Bounded-Leg Distance and Reachability Oracles

Ran Duan*
The University of Michigan

Seth Pettie†
The University of Michigan

Abstract

In a weighted, directed graph an L-bounded leg path is one whose constituent edges have length at most L. For any *fixed* L, computing L-bounded leg shortest paths is just as easy as the standard shortest path algorithm. In this paper we study *approximate distance oracles* (and *reachability* oracles) for bounded leg path problems, where the leg bound L is not known in advance, but forms part of the query. Bounded-leg path problems are more complicated than standard shortest path problems because the number of distinct shortest paths between two vertices (over all leg bounds) could be as large as the number of edges in the graph.

The bounded leg constraint models situations where there is some limited resource that must be spent when traversing an edge. For example, the size of a fuel tank or the life of a battery places a hard limit on how far a vehicle can travel in one leg before refueling or recharging. Someone making a long road trip may place a hard limit on how many hours they are willing to drive in any one day.

Our main result is a nearly optimal algorithm for preprocessing a directed graph in order to answer *approximate* bounded leg distance and bounded leg shortest path queries. In particular, we can preprocess any graph in $\tilde{O}(n^3)$ time, producing a data structure with size $\tilde{O}(n^2)$ that answers $(1+\epsilon)$-approximate bounded leg distance queries in $O(\log\log n)$ time. If the corresponding $(1+\epsilon)$-approximate shortest path has l edges it can be returned in $O(l\log\log n)$ time. These bounds are all within polylog(n) factors of the best standard all-pairs shortest path algorithm and improve substantially the previous best bounded leg shortest path algorithm, whose preprocessing time and space are $O(n^4)$ and $\tilde{O}(n^{2.5})$.

We also consider bounded leg oracles in other situations. In the context of planar directed graphs we give a time-space tradeoff for answering bounded leg *reachability* queries. For any $k \geq 2$ we can build a data structure with size $O(kn^{1+1/k})$ that answers reachability queries in time $\tilde{O}(n^{\frac{k-1}{2k}})$.

1 Introduction

In real networks the notion of an *optimal* path may not be absolute, but depend on multiple factors or different resource constraints. For example, one may want to plan a road trip that balances many factors, such as overall travel time, scenicness, and the travel time of the longest day. In this paper we consider a simple generalization of the shortest path problem that constrains paths to be composed of *short* legs. This models situations where we are, for example, constrained by the size of our vehicle's gas tank or the life of its battery.

Formally, our input is a weighted directed graph $G = (V, E, w)$, where $|V| = n, |E| = m$, and $w : E \to \mathbb{R}^+$. An L-bounded leg shortest path is a shortest path in the graph restricted to edges with length at most L. If we wanted to compute point-to-point or all-pairs shortest paths and L is known the problem would be very simple: just discard all unavailable edges and solve the problem as usual. We consider the more realistic situation where the graph G is fixed and L-bounded leg distance/shortest path queries must be answered online. In other words, we need a data structure that can answer queries for *any* given leg bound L. Our goals are to minimize the construction time of the data structure, its space, its query time, and the *quality* of the estimates returned. We say that a distance estimate is α-approximate if it is within a factor of α of the actual distance.

The bounded leg shortest path problem (BLSP) was studied most recently by Roditty and Segal [14]. (See also [3].) They showed that an $\tilde{O}(n^{2.5})$-space data structure could be built in $O(n^4)$ time that answers $(1+\epsilon)$-approximate bounded leg shortest path queries. They also showed that when the graph is induced by points in a d-dimensional l_p metric that a more time and space-efficient data structure could be built for answering $(1+\epsilon)$-approximate BLSP queries. Specifically, the construction time and space are $O(n^3(\log^3 n + \epsilon^{-d}\log^2 n))$ and $O(n^2\epsilon^{-1}\log n)$, respectively. Roditty and Segal's construction made use of complicated algorithms for computing sparse geometric spanners.

Our primary result is a new, efficiently constructible $(1+\epsilon)$-approximate BLSP data structure for arbitrary directed graphs. The construction time and space of our data structure improve significantly on Roditty and Segal's for arbitrary directed graphs and basically match the time and space usage of their structure for l_p^d metrics. In $O(n^3\epsilon^{-1}\log^3 n)$ time we can build a $O(n^2\epsilon^{-1}\log n)$-space data structure that answers distance queries in $O(\log(\epsilon^{-1}\log n))$ time and BLSP queries in $O(\log(\epsilon^{-1}\log n))$ per edge. This shaves

*Email:duanran@umich.edu
†Email:pettie@umich.edu

a factor of n off the construction time and $\sqrt{n}$ off the space of [14]. Since our data structure works for any directed graph it can be used on one induced by an l_p^d metric; unlike [14] the running time of our algorithm has no dependence on the dimension d. One of the main advantages of our algorithm is its simplicity. It is based on a generalized version of the Floyd-Warshall algorithm and retains its streamlined efficiency.

One can obviously consider the effect of bounded leg constraints on nearly any graph optimization problem. In this paper we consider two such problems. We show how to quickly anwwer bounded leg reachability queries in weighted *planar* graphs, where there is an adjustable tradeoff between space usage and query time. In particular, for any $k > 1$, $O(kn^{1+1/k})$ space suffices to answer queries in $\tilde{O}(n^{\frac{k-1}{2k}})$ time. Thorup [18] showed that standard reachability queries in an unweighted planar digraph can be answered in $O(1)$ time with an $\tilde{O}(n)$ space data structure. However, getting a comparable result in the bounded leg scenario seems much more difficult. A key idea in Thorup's data structure is to divide the graph up with path separators. However, in the L-bounded leg case a path only exists (as far as a reachability query is concerned) if all its edges have length at most L. Our algorithm is based on a new data structure for answering bounded leg reachability queries in subgraphs with a non-crossing *Monge* property.

In the full version of this paper we show how to build an approximate distance oracle that can answer bounded leg distance queries accurate to within a $2k - 1 + \epsilon$ factor. The space and query time of our oracles, $\tilde{O}(n^{1+1/k})$ and $\tilde{O}(1)$, match those of Thorup and Zwick [19] (without the bounded leg constraint) up to polylogarithmic factors.

Related Work. Several approximate distance oracles (without extra constraints) have been invented in the last few years, for general graphs [19, 15, 2, 1, 11] as well as planar graphs [18, 9, 10]. A concept closely related to bounded leg shortest paths is that of bottleneck shortest paths. Gabow and Tarjan [7] give a nearly optimal algorithm for arbitrary sparse graphs. Recent work has focused on sub-cubic bottleneck shortest path algorithms using fast matrix multiplication [20, 4].

A natural way to view the bounded leg shortest path problem is as a partially dynamic persistent data structuring problem. (The graph is constructed incrementally, one edge at a time. A query is simply asking a question about a previous version of the graph.) There is a large body of work on dynamic shortest path and reachability data structures; see [13, 16, 5] for recent work and more references.

