Approximate shortest paths guided by a small index*

Jörg Derungs, Riko Jacob, and Peter Widmayer

Institute of Theoretical Computer Science, ETH Zurich, Switzerland
ETH Zentrum, CH-8092 Zürich
{joerg.derungs, riko.jacob, peter.widmayer}@inf.ethz.ch

Abstract. Distance oracles and graph spanners are excerpts of a graph that allow to compute approximate shortest paths. Here, we consider the situation where it is possible to access the original graph in addition to the graph excerpt while computing paths. This allows for asymptotically much smaller excerpts than distance oracles or spanners. The quality of an algorithm in this setting is measured by the size of the excerpt (in bits), by how much of the original graph is accessed (in number of edges), and the stretch of the computed path (as the ratio between the length of the path and the distance between its end points). Because these three objectives are conflicting goals, we are interested in a good trade-off. We measure the number of accesses to the graph relative to the number of edges in the computed path.

We present a parametrized construction that, for constant stretches, achieves excerpt sizes and number of accessed edges that are both sublinear in the number of graph vertices. We also show that within these limits, a stretch smaller than 5 cannot be guaranteed.

1 Introduction

We study the problem of answering approximate shortest path queries on an edge weighted, undirected graph $G = (V, E)$ (called the *base graph* from now on) with n vertices and m edges, where the vertices are labeled with unique bit strings of length at most $\lceil \ell \log n \rceil$ for an $\ell \geq 1$. For practical purposes we also assume that all edge weights are *ℓ-limited precision rational numbers* whose numerators and denominators can be stored with $\lceil \ell \log n \rceil$ bits. In our specific setting, we are allowed to preprocess the (very large) base graph and store information about it (viewed as a bit string) in a memory of severely limited size; we call this an *excerpt* of the graph, and the number of bits the excerpt *size.* A path query is specified by the labels of both end vertices; the answer to the query is a path in G, returned as a sequence of vertex labels. No information can be passed from one query to the next. For answering a path query, we may read from the excerpt at no cost. In addition we may access the graph by *probing* a vertex, which *reports*

* This work has been partially supported under the EU programme COST 295 (DYNAMO) and by the Swiss SBF under contract number C05.0047.

a single edge incident to this vertex at unit cost. In a sequence of probes of the same vertex, the incident edges are returned in order of increasing weights, with equal weight edges in the same order within any probe sequence. A vertex can be *reset* at any time to its *start state*, with the effect that probing starts again at the adjacent edge with the smallest weight. When a path query starts to be processed, all vertices are in their start states. The total cost of accessing the graph for a path query (the total *probe cost*) is the number of probes for that query. Because paths with more edges tend to need more probes, we define the efficiency of probing as the ratio of the probe cost against the number of edges in the returned path, the *probe factor*.

We are interested in the trade-off between excerpt size, probe factor, and approximation ratio of the length of the returned path against the shortest path length in the base graph (also known as the *stretch*). While we are interested in the asymptotic limits of the trade-off in arbitrarily large graphs, we still give concrete values for the trade-off parameters and their limits. Our focus is on showing that graph excerpts and path finding algorithms with the desired properties exist. Still, our proofs are constructive, and the presented and implied algorithms all run in polynomial time.

We came across problems of this nature in the study of public key authentication, but we believe our problem is of interest in other domains as well, such as for external memory and caching. In public key authentication, the certificate of a public key can be retrieved from the organization that issued that certificate. These certificates can be interpreted as the edges of a huge implicit graph. An important and potentially time-consuming part of asserting that a key belongs to an alleged owner translates into finding a path in this graph.

Short paths by means of small excerpts have attracted attention earlier: Graph spanners [16] and approximate distance oracles [20] solve the special case where (expressed in our setting) the allowed probe cost for computing a path is zero, that is, the path has to be computed from the excerpt alone, without access to the base graph. Both the graph spanner and the approximate distance oracle allow for a trade-off between the two conflicting goals of keeping excerpts as well as stretch small. For graphs on n vertices, $O(n \log n)$ bits are sufficient to store a $\log n$-spanner. On the other hand, any excerpt from which paths (of arbitrary stretch) need to be computed without further information (such as base graph accesses) requires at least $\Omega(n \log n)$ bits, as shown in Section 3.1. We are therefore interested in trade-offs where the excerpt size is $o(n \log n)$ bits. Without a limit on the probe factor we can easily restrict the excerpt size. Thus, in our trade-off we limit the probe factor to $o(n)$. This does not even allow the path finding algorithm to exlore the whole neighbourhood of the returned path in dense graphs.

