Highway Dimension, Shortest Paths, and Provably Efficient Algorithms

Ittai Abraham[*], Amos Fiat[†], Andrew V. Goldberg[‡], Renato F. Werneck[§]

Abstract

Computing driving directions has motivated many shortest path heuristics that answer queries on continental scale networks, with tens of millions of intersections, literally instantly, and with very low storage overhead. In this paper we complement the experimental evidence with the first rigorous proofs of efficiency for many of the heuristics suggested over the past decade. We introduce the notion of highway dimension and show how low highway dimension gives a unified explanation for several seemingly different algorithms.

1 Introduction

Gaius Octavius Thurinus (aka Augustus Caesar) was put in charge of Roman roads (*viae*) in 20 BC. Formally, all *viae* began at the Temple of Saturn in Rome. Milestones along the roads gave distances along the road and to the Forum in Rome. According to the Cosmographia Julius Honorius, Consuls Julius Caesar and Mark Anthony sent out four scholars to map the world: Nicodemus, Didymus, Thodotus, and Polycletes. This task took them 32 years, one month, and twenty days. It is not clear how the Romans computed shortest paths.

Although the raw data about geography and roads may be more readily available today, computing shortest paths is still not trivial. Dijkstra's algorithm [6] allows us to compute point-to-point shortest path queries on any road network in essentially linear time. Unfortunately, this is impractical for large road networks, such as those of North America or Europe, where one would like to answer queries while examining only a small fraction of the graph.

Motivated by computing driving directions, several heuristics have been proposed in the preprocessing/query framework. In a preprocessing stage, these heuristics compute some auxiliary data, such as additional edges (shortcuts) and labels or values associated with vertices or edges. The auxiliary data is then used to accelerate an arbitrary number of s–t shortest path queries, typically by pruning or directing Dijkstra's algorithm.

Heuristics within this framework are based on a wide variety of ideas, such as arc flags [17, 14, 3], A^* search with landmarks [9], highway hierarchies [19, 20], reach [13, 10, 11], transit nodes [1, 2], and contraction hierarchies [8]. In experiments using real-world data, queries answered with these heuristics are an amazing improvement over plain Dijkstra: visiting a few hundred vertices is enough to answer a random query on road networks with tens of millions of intersections. All methods are exact: they are guaranteed to find the actual shortest path, not an approximation. Moreover, preprocessing is practical (as fast as a few minutes for some algorithms) and produces auxiliary data structures that require only slightly more memory than the road network alone.

Unfortunately, these excellent practical results are purely experimental, with no provably good time guarantees. In fact, it is not hard to construct inputs for which these heuristics fail to achieve any meaningful speedup. No analysis of the heuristics on any non-trivial graph classes has been known. Furthermore, there was no theoretical understanding of which properties of road networks make the heuristics perform well; previous work in this direction has been experimental only [4].

The lack of theoretical understanding of the practical shortest path algorithms suggests the following natural questions, which are the subject of this paper. Can one prove sublinear query bounds for these heuristics on a non-trivial class of networks? For what graphs does the preprocessing/query framework lead to algorithms with provably good performance? Specifically, what properties of road networks imply provably good performance for the (*de facto* excellent) heuristics above? Finally, is there a plausible explanation as to why real road networks actually satisfy such conditions?

To address these questions, we define the notion of *highway dimension*. Intuitively, a graph has small highway dimension if for every $r > 0$, there is a sparse set of vertices S_r such that every shortest path of length greater than r includes a vertex from S_r. A set is sparse if every ball of radius $O(r)$ contains a small number of elements of S_r.

[*]Microsoft Research Silicon Valley, `ittaia@microsoft.com`

[†]Tel-Aviv University, `fiat@tau.ac.il`. Part of the work done while the author was visiting Microsoft Research.

[‡]Microsoft Research Silicon Valley, `goldberg@microsoft.com`

[§]Microsoft Research Silicon Valley, `renatow@microsoft.com`

We show that low highway dimension gives provable guarantees of efficiency for the following algorithms (sometimes with small modifications): *reach (RE)*, *contraction hierarchies (CH)*, *highway hierarchies (HH)*, *transit node (TN)*, and *SHARC [3]* (which is based on arc flags). More precisely, given a connected, simple graph with n vertices, m edges, highway dimension h, maximum degree Δ, and diameter D, we can prove the following:

- Preprocessing takes time polynomial in m (and n), h and $\log D$.
- The auxiliary data it produces has size linear in m and polynomial in $\log n$, h and $\log D$.
- The s–t query returns an implicit representation of the shortest s–t path (including the path length) in time polynomial in Δ, h, $\log n$, and $\log D$. (The dependence on $\log n$ can be dropped if superpolynomial preprocessing is allowed.) If needed, the actual list of edges on the path can then be retrieved in time proportional to the list size.

Our motivation for the definition of highway dimension were the experiments performed by Bast et al. [1, 2], who exploited a very intuitive observation: when driving on a shortest path from a compact region of a road network to points that are "far away," one must pass through one of a very small number of *access nodes*.

For the US road network, the preprocessing algorithm of Bast et al. [1, 2] finds a set of approximately 10,000 transit nodes that "cover" 99% of all shortest paths, omitting only those shortest paths whose endpoints are very close to each other. Additionally, for a vertex v, removing about 10 of these transit nodes from the road network would increase the length of all sufficiently long shortest paths emanating from v. Furthermore, a variant of the algorithm uses multiple layers of transit nodes to handle local queries efficiently; the average number of access nodes in the more "local" layers is just as small. This strongly indicates that real road networks exhibit small highway dimension, at least in some average sense.

One can view this road network model as being somewhat analogous to small-world models for social networks [16, 18]. Although real social networks do not look exactly like small-world graphs, the latter give insight into and allow complexity analysis of various routing algorithms. Similarly, while real road networks may have anomalies that do not agree with small highway dimension, highway dimension gives insight into and allows rigorous analysis of many shortest path algorithms that actually work astonishingly well in practice.

To complement the experimental evidence, we seek to explain why it is reasonable to assume that real road networks have low highway dimension. Consider the following scenario: Legionnaire veterans are sent to form new coloniae (e.g. Berytus—now Beirut). Such new cities should be connected by roads to the existing road network. Roman roads were either long and fast primary roads (*viae*), shorter and slower secondary roads (*viae rusticae*), or even shorter and even slower dirt roads (*viae terrenae*). Thus, there are two different metrics involved, distance and time. Roman road planning seeks travel-time efficient roads, without excessive expenditure. A natural approach is to ensure that one does not need to follow dirt roads for too long before transferring to a better (faster) road. Moreover, one does not seek to add too many expensive roads to the network. A natural greedy approach is to connect the new colonia via primary roads only if it is sufficiently far from any entry point into such roads. Given that Rome is colonizing the known world (constant doubling dimension), this implies that the time metric has constant highway dimension.