2 All-Pairs Bounded Leg Distances

Let $G = (V, E)$ be a directed graph with a length function $w : E \to \mathbb{R}^+$. Our aim is to construct a table such that for every ordered pair of vertices (u, v) in V and any positive real number L, we can obtain a $(1 + \epsilon)$-approximate L-bounded leg distance immediately. Denote the L-bounded leg distance between $u, v \in V$ by $\delta^L(u, v)$. We say y is a $(1 + \epsilon)$-approximation of x if $x \leq y \leq (1 + \epsilon)x$.

Let $E_0 = (e_1, e_2, ..., e_m)$ be the list of edges in increasing order. Let $G_i = (V, E^{[1,i]})$, where $E^{[x,y]} = \{e_x, e_{x+1}, ..., e_y\}$, and abbreviate $\delta_{G_i}(u, v)$ by $\delta_i(u, v)$.

In this paper, v is reachable from u in a graph G is represented by $u \overset{G}{\to} v$, and v is not reachable from u in G by $u \overset{G}{\not\to} v$. The *bottleneck distance* from u to v is defined by

$$L(u, v) = \min\{w(e_i) | u \overset{G_i}{\to} v\}$$

If $u \overset{G}{\not\to} v$, then define $L(u, v) = \infty$.

As the leg bound L increases the set of usable edges grows. Therefore, the length of the shortest path from u to v in this insert-only dynamic subgraph can only decrease. When $L \geq w(e_m)$, the subgraph becomes the entire graph G. We can see that all edges in the path from u to v under leg bound $L(u, v)$ are no longer than $L(u, v)$, so $\delta^{L(u,v)}(u, v) \leq (n-1)L(u, v)$. Any path from u to v in G must contain an edge no shorter than $L(u, v)$, so $\delta_G(u, v) \geq L(u, v)$. Thus, we only need $\log_{1+\epsilon}(n - 1)$ different distances for each pair of vertices to be able to return a $(1 + \epsilon)$-approximate distance under any leg bound. The main problem is how to construct this set of distances efficiently. An obvious solution is to insert one edge at a time, then check in $O(1)$ time for every pair of vertices whether its distance changes. The total time for this trivial algorithm is $O(mn^2)$. We will use a natural divide-and-conquer method to reduce the running time to $O(\frac{1}{\epsilon} n^3 \log^3 n)$.

Our aim is to construct, for every pair of vertices (u, v), a set of bounded leg distance entries: $D(u, v) = \{(L_1, d^{L_1}(u, v)), (L_2, d^{L_2}(u, v)), ..., (L_k, d^{L_k}(u, v))\}$, where $L(u, v) = L_1 < L_2 < ... < L_k = w(e_m)$ and $d^{L_i}(u, v)$ is an *approximation* of the distance from u to v under leg bound L_i. For any leg bound L, the distance between u and v should be $(1+\epsilon)$-approximated by some $d^{L_i}(u, v) \in D(u, v)$ where L_i is the maximum among those $L_i \leq L$. Denote this by $d^L(u, v) = d^{L_i}(u, v)$. If $L < L_1$, $d^L(u, v) = \infty$. Moreover, for every (u, v), we guarantee that $|D(u, v)| \leq 2 \log_{1+\epsilon} n$, so for any given leg bound L we can find $d^L(u, v)$ in $O(\log \log_{1+\epsilon} n)$ time.

What about exact BLSP? Before delving into the details of our algorithm we first address a natural

question, which is whether the ϵ in our approximation is truly necessary and whether the dependence on ϵ in our space bound is optimal. We argue below that any BLSP data structure that performs *no* arithmetical operations in the course of a query cannot improve our space bound (in terms of ϵ) and must use space $\Theta(n^4)$ if exact distances are to be reported.

Since there can be $\Theta(n^2)$ different edges in the graph, the distance between any pair of vertices can change at most $O(n^2)$ times, that is, at most $O(n^4)$ different bounded leg distances are needed if our data structure must store every distance that it could return. However, it is not clear whether such a graph exists. In fact, if there is a graph H in which there exists a pair of vertices (u, v) having $\Theta(n^2)$ different bounded leg distances, then we can add $2n$ vertices in H: $\{u_1, u_2, \ldots, u_n\}$ and $\{v_1, v_2, \ldots, v_n\}$, and also directed edges with different lengths $\{(u_1, u), (u_2, u), \ldots, (u_n, u)\}$ and $\{(v, v_1), (v, v_2), \ldots, (v, v_n)\}$. Then in this extended graph H', there are $3n$ vertices, and for any pair of u_i and v_j, their distance varies $\Theta(n^2)$ times when leg bound increases, so in total there are $\Theta(n^4)$ different bounded leg distances.

Now consider the following directed graph $H = (V, E)$: $V = \{u = a_1, a_2, \ldots, a_k = b_0, b_1, b_2, \ldots, b_k, v\}$, and: $E = \{(a_i, a_{i+1}) | 1 \le i \le k-1, w(a_i, a_{i+1}) = 4k\} \cup \{(b_i, v) | 0 \le i \le k, w(b_i, v) = 2k + 1 - 2i\} \cup \{(a_i, b_j) | 1 \le i \le k-1, 1 \le j \le k, w(a_i, b_j) = k^2 - ik + 3k + j\}$. It is a good exercise to show that the distance from u to v varies $\Theta(k^2)$ times, thus there exist graphs with $\Theta(n^4)$ different bounded leg distances. Assuming addition and subtraction are not used during a query, this implies that $\Theta(n^4)$ space is needed to answer exact bounded leg queries and $\Theta(n^2 \epsilon^{-1} \log n)$ space is needed to answer $(1+\epsilon)$-approximate queries.

2.1 A Binary Partition Algorithm. The high-level idea of our algorithm is to find a small set of distances ($O(\log_{1+\epsilon} n)$ per vertex pair) that can $(1+\epsilon)$-approximate any L-bounded leg distance. Suppose that we have just found a reasonably accurate estimate to the distances in G_i and G_j respectively, $i < j$. Call these estimates d_i and d_j. If $d_i(u, v)/d_j(u, v)$ is sufficiently close to 1 then $d_i(u, v)$ can be considered a good-enough estimate of $\delta_{i'}(u, v)$, for all $i < i' < j$. Thus, we can focus on vertex pairs, call them P, whose distance drops significantly between G_i and G_j. Our idea is to compute a reasonably good estimate of the distances of the median $G_{(i+j)/2}$ using a version of the Floyd-Warshall algorithm (Figure 1) that just considers the pairs P. The correctness and time complexity of our algorithm will follow from two lemmas. The first says, essentially,

```
Modified-Floyd(d, P)
    d: an n × n matrix
    P: a set of vertex pairs

    for k = 1 to n do
        for all (s, t) in P do
            d[s, t] ← min{d[s, t], d[s, v_k] + d[v_k, t]}
    return d
```

Figure 1: Modified-Floyd Algorithm: As inputs, d is a matrix that contains the approximate distances for all pairs except the pairs in P. The algorithm returns the approximate distance matrix d.

that if the *Modified-Floyd* algorithm starts off with a good approximation to the distances on all vertex pairs besides P, it ends with a good approximation for all vertex pairs, including P. One problem in our divide-and-conquer approach is that errors accumulate as we break the problem into smaller pieces. The second lemma bounds the growth of these errors.