To keep our modelling reasonably simple, we do not restrict the path finding algorithm to be time or space efficient. This strengthens the impossibility results of Section 3. Our construction based on the index graph is not exploiting this loophole, it only computes shortest paths on the index graph and on explored subgraphs with r edges. In particular, for the choice of parameters that achieves sublinear index size and sublinear probe factor, this computation time (per edge in the reported graph) and computation space remain sublinear in the number of nodes of the base graph.

1.1 Results

In Section 2, we define a path finding algorithm and a corresponding excerpt, the *index graph*. We give two trade-off parameters for the construction of the index graph. The first, r, limits the probe factor to $O(r)$, the second, σ, limits the stretch to $O(\sigma)$. With these parameters we get an index graph with $\tilde{n} \leq \min\left(\frac{n \log n}{\sqrt{r}}, n\right)$ nodes and $\tilde{m} \leq \tilde{n}^{1+2/\sigma}$ edges, which can be stored with $O(\tilde{m}\ell \log n)$ bits. As mentioned above, we are interested in trade-offs with both excerpt size and probe factor sublinear in the number of vertices. And indeed, for every $\varepsilon, 0 < \varepsilon < \frac{1}{2}$, we can set $r \geq n^{2\varepsilon}$ and $\sigma \geq 2\left(\frac{1-\varepsilon}{\varepsilon}\right)$ to achieve sublinear size and probe factor and still guarantee a constant stretch.

Our construction can be seen as a natural extension of graph spanners and distance oracles. For probe factor 0, it results in a σ-spanner of the base graph.

Beside the index graph construction, we also provide bounds on the trade-off that can be achieved in different situations. Specifically, we show that for a stretch smaller than 5 (Section 3.3), or if the order in which the edges are accessed is independent of their weights (Section 3.2), the probe factor cannot be smaller than $\frac{n}{8}$, even if the excerpt size is in the same order of magnitude as a graph spanner, namely $\frac{n \log n}{8}$. Thus, the desired trade-off with sublinear size and sublinear probe factor cannot be achieved in these cases.

1.2 Related Work

The problem of finding shortest or short paths in a graph received a lot of attention in the last decades, see for example the recent survey by Zwick [21]. The role of limited amount of working storage space has been investigated, among others, for approximate distance oracles, external memory data structures, and the streaming model, as detailed below.

The approximate distance oracle [20] chooses a $k \geq 1$, uses a data structure of size $O(kn^{1+1/k})$ and returns an approximate shortest path in constant time per path edge. The returned path is at most $2k-1$ times larger than the true distance. The construction of the data structure takes $O(kmn^{1/k})$ expected time, which was recently improved to $\tilde{O}(n^2)$ time [4]. Note that the data structures in approximate distance oracles are always of size $\Omega(n \log n)$ bits, whereas our focus are excerpts that use asymptotically less bits than that.

Another possibility is the relabeling of the entire graph in a preprocessing step [7]. There, the two labels of the source and the sink are sufficient to answer (approximate) distance queries. In this setting, the total space usage of the labels is necessarily $\Omega(n\sqrt{m})$ for some graphs, and is always super-linear in the number of vertices.

