Lecture 8 Compact Routing (cont.) & Dynamic Algorithms
Scribe: Chuanqi Shen **Date:** February 5, 2014

1 Recap

To recap, we are attempting to apply distance oracles to solve the compact routing problem, defined as follows. There is an underlying graph $G = (V, E)$. Each vertex v in V has an "address label" $L(v)$. When a vertex u wants to send a packet P to v, it adds $L(v)$ to the header of P. Additionally, each vertex x of G stores a routing table R_x. When x receives a packet, it looks at the header, reads $L(v)$ and uses $L(v)$ together with its routing table R_x to decide which of its neighbors it should send the packet to.

A compact routing scheme consists of a preprocessing algorithm that constructs R_x and $L(x)$ for each node x, and of a routing algorithm that given R_x and $L(v)$ finds a neighbor of x to route the packet to. We want both R_v and $L(v)$ to be small. Notice that this also entails that if a node has many neighbors it cannot know all of them. We assume that the neighbors are accessed via ports; intuitively, these are just edges that the node x gains knowledge about, either from $L(v)$ or from R_x.

To create a compact routing scheme, we will use distance oracles. As with our distance oracle construction, we randomly select k subsets of the vertex set $A_0, A_1 ..., A_{k-1}$ with $V = A_0 \supset A_1 \supset ... \supset A_{k-1}$ such that $|A_i| \approx \frac{|A_{i-1}|}{n^{1\backslash k}}$. Also, we define $p_i(v) \in A_i$ to be the node in A_i closest to v, and $B(v) = A_{k-1} \cup \left(\bigcup_{i<k-1}\{x \in A_i \mid d(v,x) < d(v, p_{i+1}(v)\}\right)$.

In the previous lecture, we said that R_v will store $p_0(v), p_1(v), ..., p_{k-1}(v)$ and $B(v)$. Also, for each $x \in B(v)$, we will store the next node on the shortest path from v to x. $L(v)$ will store $p_0(v), p_1(v), ..., p_{k-1}(v)$.

Suppose we want to route from u to v. We would like to find the minimum j such that $p_j(v) \in B(u) \cap B(v)$. Then, if we route on the shortest path from u to $p_j(v)$ to v, then we will get $4k - 3$ approximation as proven in the previous lecture.

2 Problems

We presented a first approach last time. However, we encountered problems in the implementation.

1. Suppose we route from x to x' where x' is the next node after x on the shortest path to $p_j(v)$. What happens if $p_j(v) \notin B(x')$?
2. Suppose the current node x is $p_j(v)$. How do we route down to v?

As it turns out, problem 1 will never happen due to the following lemma.

Lemma 2.1. *If $p_j(v) \in B(x)$ and x' is the next node after x on the shortest path to $p_j(v)$, then $p_j(v) \in B(x')$.*

Proof. Note $p_j(v) \in B(x)$ implies $d(x, p_j(v)) < d(x, p_{j+1}(x))$. But

$$d(x, p_j(v)) = d(x, x') + d(x', p_j(v)) < d(x, p_{j+1}(x)) \leq d(x, p_{j+1}(x')) = d(x, x') + d(x', p_{j+1}(x')),$$

where the first equality follows by the definition of x', the first $<$ inequality follows from the fact that $p_j(v) \in A_j \cap B(x)$, the second inequality follows by the definition of $p_{j+1}(x)$ and since $p_{j+1}(x') \in A_{j+1}$.

Therefore

$$p(x', p_j(v)) < p(x', p_{j+1}(x')) \implies p_j(v) \in B(x').$$

□

Now we consider problem 2. Firstly, notice that the route from $p_j(v)$ to v is along the edges of the shortest path tree rooted at $p_j(v)$, so we will try to make use of that. We will try to achieve this using additional $O(\log n)$ memory for our routing tables and $O(\log^2 n)$ memory for our headers.

For every node x in the graph, let's compute a shortest path tree T_x rooted at x. Suppose that there is some scheme that given a tree T, creates labels $L_T(u)$ for each node $u \in T$ and routing tables $R_T(u)$ for each node $u \in T$, so that given $L_T(v)$ and $R_T(x)$, x can route to the next node x' on the tree path from x to v in T.

Then, we can augment our routing scheme for general graphs as follows:

The labels $L(u)$. The label $L(u)$ of u contains $p_0(u), \dots, p_{k-1}(u)$ and for each $j \in \{0, \dots, k-1\}$ the label $L_{T_{p_j(u)}}(u)$ for the node u in the shortest paths tree rooted at $p_j(u)$.

The routing tables R_x. The routing table R_x of x contains for each $y \in B(x)$, the routing table $R_{T_y}(x)$ for x in the shortest paths tree T_y rooted at y.

Routing. Suppose node u wants to send a message to v. First it looks at $L(v)$ and finds the minimum j such that $p_j(v) \in B(v)$. It can do this since $B(v)$ is in R_u. Now let $y = p_j(v)$ and let $T = T_y$ be the shortest paths tree rooted at y. First, u writes y and $L(v)$ in the header of the message. Then u accesses $L_T(v)$ from $L(v)$ and $R_T(u)$ from R_u and uses these to route to the next node u' on the shortest path from u to v in T. From then on, each node that gets the message learns $L(v)$ and y, and can access $R_{T_y}(x)$ since by Lemma 2.1 above and Lemma 2.2 below, $y \in B(x)$. Thus every node on the path $u \longrightarrow p_j(v) = y \longrightarrow v$ can route to the next node of the path until v is reached. By the results from the previous class, we are guaranteed to route along a $4k-3$-approximate path.