LEMMA 2.1. *Let $G' = (V', E')$ be a graph, let $P \subseteq V' \times V'$ be a set of pairs of vertices. If initially for all $(s, t) \in (V' \times V') \setminus P$, $d(s, t)$ is an α-approximation of $\delta(s, t)$, and for all $(s, t) \in P \cap E'$, $\delta(s, t) \le d(s, t) \le w(s, t)$, then the matrix d returned by this Modified-Floyd procedure satisfies: for any pair $(s, t) \in P$, $d(s, t)$ is an α-approximation of $\delta(s, t)$.*

Proof. Notice that this algorithm can never underestimate a distance $\delta(s, t)$ if there are no underestimates originally. Denote the real shortest path from s to t in G' by $s \rightharpoonup t$. For any $(s, t) \in P$, if the shortest path $s \rightharpoonup t$ is composed of only one edge, then $(s, t) \in E'$ and $\delta(s, t) = w(s, t) = d(s, t)$, so this case is trivial. Now assume that after k rounds ($k \ge 1$), for every pair of vertices $(s, t) \in P$ such that $s \rightharpoonup t$ includes only intermediate vertices from $\{v_1, \ldots, v_k\}$, $d(s, t)$ is an α-approximation of $\delta(s, t)$. In the $(k+1)$th round, if $k+1$ is the index of the highest intermediate vertex in $s \rightharpoonup t$, for $(s, t) \in P$, then the highest indices in the paths $s \rightharpoonup v_{k+1}$ and $v_{k+1} \rightharpoonup t$ are both at most k. So, by the inductive hypothesis, $d(s, v_{k+1})$ and $d(v_{k+1}, t)$ are already α-approximations of $\delta(s, v_{k+1})$ and $\delta(v_{k+1}, t)$ respectively. Therefore, after the $(k+1)$st round, $d(s, t) \le d(s, v_{k+1}) + d(v_{k+1}, t) \le \alpha\delta(s, v_{k+1}) + \alpha\delta(v_{k+1}, t) = \alpha\delta(s, t)$, so $d(s, t)$ is also an α-approximation of $\delta(s, t)$.

Suppose that we have a pretty good approximation to the distances in G_i and G_j. We want to find an approximation to the distances in G_q, where $q = \lfloor (i+j)/2 \rfloor$. If the distances of some pairs change slightly

between G_i and G_j, then we can just use their distances in G_i to estimate their distance in G_q. We can focus our attention on the pairs whose distance changes a lot between G_i and G_j.

LEMMA 2.2. *Let d_i and d_j be α^l-approximations of δ_i and δ_j, where $i < j$. Then we can find an α^{l+1}-approximation of δ_q, where $q = \lfloor (i+j)/2 \rfloor$, in $O(n|P| + j - i)$ time, where $P = \{(s,t) \mid (s,t) \in V \times V$ and $\frac{d_i(s,t)}{d_j(s,t)} > \alpha\}$.*

Proof. By definition: for all $(s,t) \in V \times V$, we have $\delta_i(s,t) \leq d_i(s,t) \leq \alpha^l \delta_i(s,t)$ and $\delta_j(s,t) \leq d_j(s,t) \leq \alpha^l \delta_j(s,t)$. Because for all $(s,t) \notin P$, $d_i(s,t) \leq \alpha d_j(s,t)$, it follows that

$$\delta_i(s,t) \leq d_i(s,t) \leq \alpha d_j(s,t) \leq \alpha^{l+1} \delta_j(s,t)$$

Since the bounded leg distance can only decrease with a larger leg-bound, for all $i \leq q \leq j, \delta_j(s,t) \leq \delta_q(s,t) \leq \delta_i(s,t)$. Therefore

$$\delta_q(s,t) \leq \delta_i(s,t) \leq d_i(s,t) \leq \alpha^{l+1} \delta_j(s,t) \leq \alpha^{l+1} \delta_q(s,t)$$

Thus $d_i(s,t)$ is an α^{l+1}-approximation of $\delta_q(s,t)$ for any $(s,t) \in (V \times V) \setminus P$.

We can add the edge set $E^{[i+1,q]} = \{e_{i+1}, e_{i+2}, ..., e_q\}$ into d_i, that is, for all $(s,t) \in E^{[i+1,q]}$, if $(s,t) \in P$, set $d_i(s,t) = \min\{d_i(s,t), w(s,t)\}$. This takes $q - i = O(j - i)$ time. We can ignore $E^{[1,i]}$ because for all $(s,t) \in E^{[1,i]}$, $\delta_i(s,t) \leq w(s,t) \leq w(e_i)$. If $\delta_j(s,t) < \delta_i(s,t)$ then $\delta_j(s,t) \geq w(e_i) \geq \delta_i(s,t)$, which is a contradiction. Thus, if $(s,t) \in E^{[1,i]}$ then $(s,t) \notin P$. Now for all $(s,t) \in P \cap E^{[1,q]} = P \cap E^{[i+1,q]}$, $\delta_q(s,t) \leq \delta_i(s,t) \leq d_i(s,t) \leq w(s,t)$. From lemma 2.1, if we take d_i and P as the input of the *Modified-Floyd* procedure, in $O(n|P|)$ time we can find an α^{l+1}-approximation of δ_q; call it d_q.

COROLLARY 2.1. *Let $k = \frac{m}{2^l}$. If we already have an α^l-approximation $d_{\lfloor i \cdot k \rfloor}$ of $\delta_{\lfloor i \cdot k \rfloor}$ for all $0 \leq i \leq 2^l$, then we can find an α^{l+1}-approximation $d_{\lfloor i \cdot \frac{k}{2} \rfloor}$ of $\delta_{\lfloor i \cdot \frac{k}{2} \rfloor}$ for all $0 \leq i \leq 2^{l+1}$ in $O(n^3 \log_\alpha n)$ time.*

Proof. Apply lemma 2.2 to all pairs of adjacent graphs $G_{\lfloor i \cdot k \rfloor}$ and $G_{\lfloor (i+1) \cdot k \rfloor}$ $(0 \leq i < 2^l)$, and let P_i be the set of pairs P for them. Since $\delta^{L(u,v)}(u,v) \leq (n-1)\delta_G(u,v)$, the number of times (u,v) can appear in the sets P_i is $O(\log_\alpha n)$. Thus, the total time taken by this procedure is $O(n \cdot \sum_{i=0}^{2^l - 1} |P_i| + m) = O(n^3 \log_\alpha n)$.

Now we can apply Corollary 2.1 repeatedly and obtain the main algorithm.