Along a different line of thought, external memory algorithms and data structures have been designed for shortest path computations, among other graph problems. See [14] for a recent overview and [15] for one of the latest shortest path algorithms. For external memory algorithms, the assumption is that main memory is limited, and external memory accesses to storage blocks are costly. Distance oracles reduce the required size of main memory, but still they are only suited for what is known as semi-external memory graph algorithms, namely situations in which the memory is big enough to store some information about every node of the graph. In this respect, our approach extends the possibilities since it uses even less space. In contrast to our model, the general external memory model has the focus on algorithms for very limited size memory and arbitrarily large graphs, and assumes that the external memory is organized in blocks (cache-lines). Our setting can be understood as a situation where the algorithm is not able to change the representation of the graph in external memory but can extract a small graph excerpt in an expensive preprocessing phase. Then, a probe corresponds to an access to external memory that retrieves the next block of edges. Hence, our results can be understood in the external memory set-up, but are certainly different in focus from the well established external memory algorithms and data structures.

One of the most recent areas with a strong emphasis on storage efficiency is the streaming model of computation, where only little information can be maintained while a long stream of data passes by. While this area started out with simple statistical and aggregate questions, it has matured to include general algorithmic problems, such as computing some graph property when the huge graph is given as a sequence of edges in arbitrary order. Interestingly, for many of these properties, such as connectedness, which is at the basis of short path computations, at least $\Omega(n)$ bits need to be stored if streams can only be read [11]. The *W-Stream* model [8], where intermediate streams are written in one pass to be read in the next pass, allows for a trade-off between memory and number of passes for single source shortest path. Such an intermediate stream can be viewed as an excerpt of the graph. Nevertheless, one cannot apply these concepts in our setting, since in the first pass the whole graph is read, the size of the intermediate streams may be in the order of the number of graph edges, and the content of the intermediate streams depends on the query and not only on the graph.

Exact and approximate shortest path acceleration [9] can be regarded as closely related to our problem. Shortest path acceleration tries to answer several path queries on the same graph efficiently. To that end, the graph is preprocessed and some data structure is stored additionally to the graph. This preprocessing should also be completed with low running time. In contrast to shortest path acceleration, we try to limit the number of graph edges that need to be examined to answer path queries, and we do not, at this point, consider the cost or running time for preprocessing. On the other hand, the data structure generated in the preprocessing of the shortest path acceleration may be as large as the graph itself, whereas in our model the size of this data structure should be really small, i.e., of size $o(n)$. In particular, the techniques used for shortest path acceleration are not directly applicable to our model. For instance, Goldberg and Harrelson [13] use a vertex labelling in the preprocessing to be used for triangulation during path queries. This is not feasible in our model since the vertex labels require $\Omega(n \log n)$ space. Sanders and Schultes [19] present a different approach, so called highway hierarchies. While the space needed to store the additional data is considerably smaller than the one needed by vertex labelling, the authors observe that the space is "a small constant factor of the input size" [19].

2 Index Graph and Path Finding Algorithm

In this section, we present an algorithmic solution for the problem of finding approximate shortest paths guided by a small index, or graph excerpt. We describe mainly the mathematical structure of our excerpt. The algorithm to construct it follows directly from this structure. First we describe roughly the path finding algorithm, and then the undirected weighted *index graph*, the graph excerpt that supports such a path finding algorithm. Details on the path finding algorithm and the index graph are given in Section 2.1 and 2.2, respectively.

Note that because we are interested in trade-offs with sublinear excerpt size, not all vertices are represented in the index graph. Thus, the path finding algorithm computes a path between two vertices s and t in the following way. First, it explores the vicinities of s and t in the base graph to find two *index nodes*, i.e., graph vertices represented in the index graph, that are closest to s and t, respectively. Then it computes a preliminary path in the index graph between the two index nodes. This preliminary path is a path with gaps. Finally, the path finding algorithm closes the gaps by exploring the base graph and thus obtains a path between s and t in the base graph.

The two trade-off parameters, r and σ, are not explicitly used in the path finding algorithm. Instead, the structure of the index graph guarantees that the probe cost for finding an index node in the vicinity of any vertex, and for closing a gap in the preliminary path, are bounded by r resp. $2r$. The index graph also limits the stretch of the obtained path to $9\sigma + 2$ (Lemma 9 and Lemma 3).