Motivated by the (somewhat tongue in cheek) discussion above, we suggest what could be a plausible generative model for road networks, and show that the networks it produces have low highway dimension (see Section 6). This provides a possible explanation for the emergence of low highway dimension networks. Our model captures the incremental manner in which roads are added over time, the fact that the underlying geometric structure on which roads are built has low doubling dimension, and the observation that long highways are typically faster to drive on than shorter roads. These results also allow one to generate synthetic networks with low highway dimension.

The notion of highway dimension may be interesting on its own. Conceivably, better algorithms for other problems can be developed and analyzed under the small highway dimension assumption.

For simplicity and clarity of exposition, in most of this paper we deal with undirected graphs. In Section 3.1, we comment on how to extend our results to directed graphs.

This paper is organized as follows. Section 2 establishes some basic notation, definitions, and background. Section 3 formally introduces the notion of highway dimension. Section 4 describes our preprocessing algorithm. In Section 5, we prove that various shortest-path heuristics are space- and time-efficient on networks of low highway dimension. Section 6 presents our generative highway model. In Section 7 we conclude with some final remarks.

2 Definitions and Dijkstra's Algorithm

The input to the preprocessing stage of a shortest path algorithm is an undirected graph $G = (V, E)$ with length $\ell(e) > 0$ for every edge e. For simplicity, we assume that all shortest paths are unique and that G is connected. We also normalize the graph so that the minimum length of an edge is one.

Let $P(u, v)$ denote the shortest path from u to v, and let $|P(u, v)|$ be its length (the sum of its edge lengths). We assume that every edge $e \in E$ is the shortest path between its endpoints (otherwise we can delete e from G). Given a non-negative r, let $B_{u,r} = \{v \in V, |P(u, v)| \leq r\}$ be the ball of radius r centered at u. Let $D = \max |P(u, v)|$ be the diameter of the network, and let Δ be the maximum degree of a vertex in G.

Dijkstra's algorithm is an efficient implementation of the scanning method for graphs with non-negative edge lengths (see e.g. [21]). For every vertex v, it maintains the length $d(v)$ of the shortest path from the source s to v found so far, as well as the predecessor $p(v)$ of v on the path. Initially $d(s) = 0$, $d(v) = \infty$ for all other vertices, and $p(v) = \textit{null}$ for all v.

Dijkstra's algorithm maintains a priority queue of unscanned vertices with finite d values, the values serving as keys. At each step, the algorithm extracts the minimum valued vertex, v, from the queue and scans it. I.e., the algorithm looks at all edges $(v, w) \in E$ and, if $d(v) + \ell(v, w) < d(w)$, sets $d(w) = d(v) + \ell(v, w)$ and $p(v) = w$. The algorithm terminates when the target t is extracted, without scanning t.

The bidirectional version of Dijkstra's algorithm is similar, but it runs a forward search from s and a reverse search from t. When an edge (v, w) is scanned by the forward search and w has already been scanned by the reverse search, the concatenation of paths s–v and w–t is a new path P from s to t (the same holds in the reverse search). The algorithm keeps track of the shortest such path found during the execution; when the searches meet, this path will be optimum.

3 Highway Dimension

To explain and give some justification to the observations of Bast et al. [1, 2] mentioned above, we propose the notion of *highway dimension*:

Definition 1. **[Highway dimension]** *Given an edge-weighted graph $G = (V, E)$, the highway dimension of G is the smallest integer h such that*

$$\begin{aligned} \forall \quad & r \in \mathcal{R}^+, \forall u \in V, \exists S \subseteq B_{u,4r}, |S| \leq h, \textit{ such that} \\ \forall \quad & v, w \in B_{u,4r}, \\ & \qquad \textit{if } |P(v, w)| > r \textit{ and } P(v, w) \subseteq B_{u,4r} \\ & \qquad\qquad \textit{then } P(v, w) \cap S \neq \emptyset. \end{aligned}$$

The definition says that for every r and every ball of radius $4r$, a small set of vertices covers all shortest paths of length greater than r which are inside the ball. Note that one could use constants bigger than 4, but then the constant in Definition 2 below should be adjusted appropriately.

The findings of Bast et al. suggest that real road networks may have low highway dimension.

We note that this definition is related to that of *doubling dimension.* A graph is said to be α doubling (or to have doubling dimension $\log \alpha$) if every ball can be covered by at most α balls of half the radius. Doubling and highway dimension are not the same, however. A square grid with unit lengths is an example of a graph of constant doubling dimension that has large $(\Theta(\sqrt{n}))$ highway dimension. A star graph with unit edge lengths has constant highway dimension and large $(\Theta(\log n))$ doubling dimension.

However, the star graph is in some sense an exception. We show that for "continuous" graphs, small highway dimension implies small doubling dimension. By continuous graphs we mean graphs where each edge is viewed as infinitely many vertices of degree two with infinitely small edges (formally the continuous graph is the geometric realization of the graph topology).

Claim 1. *If the geometric realization of the graph topology of G has highway dimension h, then its shortest path metric is h doubling.*

Proof. Consider a ball $B = B_{u,4r}$ and a set S with $|S| \leq h$ such that every shortest path P in B with $|P| > r$ contains an element of S. We claim that the union of the balls of radius r around the elements of S contains B. Suppose there is a vertex $v \in B$ not covered by the union, and let w be a vertex of S that is closest to v. Then the shortest w–v path Q does not contain any element of S as an internal vertex (or w would not be the closest vertex) and $|Q| > r$ (or v would be in the ball around w). This contradicts the choice of S. ■

It seems that a notion stronger than doubling dimension is indeed necessary to fully explain the success of speedup heuristics on road networks. It has been shown experimentally [11] that some of the speedup heuristics do not perform as well on planar grids as they do on road networks, even though both classes of graphs have low doubling dimension. Highway dimension might be necessary to explain the difference between these classes.

Next we define *shortest-path covers* and relate them to highway dimension.

Definition 2. [(r,k) **Shortest-Path Cover ((r,k)-SPC)**] *A set C is an (r,k)-SPC of G if and only if* $\forall u \in V,\ |C \cap B_{u,2r}| \le k$ and $\forall$ shortest path $P:\ r < |P| \le 2r,\ P \cap C \ne \emptyset$.

Intuitively, an (r,k)-SPC is a set of vertices that covers all paths of length between r and $2r$ and is locally sparse, i.e., has a small intersection with every ball of radius $2r$.

The constants in definitions 1 and 2 are chosen to enable the proof of the following lemma, which also relies on the upper bound on $|P|$ in Definition 2.