THEOREM 2.1. *For any graph G of n vertices and m edges, we can construct the set $D(u,v)$, for every pair of vertices (u,v), that contains a $(1+\epsilon)$-approximation of $\delta_q(u,v)$ for any $0 < q \leq m$, in $O(\epsilon^{-1} n^3 \log^3 n)$ time.*

Proof. First, set $d_0(u,v) = +\infty$ for all (u,v), and utilize the original Floyd-Warshall algorithm to compute $d_m(u,v) = \delta_G(u,v)$ for all pairs (u,v) in $O(n^3)$ time.

Then set $\alpha = (1+\epsilon)^{\frac{1}{\log m}}$, and run the procedure of Corollary 2.3 for $l = 0, 1, ..., \log_2 m - 1$. Finally we can get an $(\alpha^{\log_2 m} = 1+\epsilon)$-approximation of all bounded leg distances for δ_q where $0 < q \leq m$. Thus the total time of this algorithm is $O(n^3 \log n \log_\alpha n) = O(\epsilon^{-1} n^3 \log^3 n)$.

Every time we finish a run of the *Modified Floyd* algorithm in graph G_q, we insert $(w(e_q), d_q(u,v))$ into $D(u,v)$ for every pair (u,v) in P. This will take time $O(|P| \log \log_{1+\epsilon} n)$, which is much less than the *Modified Floyd* algorithm itself. Finally we can see that in $D(u,v)$, the two entries $(L_i, d^{L_i}(u,v))$ and $(L_{i+2}, d^{L_{i+2}}(u,v))$ must satisfy $\frac{d^{L_i}(u,v)}{d^{L_{i+2}}(u,v)} > 1 + \epsilon$ otherwise the intermediate entry $(L_{i+1}, d^{L_{i+1}}(u,v))$ would not be computed in this algorithm. So the size of $D(u,v)$ is bounded by $2\epsilon^{-1} \log n$.

We can see that the space complexity for every execution of the procedure is $O(n^2)$, and the depth of the recursion is $O(\log n)$. So the total space complexity is $O((1+\epsilon^{-1}) n^2 \log n)$.

Answering a bounded leg shortest path query In addition to answering approximate bounded leg distance queries, we also want to find a path of that distance satisfying the leg bound. Answering path queries is what made the space bound of the Roditty-Segal algorithm [14] $O(n^{2.5})$ rather than $O(n^2 \epsilon^{-1} \log n)$. Given a pair of vertices (u,v) and a leg bound L, we want to find a path γ such that $\forall e \in \gamma$, $w(e) \leq L$ and $\sum_{e \in \gamma} w(e) = d^L(u,v)$, where $d^L(u,v)$ is the $(1+\epsilon)$-approximation we obtained from the structure $D(u,v)$.

It is easy to achieve this since all our distances are obtained from the *Modified Floyd* algorithm. We can save the intermediate vertex in every step of Floyd algorithm, then recursively find the two subpaths. We will slightly change our structure and algorithm. For any pair (u,v), any entry $(L_i, d^{L_i}(u,v)) \in D(u,v)$, we define a function $\pi^{L_i}(u,v) \in V$ to be the vertex with the highest index in the real path from u to v of distance $d^{L_i}(u,v)$ under leg bound L_i; if the path only consists of one edge, then $\pi^{L_i}(u,v) = nil$. Recall that in the third line of the algorithm in Figure 1, $d[s,t]$ is assigned $\min\{d[s,t], d[s,v_k] + d[v_k,t]\}$. If $d[s,t]$ does not change after executing this line, then $\pi(s,t)$ also does not change. If $d[s,t] = d[s,v_k] + d[v_k,t]$, then $\pi(s,t)$ will be set to v_k. After this procedure, we can add the entry

GetPath(u, v, L)
  Find $(L', d^{L'}(u,v), \pi^{L'}(u,v)) \in D(u,v)$
    where $L' \leq L$ is maximal.
  If $\pi^{L'}(u,v) = nil$, return the edge (u,v).
  Else Let $w = \pi^{L'}(u,v)$.
    GetPath(u, w, L')
    GetPath(w, v, L')

Figure 2: Algorithm for finding paths

$(L, d^L(u,v), \pi^L(u,v))$ to $D(u,v)$. The procedure to find the path is shown in Figure 2:

Since the leg bound L can only decrease in each recursive call, this recursive procedure will correctly output an approximate bounded leg shortest path from u to v in $O(\log(\epsilon^{-1}\log n))$ time per edge.

Our binary partition algorithm is essentially optimal in terms of n since the fastest standard all-pairs shortest path algorithm takes $\tilde{O}(n^3)$ time [4]. For sparse graphs we can reduce the construction time of our algorithm to $\tilde{O}(mn^{3/2})$ using a two-level approach. A description of this algorithm will appear in the full version of the paper.

3 Bounded-Leg Reachability Oracles in Planar Digraphs

In this section, we will construct some compact oracles in planar digraphs which can answer the bounded leg reachability queries of the form "given a leg bound L, is v reachable from u?" in sublinear time. The best reachability oracle for directed planar graphs is due to Thorup [18] (see also [8]), which takes $O(n \log n)$ space and requires constant query time. However, they depend on having path separators of the graph. In the bounded leg reachability problem a path does not really exist unless all of its edges are below the threshold L. Instead of looking at path separators we exploit a *non-crossing* property of bounded leg reachability and use a search algorithm similar to that of Fakcharoenphol and Rao [6].

THEOREM 3.1. *Given a planar graph $G = (V, E)$, we can construct a structure of size $O(kn^{1+1/k})$ in $O(kn^{1+1/k}\log^2 n)$ time, such that any reachability query can be answered in $O(n^{\frac{k-1}{2k}}\log^2 n)$ time, where k is an integer greater than 1. When $k = 2$ ($k = 3$), we can obtain a structure of size $O(n^{3/2})$ ($O(n^{4/3})$) with query time $O(n^{1/4}\log n)$ ($O(n^{1/3}\log n)$).*

3.1 Brief Description. As in the paper by Fakcharoenphol and Rao [6], the first step is to decompose the graph into subgraphs called *pieces* such that for all $e \in E$, there is a unique piece in one level which contains e. For any integer $k > 1$, we can partition the whole graph into k levels by dividing every piece into $O(n^{1/k})$ subpieces recursively. We call the whole graph a level-0 piece, and the pieces we get from the first partition level-1 pieces, and so on. Every edge forms a level-k piece. A vertex v is a *border node* of a piece P if $v \in P$ and v is adjacent to some vertex outside P.

We can see that in this partition, every vertex in G is a border node of some piece, so any path between two vertices is composed of subpaths among border nodes. There are 3 kinds of subpaths we have to consider: the paths between border nodes of a single piece; the paths travelling from border nodes of a piece P to border nodes of a piece containing P or a piece contained in P; and the paths travelling between border nodes of pieces from the same level. (See Figure 5). The reason we partition the paths in this way is that each of these 3 kinds of paths can be decomposed into several reachability relations satisfying a certain *non-crossing property* (See Figure 4). We will give the definition of the *non-crossing property* in Section 3.2.

