E) &= 3 \\ d_B(F) &= 3 \end{aligned} \tag{1}$$

The vertices which were reached from both searches are D, E, F. The minimum distance is reached using node D and is 3. Note that node A is not scanned during the search.

3.3 Why do CH work?

Now we have to find out why this algorithm calculates correct results:
We have a closer look at the point in our example where the algorithm scans node B. In the original graph the shortest path from B to C, $P(B, C)$, goes through node A. We are not allowed to use node A since $rank(A) < rank(B)$. But in the preprocessing we added the edge (B, C) when we shortcutted node A with the right weight such $P(B, C)$ still has the same length.
That is why the algorithm produces the correct results. If a shortest path goes through a node of a lower hierarchy level, a shortcut edge was added during preprocessing.
The performance is dependent on the hierarchy of the nodes. Choosing it arbitrarely is obviously not the best thing we can do since we can always construct an example where still all nodes of the graph are scanned during the path query.
In practice different metrics are used to get well performing node hierarchies. E.g. we could prefer nodes for shortcutting where we can add the most shortcut edges. Edged difference means that we prefer to contract the nodes where we can add the most shortcut edges (called edge difference). Another good idea is to make sure that a node v which is shortcutted directly after node u is geographically far away from u. This keeps the hierarchy levels balanced over the whole map [1].

3.4 Measured Performance of CH

If you choose the right metrics you end up with a preprocessed graph of the European road network with 18 million nodes where an average path search visits less than 400 nodes!
That is pretty impressing, but these results were generated by running many random queries on one special graph (the European roads).
Since this is not a proof of the performance of the algorithm, we will now see some properties of a graph which enable us to prove the performance of a slightly modified CH-algorithm.

4 Highway Dimensions

We know that nearly all sufficiently long routes in road networks include highways. Since highways have few intersections (compared to other road types) they can be represented by few nodes in a graph representing the road network. Therefore we can say that most shortest paths in a road network pass through a small set of nodes.
The aim of highway dimensions is to quantify this property: We define the highway dimension of a graph to be the smallest number h such that for every distance r and every ball $B_{u,4r}$ with radius $4r$ around any node u the following property holds: All shortest paths connecting two nodes in this ball which have a length of at least r can be covered by a set of nodes S which contains at most h elements.

Definition 2 (Highway Dimension). *Given a graph $G = (V, E)$, its highway dimension is the smallest number h such that*

$$\forall r \in \mathbb{R}^+, \forall u \in V, \exists S \subseteq B_{u,4r}, |S| \leq h, s.t.$$

$$\forall v, w \in B_{u,4r}$$

$$if |P(v,w)| > r \text{ and } P(v,w) \subseteq B_{u,4r} \text{ then } P(v,w) \cap S \neq \emptyset$$

4.1 Shortest Path Covers (SPC)

The highway dimension of a graph is basically just a single number. We are now interested in sets of nodes with special properties.
We define a shortest path cover (SPC) as a set of nodes C which covers all shortest paths with length between r and $2r$. In addition we want an SPC to be sparse in the sense that the size of any intersection between C and any ball of radius $2r$ is at most k. SPCs have the two parameters r and k and we denote them therefore (r, k)-SPC.

Definition 3 (Shortest Path Cover). *A set C is an (r, k)-SPC of G iff $\forall u \in V, |C \cap B_{u,2r}| \leq k$ and $\forall$ shortest path $P : r < |P| \leq 2r, P \cap C \neq \emptyset$*

4.2 Highway Dimension implies SPC

Theorem 1. *If a graph G has highway dimension h then there exists for any distance r an (r, h)-SPC of G.*

Proof. Let S^* be the smallest set that covers all shortest paths P with $r < |P| \leq 2r$. We now prove by contradiction that S^* is an (r, h)-SPC: Assume there is a node u with $U = S^* \cap B_{u,2r}$ and $|U| > h$. By the definition of h there exists a set H with $|H| \leq h$ covering all shortest paths P' in $B_{u,4r}$ with $|P'| > r$ and therefore obviously all paths P with $r < |P| \leq 2r$ covered by U. We see that $(S^* - U) \cup H$ is smaller than S^* and still covers all shortest paths P with $r < |P| \leq 2r$. This contradicts the optimality of S^* and completes the proof. □

5 Modified CH

We now use the fact that given a graph with a fixed highway dimension h we can construct an (r, h)-SPC for any r to modify the CH-algorithm and prove the performance.