Lemma 3.1. *If G has highway dimension h, then for any r there exists an (r,h)-SPC of G.*

Proof. Let S^* be the smallest set that covers all shortest paths P satisfying $r < |P| \le 2r$. We prove that S^* is an (r,h)-SPC. Suppose by way of contradiction that for some u, $U = S^* \cap B_{u,2r}$ and $|U| > h$. By the definition of h, there is a set H, with $|H| \le h$, covering all shortest paths in $B_{u,4r}$ of length greater than r. In particular, H covers all shortest paths of length between r and $2r$ covered by U. Therefore $(S^* - U) \cup H$ is smaller than S^* and covers all shortest paths of length between r and $2r$, contradicting the optimality of S^*. ■

The natural exhaustive enumeration of all vertex subsets of size h or less gives an algorithm with running time $n^{O(h)}$. Adapting the greedy approximation algorithm for set cover [15] gives a polynomial-time construction ($n^{O(1)}$ time independent of the highway dimension) with a logarithmic approximation factor.

Lemma 3.2. *If G has highway dimension h, then for any r we can construct, in polynomial time, an $(r, O(h \log n))$-SPC.*

Proof. Starting from an empty set, repeatedly choose a vertex that covers the most uncovered paths, breaking ties arbitrarily. It is easy to see that this algorithm runs in polynomial time. We must show that the set it returns is an $(r, O(h \log n))$-SPC.

Pick $v \in V$ and let B_1 and B_2 be the balls centered at v of radius $2r$ and $4r$, respectively. By Definition 1, there is a set S in B_2 with $|S| \le h$ such that every shortest path in B_2 of length at least r is covered by S. We say that a shortest path P is *relevant* if its length is between r and $2r$ and P intersects B_1. Note that all relevant paths are contained in B_2 and are covered by S.

Suppose at some step the algorithm chooses a vertex w in B_1. Every path covered by w must be relevant, and by the greedy choice of w and the fact that the relevant paths are covered by S, w covers at least $1/h$ of the currently uncovered relevant paths. As the initial number of relevant paths is $O(n^2)$, the algorithm can choose $O(h \log n)$ vertices in B_1. ■

Note that one can use an alternative definition of highway dimension by defining it to be the smallest h for which (r,h)-SPC exists for all $r > 0$. For this definition, an argument similar to that used in the proof of Lemma 3.2 can be used to construct $(r, O(h \log n))$-SPC in polynomial time. We think that our original definition is cleaner. However, the alternative definition can be extended to directed graphs, as we discuss below.

3.1 Directed Graphs

In this section we extend the results to directed (asymmetric) graphs. First we define a (directed) ball. The *directed ball* $B_{u,r}$ is the subgraph of G induced by vertices v such that $\text{dist}(u,v) \le r$ or $\text{dist}(v,u) \le r$.

Unfortunately our proof of Lemma 3.1 does not work in the directed case. One way to extend the results is to use the alternative definition of the highway dimension: the smallest h for which an (r,h)-SPC exists for all r.

Another way to handle directed graphs is to assume that asymmetry is limited. For $\epsilon > 0$, we say that a graph is *ϵ-symmetric* if for all v, w we have $\text{dist}(v,w) \le (1+\epsilon)\text{dist}(w,v)$. One can show that, with appropriate change to constant factors in the definitions and the proofs, the results for undirected graphs extend to the ϵ-symmetric graphs.

4 Preprocessing

This section describes our basic preprocessing algorithm. In Section 5, this basic construct is used to show that the RE algorithm [10] is efficient on networks with low highway dimension. Moreover, with small modifications (which will be described as needed), the algorithm can also be applied to obtain variants of CH, HH, TN, and SHARC that are provably efficient. Before we get to our preprocessing algorithm, we describe an idea of Geisberger et al. that inspired it: *contraction hierarchies* [8].

4.1 Contraction Hierarchies (CH) and Shortcuts

Most state-of-the-art shortest-path heuristics, including the CH algorithm, depend crucially on a very simple notion: *shortcuts* [20]. Let $u, v \in V$ be two vertices such that the distance between u and v is d. A shortcut is a new edge $e = (u,v)$ with length d. The *shortcut operation* deletes a vertex v from the graph and adds edges between its neighbors to maintain the shortest path information. In particular, for any neighbors u, w such that $(u,v) \cdot (v,w)$ is the shortest path between u

and w and there is no alternative shortest path that does not use v, we add (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. The addition of shortcuts breaks the invariant that shortest paths in the input graph are unique. Breaking ties by favoring paths with fewer edges is sufficient for our purposes, even though it does not eliminate all ties.

Given the notion of shortcuts, CH preprocessing is trivial: define a total order among the vertices and shortcut them sequentially in this order, until a single vertex remains. The output of this routine is the set E^+ of shortcut edges, as well as the vertex order itself. Queries will always be correct, irrespective of the contraction order — see query processing below.

However, query complexity and the size of the auxiliary data required may vary greatly from one permutation to the next. The best results reported in [8] are obtained by on-line heuristics that select the next vertex to shortcut based on its current degree and the number of new edges added to the graph, among other factors. We denote the position of a vertex v in the ordering by $\mathit{rank}(v)$.

An s–t CH query essentially runs bidirectional Dijkstra search on the graph $G^+ = (V, E \cup E^+)$. However, when scanning v, only the edges (v, w) with $\mathit{rank}(v) < \mathit{rank}(w)$ are examined. The search terminates when there is no labeled vertex in either direction. At this point, each vertex v has estimates $d_s(v)$ and $d_t(v)$ on distances from s to v and from v to t. (Unlike in Dijkstra's algorithm, these estimates may be greater than the actual distances for some vertices.) A vertex u minimizing $d_s(u) + d_t(u)$ is on a shortest path from s to t, given by the concatenation of the s–u and u–t paths.

This remarkably simple algorithm is surprisingly efficient on road networks. On the European network, random queries visit fewer than 500 vertices (out of 18 million) on average. Preprocessing takes only 10 minutes on a workstation and adds fewer shortcuts than there are original edges.

4.2 The Common Preprocessing Algorithm

We are now ready to describe our preprocessing algorithm. As in CH preprocessing, we must order the vertices and shortcut them in this order. This yields the ordering and the set E^+ of shortcuts. To specify the ordering, we use a sequence of shortest-path covers.

Specifically, let $S_0 = V$ and for $1 \leq i \leq \log D$, let S_i be an $(2^i, k)$-SPC cover. Our algorithm computes these covers. Here k is h if we do not restrict preprocessing time (Lemma 3.1) and k is $O(h \log n)$ for polynomial-time preprocessing. Let $L_i = S_i - \bigcup_{j=i+1}^{\log D} S_j$. In our contraction order, the vertices in L_i come before those in L_{i+1}. Within each L_i, the ordering is arbitrary.