The main ingredient of constructing the index graph is the selection of the index nodes. As indicated above, we want enough index nodes such that we can find a shortest path from any vertex in the graph to an index node while accessing at most r edges. On the other hand, the limit of the excerpt size imposes a limit on the number of index nodes. To specify a "good" set of index nodes, we introduce the relation $\mathrm{near}_r(u, v)$ over pairs of vertices that is parametrized with the probe cost limit r. We postpone the technical definition of this r-near relation. For now, it is sufficient that if $\mathrm{near}_r(u, v)$, then with probe cost at most r we can find all shortest paths from u to v and to all vertices closer to u than v. Because every vertex is r-near to at least $\sqrt{r}$ other vertices (Lemma 5, page 8), we can compute a set of at most $\frac{n \log n}{\sqrt{r}}$ index nodes in polynomial time (Lemma 7) such that every vertex is r-near an index node. In a similar way as the *neighbourhood* of a vertex v is the set of vertices to which there is an edge from v, we call the set of vertices to which v is r-near the *vicinity* of v; the vertex v is the *center* of the vicinity of v. Note that every vertex is in its own vicinity.

The r-near relation is not only useful to select the index nodes, but also to construct the *index edges*, which represent paths in the base graph. We connect two index nodes v_1 and v_2 if there is an edge $\{u_1, p_2\}$ in the base graph such that a vertex p_1 is r-near v_1 and u_1, and p_2 is r-near v_2. Exploring from the vertices p_1 and p_2, the path finding algorithm can close the gap between v_1 and u_1 and between v_2 and p_2 with probe cost at most $2r$. The weight of the edge is the length of the path from v_1 to v_2 that results from closing the gaps, and the edge is annotated with the vertices p_1, u_1, and p_2. If the path from v_1 to v_2 has less than 6 edges, we annotate the index edge with the whole path. Note that p_1 may be the same vertex as u_1. In particular, with an edge $\{v_1, v_2\}$ in the base graph, $p_1 = u_1 = v_1$ and $p_2 = v_2$.

The limit on the probe cost is not the only feature based on these index edges. Lemma 9 shows that the distance between two index nodes in the *full index graph*, i.e., the graph induced by all index edges, is at most 3 times their distance in the base graph.

Even if we eliminate multiple edges between index nodes, the full index graph may still be arbitrarily dense, causing a large excerpt. Therefore, our *index graph* $I_r^\sigma(G)$ of a base graph G is a greedy σ-spanner [16] of the full index graph.

With Definition 4 (page 8) of the r-near relation, which is based on the exploration of a shortest path tree, we can give the following upper bounds for the size of the index graph and the performance of the path finding algorithm:

Theorem 1. *For all graphs $G = (V, E)$ on n vertices with unique $\lceil \ell \log n \rceil$ bit vertex labels and ℓ-limited precision rational edge weights, for all integer values $r \leq n$ and $\sigma, 1 \leq \sigma \leq \log n$, the index graph $I_r^\sigma(G)$ and the corresponding path finding algorithm $\mathcal{A}$ with input $(s, t, I_r^\sigma(G))$ that reports a path $\mathrm{p}(s,t)$ from any vertex s to any vertex t in G have the following properties:*

1. *the index graph $I_r^\sigma(G)$ has $\tilde{n} \leq \min\left(\frac{n \log n}{\sqrt{r}}, n\right)$ nodes and $\tilde{m} \leq \tilde{n}^{1+2/\sigma}$ edges and can be stored with $O(\tilde{m}\ell \log n)$ bits*
2. *the length of the path $\mathrm{p}(s,t)$ reported by the algorithm $\mathcal{A}$ is less than $(9\sigma + 2) \cdot \mathrm{dist}(s,t)$*
3. *the probe cost of the algorithm $\mathcal{A}$ to compute path $\mathrm{p}(s,t)$ is at most $2r$ if $\mathrm{p}(s,t)$ has less than 6 edges; the probe factor is at most $r/2$ if $\mathrm{p}(s,t)$ has at least 6 edges.*

Note that for graphs with positive integer edge weights, the probe factor can also be limited relative to the length of the returned path.

The remainder of this section gives the details of the path finding algorithm and the index graph construction, and defines the r-near relation. Theorem 1 follows directly from these details.