5.1 Preprocessing

We do not use an arbitrary hierarchy for the nodes anymore but build a special one by using a sequence of shortest path covers:
Let $S_0 = V$ and for $1 \leq i \leq \log D$ let S_i be an $(2^i, h)$-SPC. Let $L_i = S_i - \bigcup_{j=i+1}^{\log D} S_j$
In our hierarchy every node from L_i comes before L_{i+1}. The order within each L_i is arbitrary. We break the uniqueness of $rank(v)$ and set the rank of every node in the set L_i to i.
Now we shortcut the nodes in that order like before starting with the nodes from L_1.
For every node v we create the shortcut edges. Now we have to take care to destroy no shortest paths: Since we allowed nodes on the same hierarchy level, we have to add additional edges to ensure that a shortest path which goes through multiple points of the same hierarchy level can still be found.
Therefore when we shortcut a node $v \in L_i$ for every pair $u, w \in B_{v,2^{i+1}}$ where $v \in P(u, w)$ we add another edge (u, w) with length $|P(u, w)|$.
This gives us again the preprocessed graph $G^+ = (V, E \cup E^+)$.

5.1.1 Bounds of the number of added edges

The first important thing to prove the performance of queries on G is the upper bound for the number of edges from a node to higher and lower hierarchy levels.

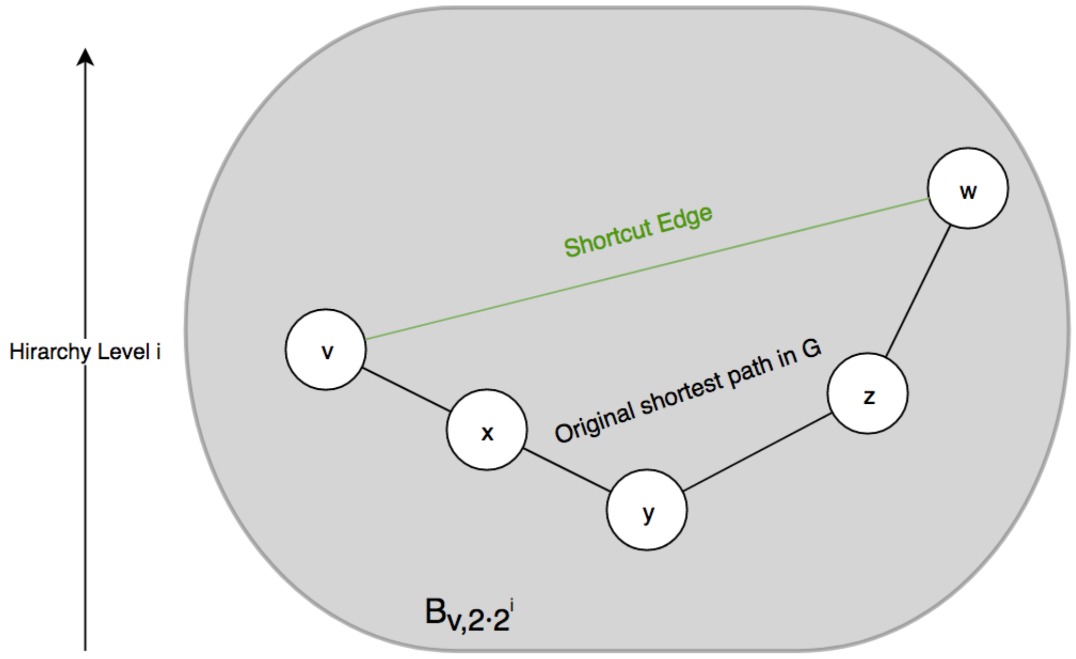


Figure 3: Bound on the number of edges

Theorem 2. *If $v \in L_i$, the number of edges (v, w) to a node in a fixed higher hierarchy level $L_j \ni w$ with $j \geq i$ is at most h.*