Lemma 4.1. *If $v \in L_i$, the number of edges $(v, w) \in E^+$ with $w \in L_j$ (for fixed $j \geq i$) is at most k.*

Proof. By construction, (v, w) represents a shortest path P in the original graph. Because the internal vertices of P were eliminated before v and w, they belong to L_x for some $x \leq i \leq j$. Also, w must belong to $B_{v,2\cdot 2^j}$. If not, the length of P would be more than $2 \cdot 2^j$, but P does not contain a vertex from L_y with $y > j$. Since $B_{v,2\cdot 2^j}$ has at most k vertices from L_j, the lemma follows. ■

Note that this only bounds the number of shortcuts that connect v to higher levels. We can obtain a similar bound for lower levels as well:

Lemma 4.2. *If $v \in L_i$, the number of edges $(v, w) \in E^+$ with $w \in L_j$ (for fixed $j < i$) is at most k.*

Proof. The shortcut (v, w) is created when we remove the last internal vertex z on the shortest path $P(v, w)$ in the underlying graph. Therefore, $z \in L_t$, for some $t \leq i$. If $|P(v, w)| > 2^{t+1}$, there must be a vertex of $L_{t'}$ (with $t' \geq t + 1$) on the path P, which contradicts our assumption that z was the last vertex removed. Because $t \leq i$, we must have $|P(v, w)| \leq 2^{t+1} \leq 2^{i+1}$. Since there can be no more than k vertices of L_i in $B(w, 2^{i+1})$, the lemma follows. ■

With the results in Section 3, these lemmas imply the following bounds on preprocessing:

Theorem 4.1. *For any graph $G = (V, E)$ with highway dimension h and diameter D there is an ordering of vertices that causes CH preprocessing to produce E^+ such that degree of every vertex in $G(V, E \cup E^+)$ is at most $\Delta + h \log D$ and $|E^+| = O(nh \log D)$. For poly-time preprocessing, the degree bound is $O(\Delta + h \log n \log D)$ and the $|E^+|$ bound is $O(nh \log n \log D)$.*

Note that the query returns a path in the augmented graph (and its length). In applications where the corresponding original path is required, we must translate each shortcut into a sequence of original edges. This can be trivially done in time linear in the size of the sequence [20], as long as we remember (as part of the auxiliary data), which were the two elements (edges or shortcuts) combined to create each shortcut added during preprocessing. This requires $O(|E^+|)$ space.

5 Query Complexity

We now study the complexity of queries for various heuristics: RE, CH, TN, and SHARC.

5.1 Reach (RE)

The RE algorithm [10] is based on the notion of *reach* [13]. Given a path P and a vertex $v \in P$ that divides P into P_1 and P_2, the reach of v w.r.t. P is $r_P(v) = \min\{|P_1|, |P_2|\}$. Let $p(v)$ be the set of *shortest* paths containing v. The *reach* of v (w.r.t. the entire graph) is $r(v) = \max_{P \in p(v)} r_P(v)$.

The preprocessing stage of the original RE algorithm heuristically adds shortcuts to the graph and computes upper bounds r on reaches in the augmented graph. An s–t query performs bidirectional Dijkstra search with pruning by reach. When the forward search labels a vertex v with distance label $d(v)$, it checks if $r(v)$ is smaller than the minimum of $d(v)$ and B, the distance label of the top heap element of the backward search. If so, it does not add v to the priority queue (if v were on the shortest path from s to t, its reach would be at least $\min\{d(v), B\}$). Symmetric pruning is done for the backward search.

Bidirectional Dijkstra search will give correct answers irrespectively of when one switches between forward and backward searches. We consider the [effective] variant that balances the two directional searches by distance traversed: repeatedly pick either forward or backward search, choosing the direction whose minimum labeled vertex distance is smaller (break ties arbitrarily).

To obtain provably good query times, we use the shortcuts E^+ generated by the common preprocessing algorithm described in Section 4.2. Intuitively, adding shortcuts reduces reaches, and the way the algorithm adds shortcuts also yields reach bounds sufficiently good for our analysis:

Lemma 5.1. *For any $v \in L_i$, $r(v) \le 2 \cdot 2^i$ in $G^+ = (V, E \cup E^+)$.*

Proof. By way of contradiction, suppose that $r(v) > 2 \cdot 2^i$. Then, there is a shortest path P in G^+ between a vertex x and a vertex y such that

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

Both P_1 and P_2 must contain vertices $u \in L_j$ for $j > i$. Among those, let u_1 and u_2 be the closest vertices to v on P_1 and P_2, respectively. It follows that all vertices of P between u_1 and u_2 (including v) will be shortcut before u_1 and u_2, which means (u_1, u_2) must be a shortcut. Since we break ties by number of hops, the path using the shortcut is shorter than P. This contradicts the assumption that P is a shortest path. ∎

With these bounds, we can prove the following about the original RE query algorithm:

Theorem 5.1. *Consider the variant of RE that balances the two searches based on the radii searched thus far. There is an ordering of vertices during preprocessing such that RE query takes $O((\Delta + h \log D)(h \log D))$ time, and a poly-time computable ordering such that the query takes $O((\Delta + h \log n \log D)(h \log n \log D))$ time.*

Proof. For the forward search from s, consider the ball $B_{s,2\cdot 2^i}$. The search does not scan any $v \in L_i$ such that v is outside the ball. This is because $r(v) \le 2 \cdot 2^i$, which implies that v is either scanned by the reverse search or not scanned at all. Therefore the search scans at most $O(k \log D)$ vertices. A similar argument holds for the reverse search. The fact that vertex degrees are bounded by $\Delta + k \log D$ completes the proof. ∎

Note that RE does not need the vertex ordering but needs reach bounds. For a vertex $v \in L_i$, we can store $2 \cdot 2^i$ (which can be represented by i) as its reach bound. The result also holds for any reach upper bounds that are at least as good as those computed by our preprocessing. In particular, it holds for optimal reach values in the graph with added shortcuts, which can be computed in polynomial time.

Note that Theorem 5.1 ignores data structure overhead. Using the appropriate heap data structure, all overhead can be amortized, with the exception of *deletemin* operations, which precede every vertex scan. Recall that the number of vertex scans is $O(k \log D)$. Using Fibonacci heaps [7], this leads to an additive term of $k \log D \log n$. Assuming integral edge lengths and defining C as the ratio between the maximum and the minimum positive edge lengths, we can use multi-level bucket data structure [5] for a $k \log D \log C$ additive term. Since $D \ge C$, this term is dominated by the bound of the theorem. From now on, we will ignore the data structure overhead.