2.1 Path Finding Algorithm

In this section, we formalize the details of the path finding algorithm and present the lemmas used to prove Points 2 and 3 of Theorem 1.

To find a path from a vertex s to a vertex t, the algorithm starts exploring from s and t until an index node $\tilde{s}$ and $\tilde{t}$ is reached, respectively. Note that if $\mathrm{near}_r(s,t)$ or $\mathrm{near}_r(t,s)$, then the algorithm has already found a shortest path from s to t. Next, the algorithm computes a shortest path from $\tilde{s}$ to $\tilde{t}$ in the index graph. For every index edge on that path, the corresponding path in the base graph is computed by exploring from the annotated vertices p_1 and p_2, if it cannot be read directly from the index edge.

We use the term *expanding an index edge* for computing a path in the base graph that connects the two index nodes incident to the index edge. Lemma 2, the limit for the probe factor, is based on the observation that if an index edge is expanded, the corresponding part of the returned path has at least 6 edges.

Lemma 2 (without proof). *The probe factor of the path finding algorithm to compute a path p between two graph vertices is at most $r/2$ if p has at least 6 edges; the total probe cost is at most $2r$ if p has less than 6 edges.*

The following lemma is based on the observation that if one end point of the path query is closer to the other end point than to an index node, the path finding algorithm will find and return a shortest path.

Lemma 3 (without proof). *If the distance between two index nodes in the index graph $I_r^\sigma(G)$ is at most k times their distance in the base graph, then the length of any path returned by the path finding algorithm is at most $3k + 2$ times the distance between the two end points.*

It remains to define an r-near relation that allows the path finding algorithm to find shortest paths from v to all vertices in the vicinity of v, while accessing at most r graph edges. The properties of r-near suggest that we define r-near based on the shortest path tree, which can be computed with Dijkstra's shortest path algorithm [10]. Note that the shortest path algorithm operates on directed graphs. This serves our purpose, since any edge $\{u, v\}$ is reported both by a probe of u and by a probe of v.

However, the shortest path algorithm in its classical formulation examines all arcs of one vertex at once. In a high degree graph, this can induce probe costs higher than any $r \in o(n)$. Therefore, we modify Dijkstra's algorithm slightly to evaluate edges lazily, that is, to only read those arcs that are really needed to determine the next vertex to be added to the shortest path tree. Similar methods to speed up Dijksta's shortest path algorithm by limiting the number of edges in the priority queue have been proposed before, see [1] for an example. The main goal in our case, however, is to limit the number of examined edges. For every vertex u we only need one out-going arc $(u \to v)$ in the priority queue at a time. Remember that we access the edges incident to one vertex in the order of increasing weights. Thus, examining and possibly adding an arc $(u \to w)$ with greater or equal weight can be postponed until $(u \to v)$ is removed from the queue. Specifically, when a vertex u (including the root) is added to the shortest path tree, we add only the first arc reported by a probe of u to the priority queue. Whenever an arc $(u \to v)$ is removed from the queue, we add the next

Algorithm 1 Shortest path with lazy edge evaluation

```
procedure EXPLORE(vertex source, vertex dest)
    add first outgoing arc of source to priority queue
    while dest not in shortest path tree do
        $(u \to v) \leftarrow$ arc on top of priority queue
        remove $(u \to v)$ from priority queue
        if $v$ not in shortest path tree then
            add $v$ to shortest path tree with edge $\{u, v\}$
            quit if $v = dest$
            add first outgoing edge of $v$ to priority queue
        end if
        add next outgoing edge of $u$ to priority queue
    end while
end procedure
```

arc reported by a probe of u. See Algorithm 1 for a formal description of the modified shortest path algorithm.

We want the vertices to be added to the shortest path tree in a deterministic order, for situations in which many vertices have the same distance to the root of the tree, and we want the root to be r-near some but not all of them. For simplicity, we use the label of the arc's target vertex as a tie-breaker, even though using a deterministic priority queue would suffice.

From here on, `Explore(u,v)` refers to the modified shortest path algorithm with source u and destination v. The definition of r-near naturally follows from `Explore`.