In a pair of two lists satisfying the non-crossing property, we will use a *range structure* that stores all the bounded leg reachability information between these two lists. The size of this structure is the product of the sizes of these two lists, but it only takes $O(\log n)$ to find the range in one list reachable from a vertex in the other list under a given leg bound. In a query of the reachability between two vertices, we can run the BFS process on range structures which can cover all the necessary subpaths, so the time needed for the BFS is roughly linear in the total number of vertices in these lists.

We build such range structures for all pairs of lists needed to cover all types of subpaths. Then we can see that the total size of the structure is roughly linear in the number of pieces partitioned every time. When given a query of two vertices, the time needed to run a BFS on such a structure is roughly linear in the number of border nodes we consider.

3.2 Range Structures. First, we introduce the non-crossing property which is similar to the Monge property in [6] (Figure 3).

PROPERTY 3.1. *Let A and B be two ordered lists. A ternary relation "$a \xrightarrow{L} b$", where $a \in A, b \in B, L \in \mathbb{R}^+$, satisfies the monotone non-crossing property if $a \xrightarrow{L} b$ implies $a \xrightarrow{L'} b$, for all $L' > L$, and if, for $v \in A$, $x, y, z \in B$, and $x < y < z$, it always holds that if $v \xrightarrow{L} x$, $v \xrightarrow{L} z$, and $v \not\xrightarrow{L} y$, then for all $u \in A$, $u \not\xrightarrow{L} y$.*

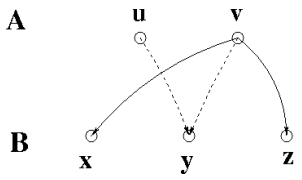

Figure 3: Non-crossing property for reachability

Now when given two ordered lists A and B and a non-crossing ternary relation "$\xrightarrow{L}$" on them, we describe a structure that can store all the information of that relation in $O(A \cdot B)$ space and can answer queries of the form "Given L and $a \in A$, find the set $\{b \in B | a \xrightarrow{L} b\}$".

The structure is composed of two arrays. The first one is $h(b)$ for all elements $b \in B$, which is defined as:

$$h(b) = \min_{a \in A} L(a, b)$$

where $L(a, b) = min\{L | a \xrightarrow{L} b\}$. When the non-crossing property holds, the elements in B that are reachable from $a \in A$ under any leg bound form a range $[x, z] \subseteq B$ with some holes in it. That is, x is the smallest element that a can reach and z is the greatest element that a can reach and the "holes" refer to the elements in (x, z) which are not reachable from *any* elements in A. We define the following structure:

For all $a \in A$, let $l_L(a)$ be the leftmost element b in B that satisfies $L(a, b) \leq L$, and $r_L(a)$ as the rightmost element b in B that satisfies $L(a, b) \leq L$. So $l_L(a)$ and $r_L(a)$ form a range R in B which is reachable from a with some holes. When the L increases, the range expands, and the holes shrink. It is obvious that the range from one element in A can change at most $|B|$ times. Let $F(a) = \{(R, L_R)\}$ where the range for a becomes R under L_R. There are at most $|B|$ changes for R, so $|F(a)| \leq |B|$. In this structure $F(a)$ is sorted in increasing order by L_R.

LEMMA 3.1. *For all $a \in A$, an element $b \in B$ is reachable from a under L if and only if $b \in [l_L(a), r_L(a)]$ and $h(b) \leq L$.*

Proof. If $b \in B$ is reachable from a under L, then $L(a, b) \leq L$, so $l_L(a) \leq b \leq r_L(a)$ and $h(b) \leq L(a, b) \leq L$.

Now consider in the case that $l_L(a) \leq b \leq r_L(a)$ and $h(b) \leq L$, if $b = l_L(a)$ or $b = r_L(a)$ then we of course have $L(a, b) \leq L$. So we only have to consider the case $l_L(a) < b < r_L(a)$. Assume under L, b is not reachable from a, since a can reach $l_L(a)$ and $r_L(a)$, from the non-crossing property, for all $a' \in A$, a' cannot reach b. So for all $a' \in A$, $L(a', b) > L$, and $h(b) > L$, contradicting to the conditions.

From this lemma we can see that the range structure consisting of the arrays h and F contain all reachability information for A and B and occupy $O(|A| \cdot |B|)$ space.

3.3 Use of Range Structures in Planar Graphs.

In this section, we describe the usage of the non-crossing property and its range structures in planar graphs. As in section 3.1, we consider a situation in which the planar graph G has been multilevel partitioned into pieces. From [6], we can assume when fixing a plane embedding, the border nodes have a clockwise order. Denote the circular ordered list of border nodes of the piece G_i by $\sigma(G_i)$.

3.3.1 Intrapiece Paths. For any piece M, consider the circular ordered list $\sigma(M)$. If we divide $\sigma(M)$ into two halves $\sigma_1(M)$ and $\sigma_2(M)$, then the bounded leg reachability between them in both directions will satisfy the non-crossing property if we only consider the paths lying inside M and not intersecting with other border nodes except the two end points. (See Figure 4(a).) We can construct the range structures for $\sigma_1(M)$ to $\sigma_2(M)$ and $\sigma_2(M)$ to $\sigma_1(M)$. Then for each $\sigma_i(M)$ $(i = 1, 2)$, we can divide it further into two lists and still use range structures for them. We perform this partition recursively resulting in a structure containing all reachability information between any pairs of the border nodes of M, The size of this data structure is $O((\frac{|\sigma(M)|}{2})^2 + 2(\frac{|\sigma(M)|}{4})^2 + ...) = O(|\sigma(M)|^2) = O(M)$.

If we only consider the paths lying outside M between the border nodes of M, which do not intersect with other border nodes, we can construct a similar structure of size $O(M)$ by the same procedure. (See Figure 4(b).) We denote all these range structures for the paths inside and outside the piece M by $\Psi(M)$.

3.3.2 Interpiece Paths. The bounded leg reachability between border nodes of two different pieces can also satisfy the non-crossing property in a circular sense. (See Figure 4(c),(d).) We only need a slight change in the range structure:

For any two pieces M and N, consider the two circular ordered list $\sigma(M)$ and $\sigma(N)$. Let $\sigma(N) = (v_1, v_2, ..., v_p)$, where v_1 is the right neighbor of v_p. When $i < j$, the range $[v_i, v_j] = \{v_i, v_{i+1}, ..., v_j\}$ and the range $[v_j, v_i] = \{v_j, v_{j+1}, ..., v_p, v_1, v_2, ..., v_i\}$.