5.2 Contraction Hierarchies (CH)

We now return to the contraction hierarchies algorithm (CH), described in Section 4.1. Recall that its query is essentially bidirectional Dijkstra with additional pruning: when scanning an edge (v, w) from v, w is labeled (added to the heap) only if $\mathit{rank}(v) < \mathit{rank}(w)$.

To get similar bounds to Theorem 5.1, we must prove that a CH query visits at most $O(k)$ vertices in each level L_i. It would be sufficient to show that all vertices visited by the forward search at level i are within distance at most $2 \cdot 2^i$ from the source s (a similar argument would hold for the backward search). This is almost true. If a vertex $v \in L_i$ is such that $|P(s,v)| \geq 2 \cdot 2^i$, then there must be a vertex $u \in L_j$ (with $j > i$) on $P(s,v)$. This means that $rank(u) > rank(v)$, so the forward CH search would never follow path $P(s,v)$ in its entirety (as desired). Because of pruning, however, not every branch of the search tree followed by CH is a shortest path. Conceivably, the search could find an alternative path (not shortest) of increasing ranks from s to v.

We propose two alternative solutions to this issue: modifying CH queries to strengthen the pruning condition, or modifying CH preprocessing only. We describe each approach in turn.

5.2.1 CH with Range Optimization

A simple way to fix CH is to add *range optimization*: When scanning an edge (v,w) with $w \in L_j$, if the label for w from the scan is $d(w) > 2 \cdot 2^j$, we do not label w, i.e., we do not put w on the priority queue. (Because a shortest path of length greater than $2 \cdot 2^j$ must pass through a vertex in L_{j+1}, we know the current path to w is not the shortest.) We remark that this modification requires knowing the level of a vertex from the preprocessing, which can be implemented with constant overhead per vertex. This approach can be seen as implicitly adding reach pruning to CH. In particular, the same time bounds hold.

Theorem 5.2. *There is an ordering of vertices such that CH query with range optimization takes $O((\Delta + h \log D)(h \log D))$ time, and a poly-time computable ordering such that the query takes $O((\Delta + h \log n \log D)(h \log n \log D))$ time.*

Although our bounds are the same, CH with range optimization would probably be better than RE in practice, since CH can perform pruning by rank as well.

5.2.2 Additional Shortcuts

Adding range optimization to CH is natural, but somewhat unsatisfactory: the entire analysis follows from RE. We now consider an alternative approach that keeps the original CH query algorithm intact. All we need is a slightly modified version of the preprocessing algorithm of Section 4.2.

We shortcut vertices in the same order as before. When we shortcut vertex $v \in L_i$, we still create edges between its neighbors as needed. However, the modified version creates some *additional* edges: for every pair $\{u,w\} \subseteq B_{v,2^{i+1}}$ (in the current graph) such that $v \in P(u,w)$, we create a new edge (u,w) with length $|P(u,w)|$. Note that, even with these extra edges, Lemma 4.1 still applies. We denote the augmented set of shortcuts (which includes E^+) by E^*.

We also change the output of the preprocessing algorithm: instead of producing a total order, it sets $rank(v) = i$ for all vertices at level i (intuitively, we allow ties in the global vertex order). Queries remain unchanged: run bidirectional Dijkstra with pruning by rank (when scanning an edge (v,w), only label w if $rank(v) < rank(w)$).

The modified shortcut strategy ensures that for every original shortest s–t path there is a corresponding path in $G^+ = (V, E \cup E^*)$ with no consecutive vertices on the same level. The one possible exception is the highest level, which may contain two (adjacent) vertices, but not more. We handle the special case as follows: if, when scanning a vertex v in the forward direction, we find a neighbor w on the same level that has been scanned in the reverse direction, we check if $d_s(v) + \ell(v,w) + d_t(w)$ is the shortest path seen so far. (We perform a similar test during the reverse search.) Note, however, that CH pruning still applies: we do *not* add w to the heap.

The performance of this version of the CH query is the same as RE.

Theorem 5.3. *Consider the variant of CH with additional shortcuts. The total number of shortcut edges is bounded by $|E^*| = O(nh \log D)$ or $O(nh \log n \log D)$ for poly-time preprocessing. The query takes $O((\Delta + h \log D)(h \log D))$ time or $O((\Delta + h \log n \log D)(h \log n \log D))$ for polynomial-time preprocessing.*

Note that this version of the CH algorithm can also be seen as a stronger version of highway hierarchies (HH) [20]. A predecessor of CH, HH uses a partial ordering of the vertices, with incomparable sets forming hierarchy levels. The query algorithm never goes down the hierarchy, but it is allowed to move "sideways" (between vertices on the same level). In our variant of CH, each L_i acts as a level in the highway hierarchy, but we only allow the searches to go strictly up.

We still have one issue to address: retrieving the original shortest paths. Each extra shortcut we add may represent an original path with several edges (not just two, as in Section 4.2). Consider a shortcut (u,w) added when eliminating $v \in L_i$: because all lower-level vertices have been eliminated at this point, this shortcut corresponds to a path of at most k vertices in $B_{v,2 \cdot 2^i}$. We store this path with the new shortcut. This extra information requires $O(nk^2 \log D)$ space in total.

5.3 Transit Nodes (TN)

We now consider the TN algorithm [1]. As already mentioned, this algorithm is based on the observation that anyone driving from a small region to faraway destinations must pass through one of a small number of *access nodes*. The union of all access nodes constitutes the set of *transit nodes*.

The TN preprocessing algorithm uses heuristics to find a compact set of transit nodes that is locally small, i.e., any vertex v has a small set $A(v)$ access nodes contained in the set of transit nodes. It then adds shortcuts between each vertex v and its access nodes. Finally, it computes and stores a (quadratic-sized) sized table of distances between pairs of transit nodes. (Note that each table entry effectively corresponds to a shortcut in the original graph.)

To answer an s–t query, the algorithm first uses a fast *distance filter* (based on geometric distances) to estimate how close s and t are. If the estimated distance is small, a Dijkstra-based algorithm is used to find the actual shortest path. Otherwise (if the estimated distance is large enough) the algorithm explicitly looks at all three-edge paths of the form (s, s', t', t), where $s' \in A(s)$ and $t' \in A(t)$. The smallest such path is the shortest path from s to t. Note that this can be done in time $O(|A(s)| \cdot |A(t)|)$ using the table computed during preprocessing.

Further improvements to the TN algorithm [2] include the use of highway hierarchies to speed up preprocessing and local queries, and hierarchical transit nodes.

This algorithm is remarkably fast in practice. On continental-sized maps, the preprocessing algorithm can find a set of approximately 10 000 transit nodes such that every vertex has about 10 access nodes. This means that long-range queries (more than 99% of the total) can be performed with about 100 table lookups, which takes a few microseconds. Note that, unlike all other algorithms we study, long-range TN queries are not Dijkstra-based.