Definition 4. *A vertex u is r-near another vertex v,* $\mathrm{near}_r(u, v)$*, if and only if* `Explore(u,v)` *adds at most r edges to the priority queue.*

Based on this definition, we can bound the size of the vicinity of any vertex.

Lemma 5. *A vertex is r-near at least $\lceil\sqrt{r}\rceil$ vertices, including itself.*

Proof. Assume that `Explore(u,v)` is about to add v as the $(k + 1)$st node to the shortest path tree. At that point, at most k^2 arcs can have been added to the priority queue: $k(k - 1)$ arcs between tree nodes, and k arcs from tree nodes to vertices that are not in the tree yet (including v). For $k = \lceil\sqrt{r}\rceil - 1 < \sqrt{r}$ we get $k^2 < r$ and therefore $\mathrm{near}_r(u, v)$. □

Lemma 6. *A vertex is r-near at most $r + 1$ vertices, including itself.*

Proof. For every edge that is added to the priority queue in `Explore`, at most one vertex can be added to the shortest path tree. □

2.2 Index Graph

In this section, we describe the details of the index graph construction, and present the lemmas used to prove Point 1 of Theorem 1.

As outlined at the beginning of Section 2, we first select the index nodes, then compute the index edges, and lastly build the index graph $I_r^\sigma(G)$ as a greedy σ-spanner [16] of the full index graph induced by the index edges.

Lemma 7. *In every graph G on n vertices there is a set $\tilde{V}$ of at most $\min\left(\frac{n \log n}{\sqrt{r}}, n\right)$ vertices such that every vertex in G is r-near a vertex in $\tilde{V}$. The set $\tilde{V}$ can be computed in $O(rn \log n)$ time.*

Proof. The size limit n is trivial because the set $\tilde{V}$ cannot contain more vertices than the graph G, and every vertex is r-near itself.

The set $\tilde{V}$ of vertices can be interpreted as a Set Cover [12] solution. The graph vertices form the universe of the Set Cover instance, and for every vertex v there is a set containing all vertices that are r-near to v. This Set Cover instance has n elements and n sets. Every element is in at least $\sqrt{r}$ sets because every vertex is r-near at least $\sqrt{r}$ vertices (Lemma 5). Thus, with Lemma 8, Greedy Set Cover [6] computes a set $\tilde{V}$ with at most $(\log_2(\sqrt{r}) + 1)\,\frac{n}{\sqrt{r}} \leq \frac{n \log n}{\sqrt{r}}$ vertices.

The sets of the Set Cover instance can be computed in $O(r \log r)$ per vertex using `Explore`. The upper bound of $r + 1$ for the sets (Lemma 6) limits the running time of Greedy Set Cover to $O(rn \log n)$. $\square$

Lemma 8 (Alon, Spencer [2]). *Given a universe U with at most n elements and at most n subsets of U. If every element is in at least k sets, then the number of sets selected by the Greedy Set Cover algorithm [6] is at most $(\log_2(k) + 1)\,\frac{n}{k}$.*

This concludes the selection of the index nodes. The structure of the index edges is already described at the beginning of Section 2. The index edges can be computed efficiently by considering every vertex as point p_1 and creating an index edge for every edge $\{u_1, p_2\}$ with $\mathrm{near}_r(p_1, u_1)$. Lemma 9 limits the distances in the full index graph.

Lemma 9. *The distance between any two index nodes in the* full index graph *induced by the set of all index edges is at most three times their distance in the base graph.*

Proof. Let p be a shortest path in the base graph from index node $\tilde{s}$ to index node $\tilde{t}$. We use a sequence of vertices p_i on the path p, with increasing distance from $\tilde{s}$, to split p into several parts, such that for each part there is an index edge. These index edges will form a path from $\tilde{s}$ to $\tilde{t}$.

For every p_i, let $\tilde{v}_i$ be the closest index node and d_i be the distance from p_i to $\tilde{v}_i$. Note that p_i is r-near $\tilde{v}_i$. As p_1 we take any vertex on p for which $\tilde{v}_1 = \tilde{s}$. The last vertex p_k of the sequence has $\tilde{v}_k = \tilde{t}$. There must be such a p_k because $d_i \leq \mathrm{dist}(p_i, \tilde{t})$ for every p_i.