If we only count the paths which do not intersect with $\sigma(M)$ and $\sigma(N)$ except the two end points, then for any $u \in \sigma(M)$, if u can reach $v_i, v_j \in \sigma(N)$, then the range between v_i and v_j and the two paths $u \rightarrow v_i$

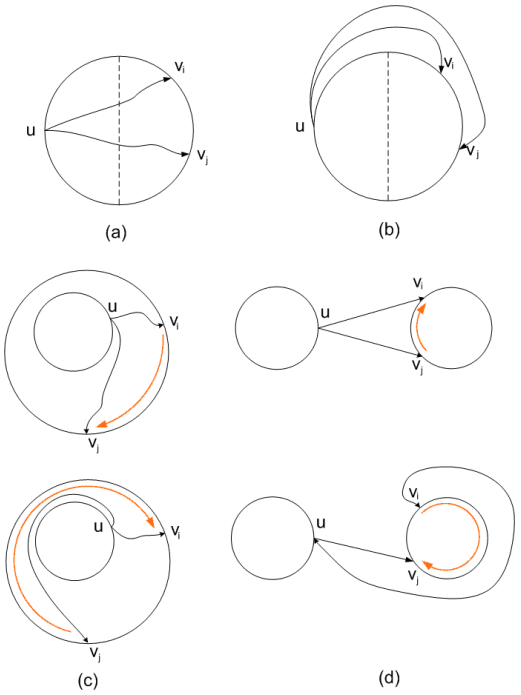

Figure 4: Non-crossing property for border nodes of pieces.

and $u \to v_j$ will form a closed triangle. Whether it is the range $[v_i, v_j]$ or $[v_j, v_i]$ depends on the topological structure of these two paths. When the pieces M and N are disjoint, we consider the paths in the subgraph $G \setminus (M \cup N)$ (See Figure 4(d).) When one piece M contains the other piece N, we consider the paths in the subgraph $M \setminus N$. (See Figure 4(c).)

Suppose the range for u is $[v_i, v_j]$ under leg bound L. When the leg bound increases, if u can reach a vertex v_k outside $[v_i, v_j]$, then the range will expand to $[v_k, v_j]$ or $[v_i, v_k]$. Thus, if we just count the paths which do not intersect with $\sigma(M)$ and $\sigma(N)$, $\sigma(M)$ and $\sigma(N)$ will satisfy the **non-crossing property** in circular order:

PROPERTY 3.2. *For any leg bound L and $u \in \sigma(M)$, $v_i, v_j \in \sigma(N)$. If $u \to v_i$ and $u \to v_j$, then either the open interval $R = (v_i, v_j)$ or $R = (v_j, v_i)$ will satisfy: for any vertex v in R, if u cannot reach v, then all the vertices in $\sigma(M)$ cannot reach v.*

We can still use our range structure to change the leftmost or the rightmost end point of the range to v_k. Define the range structure for $\sigma(M)$ to $\sigma(N)$ and $\sigma(N)$ to $\sigma(M)$ by $\Psi(M, N)$.

3.4 Main Structure. As in the paper by Fakcharoenphol and Rao [6], an n-node planar graph G can be partitioned into pieces $G_1, G_2, ..., G_{O(p)}$ where any G_i contains no more than $\frac{n}{p}$ edges and $O(\sqrt{\frac{n}{p}})$ border nodes.

For any integer $k > 1$, we can partition the whole graph into k levels by dividing every piece into $O(n^{1/k})$ subpieces each time. We call the whole graph as a level-0 piece, and the pieces we get from the first partition level-1 pieces, etc Every single edge is a level-k piece. The border nodes of every level-k piece is the two end points of the edge. So a level-i piece contains $O(n^{1-i/k})$ nodes and $O(n^{\frac{k-i}{2k}})$ border nodes. For every vertex v, we denote one of the level-i pieces containing it by $P_i(v)$. If it is contained in more than one level-i piece, then it is a level-i border node. Denote the minimal level in which v is a border node by j_v. Our structure consists of the following parts:

1. In a level-i piece $(0 \le i < k)$ P, for every pair of level-$(i+1)$ pieces p_1 and p_2 in P, the range structure $\Psi(p_1, p_2)$ with respect to the subgraph $G \setminus (p_1 \cup p_2)$.

2. For every level-i piece $(1 \le i \le k)$ P, for its level-$(i-1)$ parent piece P', the range structure $\Psi(P, P')$ with respect to the subgraph $P' \setminus P$.

3. For every level-i piece P, the structure $\Psi(P)$.

Now we analyze the size of this structure. In part 1, the space needed for every level-i piece P is $O(n^{2/k}) \times O(|\sigma(p_1)|) = O(n^{\frac{k-i+1}{k}})$, and the space needed for all level-i pieces is $O(n^{1+1/k})$. In part 2, the space for every level-i piece P is $O(n^{1-\frac{2i-1}{2k}})$, and total space for all level-i pieces is $O(n^{1+\frac{1}{2k}})$. In part 3, the space needed for all level-i piece is $O(n)$. Thus, the total size of the above structure is $O(kn^{1+1/k})$.

LEMMA 3.2. *Given a leg bound L in G, the query of the reachability from u to v can be answered in $O(n^{\frac{k-1}{2k}} \log^2 n)$ time.*

Proof. When answering a query of the reachability from u to v given a leg bound L, we find $J = \min\{i | P_i(u) \neq P_i(v)\}$. Thus, $P_{J-1}(u) = P_{J-1}(v)$. Then, using the set of structures $S(u, v)$ given below, we can run the BFS algorithm from u on the vertex set:

$$(\bigcup_{i=J}^{j_u} \sigma(P_i(u))) \cup (\bigcup_{i=J}^{j_v} \sigma(P_i(v)))$$

where $S(u,v)$ consists of:

$$\begin{aligned}
&\Psi(P_i(u)) && (J \le i \le j_u)\\
&\Psi(P_i(v)) && (J \le i \le j_v)\\
&\Psi(P_i(u), P_{i-1}(u)) && (J < i \le j_u)\\
&\Psi(P_{i-1}(v), P_i(v)) && (J < i \le j_v)\\
&\Psi(P_J(u), P_J(v))
\end{aligned}$$

When running the BFS algorithm on range structures, initially all vertices except the source u are unreachable. Every time we select a reachable vertex w, we find the ranges reachable from w in all range structures associated with w. After this step, we say w has been scanned. In a range structure containing w, the new reachable range obtained by scanning w can be merged with the reachable ranges already found. By Lemma 3.2, the scanning and merging processes in total take $O(\log n)$ time. In the next step, we can randomly select a unscanned vertex w' in the reachable ranges in any range structure which satisfies $h(w') \le L$. From Lemma 3.2, w' must be reachable from u. We continue this process until v is scanned or all reachable vertices have been scanned.

Since we need $O(\log n)$ time to scan a vertex, the total time needed for the above procedure is the sum of number of vertices in all range structures in $S(u,v)$. For structures like $\Psi(M)$, the total number of vertices in all range structures is $O(\sqrt{|M|}\log n)$. Thus, we can see that the total time for the BFS algorithm is $O(\log^2 n(n^{\frac{k-1}{2k}} + n^{\frac{k-2}{2k}} + n^{\frac{k-3}{2k}} + ...)) = O(n^{\frac{k-1}{2k}}\log^2 n)$.

Now we have to prove that any path that may reach v from u is composed of the subpaths in $S(u,v)$.