A variant of the Transit Node algorithm. We propose the following TN variant: during an s–t query, consider paths of the form (s, x, t), with $x \in A(s) \cap A(t)$. Note that these paths have at most one intermediate vertex, instead of two as in the original transit node algorithm.

We further modify the preprocessing of Section 4.2 as follows. After computing the sets S_i and L_i, we add shortcuts connecting each vertex v to all vertices in S_i within distance $2 \cdot 2^i$ from v. Let E' be the set of edges thus added. From Lemma 4.1, $|E'| \leq nk \log D$ (where $k = h$ or $k = O(h \log n)$). The TN query variant works on the graph $G' = (V, E \cup E')$: given s and t, we look only at edges in $E \cup E'$ adjacent to s and t, and pick the shortest one- or two-edge path discovered in the process.

To see that the query algorithm is correct, let i be such that $2^{i-1} < |P(s,t)| \leq 2^i$. If $(s,t) \in E \cup E'$, we find the shortest path. Otherwise, the shortest path between s and t contains a vertex $v \in S_{i-2}$, and both s and t are connected to v, so we find the shortest path as well.

THEOREM 5.4. *The total number of shortcut edges is $|E'| = O(nh \log D)$, or $O(nh \log n \log D)$ for poly-time preprocessing. If we do not restrict preprocessing time, TN query takes $O(\Delta + h \log D)$ time, and for poly-time preprocessing the query takes $O(\Delta + h \log n \log D)$ time.*

Note that these bounds are better than for CH, HH, and RE. The space bound is asymptotically the same, unless we need to output the original path in linear time.

As every edge in E' corresponds to a large number of edges in E, we cannot afford to store this mapping directly. Instead, the preprocessing algorithm outputs a representation of each edge in E' as a sequence of edges in $E \cup E^+$ (recall that E^+ is the set of shortcuts added by the common preprocessing algorithm). Every shortcut is a shortest path, and any shortest path in $G^+ = (V, E \cup E^+)$ has at most $k \log D$ edges (no more than the number of scans performed by RE in G^+). Thus, an upper bound on the total space required by our extended representation is $O(n(k \log D)^2)$.

5.4 The SHARC algorithm

The SHARC algorithm [3] combines the ideas of arc flags [17, 14] and shortcuts. The core idea of arc flag preprocessing is to label nodes (assigning them to “regions”) and attach additional information to each arc (each edge (u, v) consists of two arcs $u \to v$ and $v \to u$).[1] Given the target's label, a modified Dijkstra checks whether an arc can or cannot be on the shortest path to the target. In SHARC the labeling of nodes is done using an iterative and hierarchical heuristic partitioning. We note that this approach has similarities to the notion of labeled routing in distributed graph algorithms [22].

One of the main advantages of SHARC is that it provides good empirical results even when executed in a unidirectional manner. Unidirectional Dijkstra variants are particularly important for time-dependent route planning (bidirectional Dijkstra cannot easily perform a reverse search from the target, since the arrival time is not known).

While SHARC uses heuristics to compute shortcuts and to decide how to partition the map to compute arc flags, we suggest a modification of SHARC that uses low highway dimension to compute both the shortcuts and the labels of each vertex and edge. Roughly speaking, an arc $u \to v$ is given the flag of vertex w if the

[1]To avoid confusion we use “arc” to mean “directed edge”.

shortest path from u to w goes through v, and w belongs to some S_i and the distance between u and w is at most 2^{i+1}. The label of each vertex is approximately the list of the k nearest points in S_i for each scale i. Hence if $2^i < d(s,t) \leq 2^{i+1}$ then there exists a vertex $w \in S_i \cap P(s,t)$ and by definition there will be a shortcut from s to w, the arc $s \to w$ is given the flag w and w is part of the label of t.

1. **Preprocessing**. The preprocessing algorithm is the same as for TN. If $v \in S_i$ and $P|(u,v)| \leq 2^{i+1}$ then we add a shortcut arc from u to v. Note that each vertex creates at most $O(k \log D)$ out-going edges, so the total memory is $O(nk \log D)$.

2. **Arc flags**. An arc $u \to v$ is given the flag of vertex w if $w \in S_i$, $|P(u,w)| \leq 2^{i+1}$, and $v \in P(u,w)$. Note that u can have at most $k \log D$ flags. Recall that $P(u,w)$ is defined on the graph $G^+ = (V, E \cup E^+)$ that includes the shortcuts and breaks ties by choosing the shortest paths with the least number of hops.

3. **Vertex labels**. The label of a vertex u is the sequence $F(u) = F_1, \ldots, F_{\log D}$, where $F_i = S_i \cap B_{u,2^{i+1}}$. Note that $F(u)$ contains at most $k \log D$ elements. Hence storing a table mapping each vertex to its label requires $O(nk \log D)$ space.

4. **Query**. Given a target t, at any vertex u we modify Dijkstra to consider only arcs $u \to v$ such that $v \in F_j(t)$, where $j = \min\{i \mid \exists u \to w, w \in F_i(t)\}$.

THEOREM 5.5. *The modified SHARC algorithm has polynomial-time preprocessing requiring space $O(nh \log n \log D)$ and such that each query takes $O((h \log n \log D)^2)$ time.*

Proof. Given a fixed target, there are at most $O(h \log n \log D)$ arcs that the modified Dijkstra can reach from the source (the arcs leading to vertices in the label of the target). From each such vertex the only arcs taken are those to vertices in the label of the target. Therefore the total number of vertices explored is $O(h \log n \log D)$, and the total possible arcs explored is $O((h \log n \log D)^2)$. It is easy to verify that the shortest path from s to t must be included in this search. ■

We can similarly obtain a (superpolynomial-time) preprocessing algorithm that enables unidirectional queries to take $O((h \log D)^2)$ time. Finally, we note that, as for the TN auxiliary space requirements, we need more auxiliary data to output the path in the original graph efficiently.

6 Emergence of Networks with Low Highway Dimension

The formation of real road networks is a complex process involving geographic, economic, political, and cultural aspects. In this section we propose a simple model that hopefully captures some of these aspects. We then show that networks formed by this process have low highway dimension. This provides a plausible explanation for the emergence of low highway dimension networks. We do not claim our model captures all aspects of road network creation (just as the small-world models for social networks [16, 18] do not claim to capture all aspects of social network topology).

We would like to capture three properties of road network formation.

1. Roads are built in an incremental manner over time, and each decision is typically done in a local manner—without necessarily taking into account a centralized global planner. Hence we consider a *decentralized* and *on-line* process of forming a road network.