For a given vertex p_i, we set p_{i+1} as the first vertex on path p with $\mathrm{dist}(p_i, p_{i+1}) \geq d_i$. Since the left neighbour u_i of p_{i+1} on the shortest path is r-near p_i, there is an index edge $\{\tilde{v}_i, \tilde{v}_{i+1}\}$ with weight at most $2\,\mathrm{dist}(p_i, p_{i+1}) + d_{i+1}$. These index edges form a path from $\tilde{s}$ to $\tilde{t}$. The weight of this path is at most $\sum_{i<k}(2\,\mathrm{dist}(p_i, p_{i+1}) + d_{i+1}) \leq d_k + \sum_{i<k} 3\,\mathrm{dist}(p_i, p_{i+1}) \leq 3\,\mathrm{dist}(\tilde{s}, \tilde{t})$.

Note that it does not matter if two index nodes $\tilde{v_i}$ and $\tilde{v_j}$ are actually the same vertex, since this only shortens the length of the path in the full index graph. □

To compute the index graph $I_r^\sigma(G)$ as a σ-spanner of the full index graph, we use the *Greedy σ-Spanner* algorithm [3]. With this algorithm, we can compute the index graph from the index edges in polynomial time. The Greedy σ-Spanner also guarantees an upper bound on the edges in the index graph:

Lemma 10 (Regev [18], combined with Bollobás [5]). *The Greedy σ-Spanner of every graph on $\tilde{n}$ vertices has at most $\tilde{n}^{1+2/\sigma}$ edges.*

3 Impossibility Results for Trade-Offs

In this section, we show various impossibility results for trade-offs between excerpt size, stretch, and probe factor.

Throughout this section, we allow all operations on vertex labels. This is in contrast to the path finding algorithm and the index graph construction presented in Section 2, where we only use comparisons. Additionally, for all impossibility results presented in this section, we restrict the vertex labels to integers from 1 to n on graphs with n vertices. There is also no assumption on the form or content of the graph excerpt, except that it can be stored as a bit string.

We use the same technique to prove all three limits presented in this section. For each limit, we present a family of graphs such that for every pair of graphs in the family, there is at least one pair of vertices for which the connecting paths that may be returned by the path finding algorithm differ. Additionally, the probe cost for distinguishing two graphs of the family is above the allowed limit of the probe cost (which is zero in Section 3.1).

3.1 Lower Bound for Spanners and Oracles

In this section we show that if the path finding algorithm is not allowed to access the base graph, then the size of the excerpt needs to be in $\Omega(n \log n)$.

Lemma 11. *There is no path finding algorithm $\mathcal{A}(s, t, \{0,1\}^L)$, with $L \leq \frac{n \log n}{2}$, that returns a path from vertex s to vertex t for all graphs G on n vertices with integer labels 1 to n.*

Proof. Consider the family of trees with unique vertex labels 1 to n. There are $n^{n-2} > 2^{\frac{n \log n}{2}}$ different trees on the vertices with unique labels 1 through n [17]. Therefore, with $L \leq \frac{n \log n}{2}$, at least two different trees T and T' are represented by the same bit string of length L. However, there is one pair of vertices s and t such that the path from s to t is different in the two trees. Otherwise, all edges in T must also be in T', and the two trees are identical. Hence, it is not possible for the algorithm $\mathcal{A}$ to decide which path to return without accessing the base graph. □

3.2 Edge Orders

In this section we demonstrate that the order in which the edges are reported is important for achieving good trade-offs between excerpt size, number of accessed edges and stretch. More precisely, we show that if the order in which the edges adjacent to one vertex are reported is based on a criterion that is independent of the edge weights, then the stretch cannot be limited to a value independent of the base graph's edge weights, even if the excerpt size is in the same order of magnitude as a $\log n$-spanner and the probe factor is linear in the number of vertices. We denote the combination of the orders in which the edges adjacent to the individual vertices are reported as an *edge order*.