LEMMA 3.3. *In Lemma 3.2, if v is reachable to u under leg bound L, then the BFS algorithm on $S(u,v)$ must end with v scanned.*

Proof. In a path from u to v, define u_i^l and v_i^l to be the last vertices in $\sigma(P_i(u))$ and $\sigma(P_i(v))$ respectively, and define $u_i^f (i > j_u)$ to be the first vertex in $\sigma(P_i(u))$ after u_{i-1}^l, and define $v_i^f (i < J)$ to be the first vertex in $\sigma(P_i(v))$ after v_{i+1}^l. Also $u_{j_u}^f$ is u, and v_J^f is the first vertex in $\sigma(P_J(v))$.

If we also regard $\Psi(P)$, $\Psi(P,P')$ as the set of paths they can represent. Then the subpaths from u_i^f (v_i^f) to u_i^l (v_i^l) are in $\Psi(P_i(u))$ $(\Psi(P_i(v)))$, and $u_i^l \to u_{i-1}^f (J < i \le j_v)$ are in $\Psi(P_i(u), P_{i-1}(u))$, $v_i^l \to v_{i+1}^f (J \le i < j_v)$ are in $\Psi(P_i(v), P_{i+1}(v))$, and $u_J^l \to v_J^f$ is in $\Psi(P_J(u), P_J(v))$. An example is shown in Figure 5.

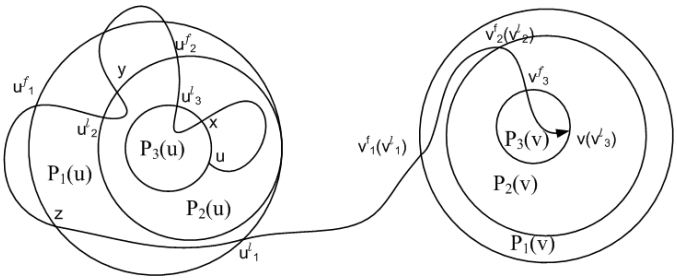

Figure 5: A path from u to v. In this graph, the paths $u \to x$ and $x \to u_1^l$ is contained in $\Psi(P_3(u))$, and the path $u_1^l \to u_2^f$ is in $\Psi(P_3(u), P_2(u))$, the paths $u_2^f \to y \to u_2^l$ is both in $\Psi(P_2(u))$, the path $u_2^l \to u_1^f$ is in $\Psi(P_2(u), P_1(u))$, the paths $u_1^f \to z \to u_1^l$ is both in $\Psi(P_1(u))$, and the path $u_1^f \to v_1^f$ is in $\Psi(P_1(u), P_1(v))$.

3.5 Construction and Simplification.

LEMMA 3.4. *The above structure of size $O(kn^{1+1/k})$ and query time $O(n^{\frac{k-1}{2k}}\log^2 n)$ can be constructed in time $O(kn^{1+1/k}\log^2 n)$.*

The construction of this structure will be discussed in Appendix 5.1.

When $k = 2$ or $k = 3$, we can get rid of one log factor in the query time and obtain a structure of size $O(n^{3/2})$ with query time $O(n^{1/4}\log n)$, and a structure of size $O(n^{4/3})$ with query time $O(n^{1/3}\log n)$. When $k = \log_2 n$, we can simplify the structure of size $O(n\log n)$ with query time $O(\sqrt{n}\log^2 n)$. These are discussed in Appendix 5.2.

4 Conclusion

We have shown that a $(1+\epsilon)$-approximate bounded leg distance oracle can be constructed in $O(\epsilon^{-1}n^3\log^3 n)$ time that occupies $O(\epsilon^{-1}n^2\log n)$ space. Although the construction time could perhaps be improved by a few log factors the dependence on ϵ *cannot* be improved without a fundamentally new approach to the problem. In particular the query algorithm would need to perform some arithmetical operations in the course of answering a distance query in order to improve our space bounds.[1]

The main problem left open by this work is to improve our *reachability* data structures for planar graphs. Two obvious problems are to handle approximate bounded leg *distance* queries or to improve our time-space tradeoff. There is no reason to believe that the best possible tradeoff—$O(1)$ query time and linear

[1]This is not a problem specific to the bounded leg case. The best data structure known for answering exact distance queries in $O(1)$ time is simply an $n \times n$ table.

space—could not be achieved with more sophisticated techniques.

References

[1] S. Baswana and T. Kavitha. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *Proc. 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 591–602, 2006.

[2] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in expected $O(n^2)$ time. *ACM Trans. Algorithms*, 2(4):557–577, 2006.

[3] P. Bose, A. Meheswari, G. Narasimhan, M. Smid, and N. Zeh. Approximating geometric bottleneck shortest paths. *Computational Geometry: Theory and Applications*, 29:233–249, 2004.

[4] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 590–598, New York, NY, USA, 2007. ACM Press.

[5] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. 35th ACM Symp. on the Theory of Computing*, 2003.

[6] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. System Sci.*, 72(5):868–889, 2006.

[7] H. N. Gabow and R. E. Tarjan. Algorithms for two bottleneck optimization problems. *J. Algorithms*, 9(3):411–417, 1988.

[8] P. N. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proc. 13th Ann. ACM-SIAM Symp. On Discrete Algorithms (SODA)*, pages 820–827, 2002.

[9] P. N. Klein and S. Subramanian. A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs. In *Algorithms and data structures (Montreal, PQ, 1993)*, volume 709 of *Lecture Notes in Comput. Sci.*, pages 442–451. Springer, Berlin, 1993.

[10] L. Kowalik and M. Kurowski. Oracles for bounded-length shortest paths in planar graphs. *ACM Trans. Algorithms*, 2(3):335–363, 2006.

[11] M. Mendel and A. Naor. Ramsey partitions and proximity data structures. *J. Eur. Math. Soc. (JEMS)*, 9(2):253–275, 2007.

[12] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoret. Comput. Sci.*, 312(1):47–74, 2004.

[13] L. Roditty. A faster and simpler fully dynamic transitive closure. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 404–412, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[14] L. Roditty and M. Segal. On bounded leg shortest paths problems. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 775–784, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[15] L. Roditty, M. Thorup, and U. Zwick. Deterministic constructions of approximate distance oracles and spanners. In *Automata, languages and programming*, volume 3580 of *Lecture Notes in Comput. Sci.*, pages 261–272. Springer, Berlin, 2005.

[16] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science*, page 679, Washington, DC, USA, 2002. IEEE Computer Society.

[17] S. Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms—ESA '93 (Bad Honnef, 1993)*, volume 726 of *Lecture Notes in Comput. Sci.*, pages 372–383. Springer, Berlin, 1993.

[18] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024 (electronic), 2004.

[19] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24 (electronic), 2005.

[20] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 585–589, New York, NY, USA, 2007. ACM Press.

5 Appendix

5.1 Construction of the reachability oracle.

Proof of theorem 3.7:

The time needed for the piece decomposition is $O(n \log n)$. [17, 6]

In order to construct this structure efficiently, we must first construct a *dense distance graph* like the one of Fakcharoenphol and Rao [6]. A *dense distance graph* has a k-level decomposition of the graph as in the structure, and for every piece P in it, it contains the all-pair distances of the border nodes of P only considering the path inside P, that is, $\{L_P(u,v) | u, v \in \sigma(P)\}$.