2. The underlying geometric space on which roads are built has some low-dimensional structure. To capture this property, we assume that the underlying geometric space has a low doubling dimension. (Recall that a metric has doubling dimension $\log \alpha$ is every ball of radius $2r$ can be covered by α balls of radius r.)

3. Longer highways are typically faster than short roads. (For example, interstate highways are typically faster than state highways, which are faster than local inter-neighborhood roads, which are faster than small inter-neighborhood roads, and so on.) To formalize this we introduce a *speedup parameter* $0 < \delta < 1$ and define the *traversal time* $\tau(u,v)$ of a road segment with endpoints u, v to be $d(u,v)^{1-\delta}$.

Consider the following model that is similar in spirit to the dynamic spanner construction of [12]. Start with a metric space (M,d) with doubling dimension $\log \alpha$. An on-line adversary supplies a sequence of points $v_1, v_2, \ldots \subseteq M$, where v_t is the new location added at time t. When v_t arrives, we need to connect v_t to the existing road network. We do this by connecting v_t to nearby peers at appropriate scales.

Intuitively, if v_t is a new city, we connect it to nearby cities; if it is a new neighborhood, we connect it to nearby neighborhoods; etc. However, if neighborhoods are created further and further away from the city center, there comes a point where we connect the new neighborhood not only to nearby neighborhoods, but also to nearby city centers.

Formally, let D be the diameter of the metric space. For each integer $0 \le i \le \log D$ we maintain a set C_i such that any two vertices in C_i are at least 2^i apart (in the metric space). We say that C_i is a 2^i-cover. Our covers are hierarchical, i.e., if $v \in C_i$, v is in C_j for all $j < i$. All vertices are in C_0. We say that a vertex v is a *level i vertex* if i is the maximum index for which $v \in C_i$.

When a new point v_t arrives, we find the lowest i for which there is a vertex at distance 2^i from v and place v_r into all C_j for $0 \le j < i$. (If no such i exists, we place v_r into all covers.)

After determining which covers v_r belongs to, we connect it to other vertices as follows. For each i such that v_r is in C_i, add two types of edges from v_r:

1. Edges to all vertices in $B_{v_t,6\cdot 2^i} \cap C_i$ other than v_r. Note that it is possible that $B_{v_t,6\cdot 2^i} \cap C_i = \{v_r\}$, i.e., no edges are added. For each such edge a, note that $2^i \le d(a) \le 6 \cdot 2^i$.

2. If $i < \log D$ and $v_r \notin C_{i+1}$, we connect v_r to the closest vertex in C_{i+1}. Note that the length of such an edge is at most 2^{i+1} because we have not added v_r to C_{i+1}. The length is also greater than 2^i, since the vertex we connect v_r to is in C_i.

Note that we add at most $\alpha^4 \log D$ edges. This is because for each vertex v_r, we add at most α^4 edges for each $0 \le i \le \log D$. To see this, we need to prove that $B_{v_r 6\cdot 2^i}$ contains at most α^4 vertices in C_i. In fact, we show this for $B_{v_r 8\cdot 2^i}$. By the definition of α, $B_{v_r 8\cdot 2^i}$ can be covered by α^4 balls of radius 2^{i-1}. By the definition of C_i, each of the balls in the cover contains at most one vertex of C_i.

Recall that we have a speedup parameter $\delta \in (0,1)$, so a highway of geographic length $d(u,v)$ has a traversal time of $\tau(u,v) = d(u,v)^{1-\delta}$. To simplify the analysis, we fix $\delta = 0.25$. The proofs below can be modified to show that Theorem 6.1 holds for any constant $\delta \in (0,1)$. We keep symbolic δ in the exponents of expressions to make this more transparent.

We shall refer to shortest paths with respect to traversal times as *fastest paths*. By shortest paths we will mean shortest paths with respect to d.

For the rest of this section, let $x \ne y \in V$ be such that $2^i < d(x,y) \le 2^{i+1}$. Let Q be a fastest x–y path and let a be the longest edge on Q.

LEMMA 6.1. *For $\delta = 0.25$, there exists an x–y path P such that $\tau(P) < 9 \cdot 2^{i(1-\delta)}$.*

Proof. Consider the following recursive construction of a sequence of vertices $x = x_0, x_1, \dots, x_i$. Given x_{j-1}, let x_j be x_{j-1} if $x_{j-1} \in C_j$; otherwise, let x_j be the vertex in C_j we connected x_{j-1} to when adding it to the graph. Note that $d(x_{j-1}, x_j) \le 2^j$. Therefore there is a path P_x from x to $x_i \in C_i$ of length $d(P_x) \le \sum_{j=1}^{i} 2^j \le 2 \cdot 2^i$. The transit time $\tau(P_x)$ is at most

$$\sum_{j=1}^{i} 2^{j(1-\delta)} \le \frac{2^{(i+1)(1-\delta)} - 1}{2^{(1-\delta)} - 1} < 2.5 \cdot 2^{i(1-\delta)}.$$

The last step uses the fact that $\frac{2^{(1-\delta)}}{2^{(1-\delta)}-1} < 2.5$ for $\delta = 0.25$.

Similarly, there is a path P_y from y to $y_i \in C_i$ with $d(P_y) \le 2 \cdot 2^i$ and $\tau(P_y) < 2.5 \cdot 2^{i(1-\delta)}$.

Both x_i and y_i are in C_i and $d(x_i, y_i) \le 6 \cdot 2^i$. Without loss of generality, assume that y_i has been added after x_i. Then we added the edge (x_i, y_i) as well. This edge in combination with P_x and P_y give an x-y path P with

$$\tau(P) < 5 \cdot 2^{i(1-\delta)} + 6^{(1-\delta)} 2^{i(1-\delta)} \le 9 \cdot 2^{i(1-\delta)}$$

using the inequality $6^{0.75} < 4$. ■

LEMMA 6.2. *For $\delta = 0.25$, the longest edge a on the fastest path Q is such that*

$$2^i \cdot 9^{-1/\delta} \le d(a) \le 2^i \cdot 9^{1/(1-\delta)}.$$

Proof. To get the upper bound, we observe that the traversal time of a cannot exceed the traversal time of P in Lemma 6.1: $d^{1-\delta}(a) \le 9 \cdot 2^{i(1-\delta)}$, which implies the desired bound.