Proof. (v, w) is the shortcut of a shortest path P in the original graph. Since $(v, w) \in L_i$ all the internal vertices of the path have to been shortcut before v and w. Therefore these internal vertices have to be on lower (or the same) hierarchy levels L_x with $x \leq i \leq j$. Also w has to be in the ball $B_{v,2 \cdot 2^j}$. Otherwise $|P|$ would be greater than $2 \cdot 2^i$ which can not be true since P does not contain a vertex from a higher hierarchy level than j, but all shortest paths longer than $2 \cdot 2^j$ have to be covered by at least one vertex of L_y with $y > j$. Since $B_{v,2 \cdot 2^j}$ can contain at most h vertices from L_j (by the definition of an SPC) the proof is completed. □

We can prove the same for connections to lower hierarchy levels in a similar way. Therefore we can bound the degree of the vertices in the preprocessed graph G^+ to $\Delta + h \log D$ since we have $\log D$ hierarchy levels and a node can be incident to at most Δ edges from the original graph.
Also the number of edges added during the preprocessing can be bounded: $|E^+| \in O(nh \log D)$.

5.2 Reach

The last definition we need is reach.

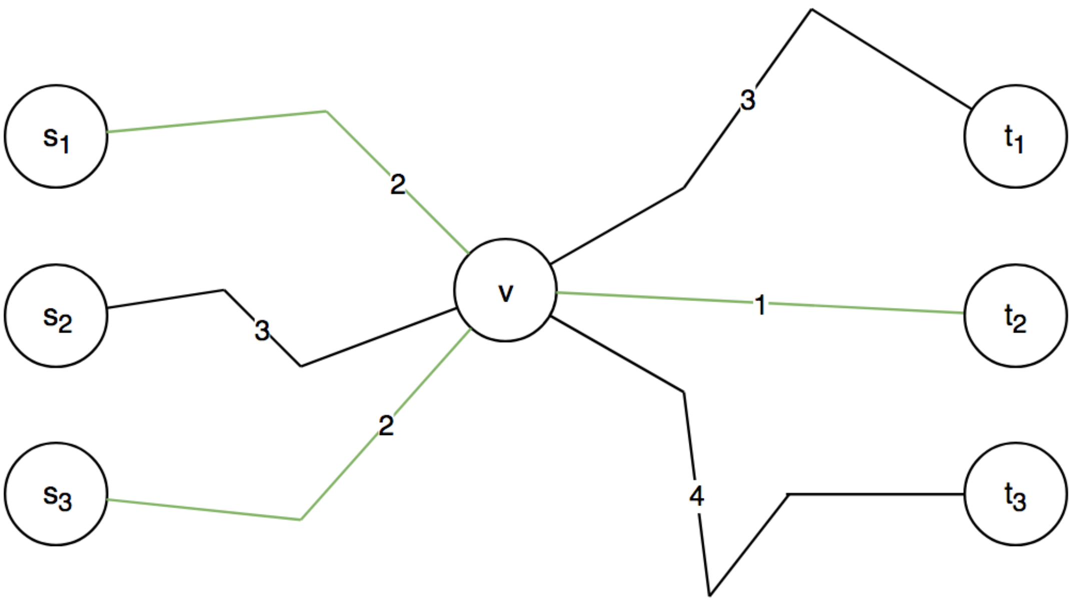


Figure 4: Reach of node v is 2

Definition 4 (Reach)**.** *The reach of a node v with respect to a shortest path P which is divided by v into P_1 and P_2 is $r_P(v) = \min(|P_1|, |P_2|)$. The reach of a node is $r(v) = \max_{P \in p(v)} r_P(v)$.*

In Figure 4 we see 3 shortest paths $(P(s_1, t_1), P(s_2, t_2), P(s_3, p_3))$ through the node v. The shorter half of each path is marked in green. The reach of the node v is the length of the longest green segement which is 2.