For the non-crossing arrays A and B, if we already have all the distances from the vertices in A to the vertices in B, then the number of edges in these arrays is $|A| \times |B|$. Can we use this structure in a Dijkstra algorithm and only cost $\widetilde{O}(|A| + |B|)$ time in total?

FACT 5.1. *If the ordered lists A and B satisfy the non-crossing property, then they satisfy the Monge property: $\forall u, v \in A, x, y \in B$, if $u \leq v$ and $x \leq y$, then $\max(L(u,x), L(v,y)) \leq \max(L(u,y), L(v,x))$.*

So, in the the Dijkstra algorithm, the vertices in B whose shortest leg paths go through a scanned vertex in A form a range in B. When we scan a new vertex u in A, instead of testing every node in B to check whether its distance decreases, we take $O(\log |B|)$ time to find the start and the end of its range. Since the vertices inside the range have longer or equal distances than u,

they must all be unscanned. So we can find the node in this range with minimum distance in $O(\log |B|)$ time and then put u and its range in a heap H. When we scan some vertex in this range later, we can split it into two ranges and find the minima of them and then put them into H. This will also take $O(\log |B|)$ time. The total time for the lists A and B is $O((|A|+|B|)\log |B|)$.

As above, for a piece M, consider the circular ordered list $\sigma(M)$. If we divide $\sigma(M)$ into two halves, they will satisfy the non-crossing property if we only consider the paths lying inside M and not intersecting with other border nodes except the two end points. And we can partition them recursively. So if we already have the dense distance graph for M, we can use it in Dijkstra algorithm for a total time of $O(|\sigma(M)|\log^2 |\sigma(M)|) = O(\sqrt{|M|}\log^2 |M|)$.

Now suppose we already obtain the dense distance graphs for all level-i ($1 < i \leq k$) pieces. For any level-$(i-1)$ piece P, we would find the dense distance graph for it. Since a border node for P is also a level-i border node, consider the dense distance graphs for all the level-i pieces in P, and for any $u \in \sigma(P)$ as a start point, run the Dijkstra algorithm on it. The total time needed for constructing the dense distance graph for P is $O(n^{\frac{2k-2i+3}{2k}}\log^2 n)$, and therefore the total time for level-$(i-1)$ is $O(kn^{1+1/(2k)}\log^2 n)$.

For a detailed description of this procedure, see Fakcharoenphol and Rao. [6]

In a level-i piece P, to compute the distances $L_{G\setminus P}(u,v)$ for any border nodes u and v of P, find the level-$(i-1)$ piece P' that contains P (If we already have the distances for P'), then we can run the Dijkstra algorithm for all the level-i pieces in P'. This procedure also finds the distances needed for paths lying outside P between the border nodes of P. The total time needed for this step is $O(kn^{1+1/k}\log^2 n)$.

We need to construct the range structure for two circular ordered vertex lists $\sigma(M)$ and $\sigma(N) = (v_1, v_2, ..., v_p)$ given the all-pair bottleneck distances: $\{L(u,v)|\forall u \in \sigma(M), v \in \sigma(N)\}$. First, for all $v \in \sigma(N)$, compute $h(v) = \min_{u\in\sigma(M)} L(u,v)$. Then, for any $u \in \sigma(M)$, consider the list: $Q = (L(u,v_1), L(u,v_2), ..., L(u,v_p))$. Sort this list in increasing order, obtaining Q'. Suppose the minimum element is $L(u,v)$, so when $L = L(u,v)$, the range for u is just $[v,v]$. If the range for u is $[v_i,v_j]$ when the leg bound is L, when the leg bound increases, u may reach a vertex v_k outside $[v_i,v_j]$, then the range will expand. Any vertex v' satisfying $L(u,v') > L(u,v_k)$ in the new range will satisfy $L(u,v') = h(v')$, because under leg bound less than $L(u,v')$ greater than $L(u,v_k)$, u cannot reach v', by the non-crossing property, u cannot reach all other vertices in $\sigma(M)$. Checking all vertices in a range to see whether they satisfy $L(u,v') = h(v')$ takes $O(\log |\sigma(N)|)$ time. So the total time needed build up the range structure is $O(\sqrt{|M||N|}\log |N|)$.

5.2 Special Cases when $k = 2$, $k = 3$ and $k = \log n$.

LEMMA 5.1. *We can construct a structure of size $O(n^{4/3})$ with query time $O(n^{1/3}\log n)$, compared to the query time of $O(n^{1/3}\log^2 n)$ in the above structure.*

When $k = 3$, add another part into the structure:

- For every vertex u, construct $\Psi(u, P_1(u)) = \{(w, L_G(u,w))|w \in \sigma(P_1(u))\}$ and $\Psi(P_1(u), u) = \{(w, L_G(w,u))|w \in \sigma(P_1(u))\}$. Since $|\sigma(P_1(u))| = n^{1/3}$, the total size of this part is also $O(n^{4/3})$.

LEMMA 5.2. *Given a leg bound L in G, the query time of the reachability from u to v for any pair of vertices (u,v) in G can be reduced to $O(n^{1/3}\log n)$.*

Proof. If u and v are in the same level-1 piece, that is, $J > 1$, then the query time in the original structure is actually $O(n^{1/6}\log^2 n)$. If u and v are not in the same level-1 piece, we can run the Dijkstra algorithm for the vertices $\{u\} \cup P_1(u) \cup P_1(v) \cup \{v\}$ using the structures $\Psi(u, P_1(u))$, $\Psi(P_1(u), P_1(v))$ and $\Psi(P_1(v), v)$. So, the running time is only $O(n^{1/3}\log n)$.

Now we finish the proof of theorem 5.2

To construct this part, consider the edges in a level-1 piece P and the all-pair distances $L_{G\setminus P}$ for the border nodes of P, and run the normal Dijkstra algorithm from all border nodes of P. Then, the time needed for one border node is $|P| = O(n^{2/3})$, so the total time is $O(n^{4/3})$.

The same idea also works for $k = 2$:

LEMMA 5.3. *We can construct a structure of size $O(n^{3/2})$ with query time $O(n^{1/4}\log n)$, compared to the query time of $O(n^{1/4}\log^2 n)$ in the above structure.*

When k increases to $\log n$, then every piece is divided into two subpieces, as in the paper by Fakcharoenphol and Rao [6]. We perform a binary partition for every piece every time and obtain two subpieces. A piece with n nodes and r border nodes can be divided into two subpieces, such that each subpiece has no more than $\frac{2}{3}n$ nodes and at most $\frac{2}{3}r + c\sqrt{n}$ border nodes, for some constant c. The recursion stops when a piece contains a single edge. So, the total number of levels of partition is $O(\log n)$. [6] We can then obtain a structure of size $O(n\log n)$ with query time $O(\sqrt{n}\log^2 n)$.