Lemma 12. *For every edge order of the complete graph G_n on n vertices and every positive integer k, there is a family of weight functions $\mathcal{W} : V \times V \to \mathbb{R}^+$ such that there is no path finding algorithm $\mathcal{A}(s, t, \{0, 1\}^L)$, with $L \leq \frac{n \log n}{8}$, that returns a path $\mathrm{p}(s, t)$ with stretch k and probe factor $\frac{n}{8}$ for every vertex pain s, t and every weight function in $\mathcal{W}$.*

Proof Idea: We divide the vertices into two sets A and B, with $|A| = n/4$. Which vertices are in A depends on the edge order. The set B forms a complete subgraph with all edge weights 1. Every vertex in A has exactly one edge to a vertex in B with weight 1, all other edges have weight $2k + 1$. Which edge has weight 1 depends on the specific weight function. The distance between a vertex a in A and a vertex b in B is thus at most 2, and any path from a to b with stretch k must start with the one edge adjacent to a that has weight 1. However, there are more than $2^{\frac{n \log n}{8}}$ such weight functions for which the weights of the first $n/8$ edges adjacent to every vertex do not differ, which makes it impossible to distinguish any two graphs with probe factor $n/8$. $\square$

3.3 Stretch Limit

In this section we argue for finding only approximate shortest paths instead of true shortest paths, although the base graph can be accessed while computing a path. More precisely, we show that even with excerpts whose size is in the same order of magnitude as a $\log n$-spanner, and with a linear probe factor, no stretch smaller than 5 can be achieved.

Lemma 13. *For every $\varepsilon > 0$ there is a family $\mathcal{G}_n^\varepsilon$ of graphs on n vertices with integer labels 1 to n, such that there is no path finding algorithm $\mathcal{A}(s, t, \{0, 1\}^L)$, with $L \leq \frac{n \log n}{8}$, that returns a path from s to t with stretch $5 - \varepsilon$ and probe factor $n/8$ for every vertex pair s, t in every graph $G \in \mathcal{G}_n^\varepsilon$.*

Proof Idea: We divide the graph vertices into four equally sized sets A, B, C, and D. Sets A and B form a complete bipartite graph with edge weights $1 - \frac{\varepsilon}{8}$, as do sets C and D. Additionally, there is a perfect matching with edge weights 1 between the vertices of A and the vertices of D. The difference between the graphs in $\mathcal{G}_n$ lies only in the matching. The only path between two vertices $a \in A$

and $d \in D$ with stretch $5 - \varepsilon$ is $\{a, d\}$ if that edge exists in the graph. However, there are more than $2^{\frac{n \log n}{8}}$ such matchings, and the probe cost for finding out if $\{a, d\}$ is in the graph is larger than $\frac{n}{4}$. $\square$

References

1. L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM*, 52(1):25–53, 2005.
2. N. Alon and J. Spencer. *The Probabilistic Method.* John Wiley, 1992.
3. I. Althöfer, G. Das, D. P. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *SWAT '90*, pages 26–37, 1990.
4. S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $\tilde{O}(n^2)$ time. In *SODA '04*, pages 271–280, 2004.
5. B. Bollobás. *Extremal Graph Theory.* Academic Press, 1978.
6. V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.
7. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.
8. C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *SODA '06*, pages 714–723, 2006.
9. C. Demetrescu, A. Goldberg, and D. Johnson, editors. *9th DIMACS Challenge on Shortest Paths*, to appear, available at http://www.dis.uniroma1.it/∼challenge9.
10. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
11. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *SODA '05*, pages 745–754, 2005.
12. M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
13. A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA '05*, volume 16, pages 156–165, 2005.
14. I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies*, pages 62–84, 2002.
15. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *ESA '03*, pages 434–445, 2003.
16. D. Peleg and A. A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.
17. H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, 27:742–744, 1918.
18. H. Regev. The weight of the greedy graph spanner. Technical Report CS95-22, Weizmann Institute Of Science, July 1995.
19. P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA '05*, pages 568–579, 2005.
20. M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.
21. U. Zwick. Exact and approximate distances in graphs - a survey. In *ESA '01*, pages 33–48, 2001.