Theorem 3. *In the preprocessed Graph G^+ for any $v \in L_i$ holds $r(v) \leq 2 \cdot 2^i$.*

Proof. For this proof we refine our definition of a shortest path: If there are shortest paths with minimal length between two nodes u and v only the ones with the minimum number of contained nodes are considered as shortest paths.
We prove the theorem by contradiction: Assume that $r(v) > 2 \cdot 2^i$. By definition of $r(v)$ there has to be a shortest path $P(x, y)$ in G^+ with

1. P contains v
2. the subpath P_1 from x to v and the subpath P_2 from v to y are both longer than $2 \cdot 2^i$.

P_1 and P_2 must both contain vertices $u \in L_j$ with $j > i$ since they are shortest paths with length $> 2 \cdot 2^i$. Among these let u_1 and u_2 be the nodes closest to v on these paths P_1 and P_2 respectively. This implies that all vertices on P between u_1 and u_2 (including v) are shortcut before u_1 and u_2 since they are on a lower hierarchy level. Therefore we know that (u_1, u_2) is a shortcut edge in G^+. With this edge we can construct a new shortest between x and y which contains fewer nodes than $P(x, y)$ (at least v will not be in the new path). This contradicts that $P(x, y)$ is shortest path. $\square$

5.3 Query Processing Time

What is the processing time of a shortest path query from s to t?
For the forward search we look at the ball $B_{s,2 \cdot 2^i}$: The search will not scan any vertex $v \in L_i$ which is not inside this ball. This holds because $r(v) \leq 2 \cdot 2^i$ but $|P(s, v)| > 2 \cdot 2^i$. Therefore v will be scanned either by the backward search or not at all.
Since $1 \leq i \leq \log D$ and there are at most k vertices in $B_{s,2 \cdot 2^i}$, the forward search will scan at most $O(k \log D)$ vertices. A similar argument holds for the backward search therefore we scan in total $O(k \log D)$ vertices.
If we scan a vertex we have to look at all of its neighbours. Since the maximum degree (which is the number of neighbours) of every vertex is $\Delta + h \log D$ (see Theorem 2), the whole processing time is in $O((\Delta + h \log D)(h \log D))$.

6 Remarks

6.1 Polynomial preprocessing time

In the calculations we ignored the processing time for the SPCs which is exponential for (h, r)-SPCs. If we want polynomial processing time, we get only $(h \log n, r)$ SPCs, which increases the whole processing time a bit.

6.2 Data structure overhead

We also ignored the overhead for the data structure in the Dijkstra search. We could use a Fibonacci Heap. Since we scan $O(k \log D)$ vertices we get an additive extra term of $O(k \log D \log n)$.

7 Outlook

We used the notion of highway dimensions to prove the performance of a CH algorithm. The paper also shows proofs for the performance of other pathfinding algorithms.
The problem with highway dimensions is that it is nearly impossible to compute them for a big graph like a real world road networks since the computation is probably NP-hard. The paper tackles this problem by introducing a model of the formation of road networks and calculating its highway dimension. The model is build on top of the following assumptions:

1. Roads are built incremental and in a local manner. This means that road building is a decentralized and on-line process.
2. The geometric space in which roads exist has a low dimension.
3. Longer roads are typically faster than shorter ones. This means that highways have shorter traversal times per kilometer than local roads.

A road network which is built onto these has a provable constant highway dimension.
Of course this model is not perfect but it gives a first intuition why the real world road networks have a low highway dimension.

8 Conclusion

We saw how to compute a shortest path without scanning the whole graph using CH. E.g this makes it possible to perform path queries on big road networks on mobile devices. We also saw highway dimensions to actually prove the performance of CH. In general highway dimensions enable us to prove the performance of a whole group of path finding algorithms.

References

[1] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. *Contraction hierarchies: Faster and simpler hierarchical routing in road networks*, pages 319–333. WEA, 2008.