To get the lower bound, note that the average speed (distance/time) on Q is at most the speed on a, which is $d^{\delta}(a)$. Thus the traversal time of Q is at least 2^i divided by the speed, and applying the lemma again we get

$$\frac{2^i}{d^{\delta}(a)} \le 9 \cdot 2^{i(1-\delta)}.$$

This implies the lower bound. ■

LEMMA 6.3. *For $\delta = 0.25$, the fastest x–y path goes through a vertex $v \in C_k$ with $i - 16 < k < i + 5$.*

Proof. Let v, w be the endpoints of a and without loss of generality assume that v has been added to the graph after w. The edge has been added because $w \in C_k$ for some k, and either $v \in C_k$, in which case $2^k \le d(a) \le 6 \cdot 2^k$, or $v \in C_{k-1}$ and $2^k \le d(a) \le 2^{k+1}$. In both cases, the former bound applies

$$2^k \le d(a) \le 6 \cdot 2^k.$$

We combine these bounds with those of Lemma 6.2 in two ways. First, $2^k \le 2^i \cdot 9^{1/(1-\delta)}$ and therefore $k \le i + \frac{1}{1-\delta} \log 9 < i + 5$. Second, $2^i \cdot 9^{-1/\delta} \le 6 \cdot 2^k$ and therefore $k > i - 3 - (\log 9)/\delta > i - 16$. ■

Theorem 6.1. *For $\delta = 0.25$, a network constructed as above has a traversal time metric whose highway dimension is $\alpha^{O(1)}$.*

Proof. First we bound the number of vertices of C_i in $B_{v,4r}$. The ball can be covered by $\alpha^{\log(8r/2^i)}$ balls of radius 2^{i-1}. Since each of the balls contains at most one vertex of C_i, this gives the desired bound.

Consider shortest paths longer than r in $B_{v,4r}$. By Lemma 6.3, these paths are covered by the intersection of the ball with C_i for $\lfloor \log r \rfloor - 7 \le i \le \lceil \log r \rceil + 9$. The number of relevant covers is constant, and the intersection of each cover with the ball is constant as well. $\blacksquare$

The theorem shows that a fairly simple model can be used to generate networks with constant highway dimension. We do not attempt to model the "Steiner" property of real road networks, where a new vertex may be connected to a point on an existing edge, which corresponds to creating a new intersection on an existing road segment. We also allow adversarial vertex placement, but in real life new vertices are usually added close to existing access points. A more sophisticated model may lead to tighter bounds.

7 Discussion

We have shown that having small highway dimension formally guarantees good query performance for variants of many of the recent shortest-path speedup algorithms (RE, CH, HH, TN, SHARC). No formal performance guarantees had been previously known for these algorithms. Our results shed light on what might be the underlying reason for their remarkably good performance on road networks. We believe our notion of highway dimension may help to further expand the possibilities of future route planning services.

Our definition of highway dimension has been motivated by the good practical performance of recent shortest path algorithms. In particular, the set of transit nodes of [1, 2] is similar to a shortest-path cover: all long enough shortest paths go through a transit node. However, the set is sparse only in a local sense: on average, each vertex has a small number of access nodes, the transit nodes it has to be aware of. It is possible that real road networks have small highway dimension only in a weaker sense, i.e., for some values of r some vertices may have many cover elements nearby, but on average the number of nearby elements is small. Moreover, road networks have other properties that may help to explain the good practical performance of the recent algorithms (beyond what we could prove). For example, they are almost planar and have small separators.

Recall that our SPC algorithm is greedy, always selecting vertices that cover the most shortest paths. As highways are more extensively used in road networks, the algorithm tends to pick highway nodes. The cover it produces may be closer to optimal than our worst-case bound implies.

The following *rest area location problem* is closely related to highway dimension and SPCs. Given a graph with transit times on edges and a parameter T, one would like to find the smallest number of rest areas subject to the following conditions: The rest areas are located at vertices, and each trip of duration T or more along a fastest path passes through at least one rest area. This problem appears to be NP-hard, and a variant of the greedy set-cover algorithm gives an $O(\log n)$ approximation. An interesting open question is whether a better approximation is possible in polynomial time, which may be the case because of a special structure of the sets involved.

An interesting open question is an experimental study of issues related to highway dimension of real road networks. In particular, it would be interesting to measure the worst-case highway dimension as well as the distribution of cover sizes for different r and different balls. Unfortunately the underlying problems are probably NP-hard, and even our polynomial-time approximation algorithm is too slow to be practical for continent-size networks. Therefore such a study will have to include new algorithms or heuristics for bounding the highway dimension and related parameters.

Acknowledgments

We would like to thank Daniel Delling for suggesting the use of highway dimension in the analysis of the SHARC algorithm.

References

[1] H. Bast, S. Funke, and D. Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In C. Demetrescu, A.Ṽ. Goldberg, and D.S̃. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, pages 175–192. AMS, 2009.

[2] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proc. 9th International Workshop on Algorithm Engineering and Experiments*, pages 46–59. SIAM, 2006. Available at http://www.mpi-inf.mpg.de/ bast/tmp/transit.pdf.

[3] R. Bauer and D. Delling. SHARC: Fast and robust unidirectional routing. In *Proc. 10th International Workshop on Algorithm Engineering and Experiments*, pages 13–26, 2008.

[4] R. Bauer, D. Delling, and D. Wagner. Shortest Path Indices: Establishing a Methodology for Shortest-Path Problems. Unpublished manuscript, http://digbib.ubka.uni-karlsruhe.de/volltexte/1000006961, 2009.

[5] E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[6] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

[7] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

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

[9] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[10] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proc. 8th International Workshop on Algorithm Engineering and Experiments*, pages 38–51. SIAM, 2006.

[11] A.Ṽ. Goldberg, H. Kaplan, and R.F̃. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A.Ṽ. Goldberg, and D.S̃. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, pages 93–140. AMS, 2009.

[12] L. Gottlieb and L. Roditty. An optimal dynamic spanner for doubling metric spaces. In *Proc. 16th Annual European Symposium Algorithms*, pages 478–489, 2008.

[13] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. 6th International Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[14] M. Hilger, E. Köhler, R.H̃. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A.Ṽ. Goldberg, and D.S̃. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, pages 73–92. AMS, 2009.

[15] D. Johnson. Approximation algorithms for combinatorial problems. *J. Comp. and Syst. Sci.*, 9:256–278, 1974.

[16] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proc. 32th Annual ACM Symposium on Theory of Computing*, pages 163–170. ACM, 1999.

[17] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *IfGIprints 22, Institut fuer Geoinformatik, Universitaet Muenster (ISBN 3-936616-22-1)*, pages 219–230, 2004.

[18] S. Milgram. The Small World Problem. *Psychology Today*, 1:61–67, 1967.

[19] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proc. 13th Annual European Symposium Algorithms*, pages 568–579, 2005.

[20] P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proc. 14th Annual European Symposium Algorithms*, pages 804–816, 2006.

[21] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[22] M. Thorup and U. Zwick. Approximate distance oracles. *J. Assoc. Comput. Mach.*, 52(1):1–24, 2005.