Lemma 2.2. *Suppose that x lies on the shortest path between $p_j(v)$ and v. Then $p_j(v) \in B(x)$.*

Proof. Let J be the largest index such that $p_j(v) \in A_J$. Then in particular, $p_j(v) = p_J(v)$, and for $p_{J+1}(v) \neq p_j(v)$. Suppose that $p_j(v) \notin B(x)$. Then, $d(x, p_j(v)) \geq d(x, p_{J+1}(x))$ by the definition of $B(x)$ and J. However, then we have

$$d(v, p_{J+1}(v)) \leq d(v, p_{J+1}(x) \leq d(v, x) + d(x, p_{J+1}(x)) \leq d(v, x) + d(x, p_j(v)) = d(v, p_j(v)),$$

which implies that $p_{J+1}(v) = p_j(v)$ and contradicts $p_{J+1}(v) \neq p_j(v)$. Thus, $p_j(v) \in B(x)$. □

3 Routing on trees

Here we show that one can route along a tree T with routing tables of size $O(\log n)$ and labels of size $O(\log^2 n)$. This increases the routing table size of the general graph algorithm by a $O(\log n)$ factor, and the label size there by a $O(\log^2 n)$ factor.

Let T be a tree rooted at a node r.

First Attempt. Let us perform a DFS traversal of T and label the nodes in DFS traversal order. Identify each node with its DFS number. Observe that after doing so, the nodes in any subtree of T can be represented as a consecutive interval of integers as follows. For each $x \in T$, let $f(x)$ denote the descendent of x with largest DFS label. Then x is on the shortest path from the root to any node in $[x, f(x)]$. Hence, one potential protocol is as follows. Suppose we are at x and we want to route to y. Then, search for a child c of x such that $v \in [c, f(c)]$. Once we find such a child c, we will route to c.

However, since a node can have $\Omega(n)$ children, that can make some of our routing tables huge. Therefore, we need to find some way to reduce the number of edges to children that we store for each node.

Second Attempt For $x \in T$ and for each child x' of x, denote x' a "light" child of x if the subtree rooted at x' contains at most half the nodes in the subtree rooted at x. Otherwise, denote x' as a "heavy" child. Observe that each node x has at most 1 heavy child. Also, any path along T from a node x to some descendant v can contain at most $O(\log n)$ light children since at each light child the number of descendents halves. We will make use of these two properties.

For each $x \in T$, we store the heavy child $h(x)$ of x in $R_T(x)$. As in the first attempt we also store $f(x)$ and x in $R_T(x)$. Now, when we route a packet to v we first check if $v \in [h(x), f(x)]$. If so, then we can route to the heavy child $h(x)$ that we access from $R_T(x)$. If not, we check if $v \in [x, f(x)]$ and if not, we route to the parent $p(x)$ of x that we also store in $R_T(x)$. Otherwise, we conclude that the next node on the path to v is a light child of x. We will get this light child from $L_T(v)$.

Now we describe what we store in $L_T(v)$. Let $e_1, e_2, \ldots, e_t$ be all the light children along the path from the root r to v. We will store the pairs $(p(e_i), e_i)$ in our header $L_T(v)$ ($p(x)$ denotes the parent of x). Now suppose that as above we are at x and have established that the path to v goes through a light child of x. Then we search through $L_T(v)$ for a pair $(p(e_i), e_i)$ where $x = p(e_i)$. If v is a descendent of x but not of $h(x)$ then this pair will be in $L_T(v)$. Then we route to e_i.

By the two properties that we described, the size of $R_T(x)$ for each x is just 4 integers, $x, f(x), h(x), p(x)$ each having $O(\log n)$ bits, thus taking $O(\log n)$ space. The size of $L_T(x)$ is $O(\log n)$ integer pairs, which takes $O(\log^2 n)$ space as promised.

Below, we give the steps to the entire tree routing algorithm.

Algorithm 1: Route(x, $L(v)$)

```
if $v$ is not contained in $[x, f(x)]$ then
    Route($p(x), L(v)$);
    Report done;
else
    if $x = v$ then
        Report done.;
    if $x$ has heavy child then
        Let $h(x)$ be heavy child of $x$;
        if $v$ is contained in $[h(x), f(h(x))]$ then
            Route($h(x), L(v)$);
            Report done;
    Find pair $(p(e_i), e_i)$ in $L(v)$ such that $p(e_i) = x$;
    Route($e_i, L(v)$);
    Report Done;
```

Note that for our application for general graphs we get $|R_v| \approx |B(v)| = O(kn^{1\backslash k})$ and $|L(v)| = O((k + \log n) \log n)$, both of which are pretty good.

4 Dynamic Algorithms

So far, the algorithms we have discussed work only on static graphs. However, we want to know if we can build algorithms that can handle changes to the graphs. If we only change the graph by a tiny bit, it seems that we may only need to find a little extra bit of information with an additional bit of computation. Algorithms that can handle changes to the graph are called dynamic algorithms.

Dynamic algorithms can be decomposed into 3 parts

1. Preprocessing: The original graph is preprocessed, and some data structure is created and stored.

2. Query: Some information about the graph is queried.

3. Update: Some change is made to the graph (e.g. edges are inserted/removed, weights are changed etc).

As an example, in the reachability problem, given a query (u, v), we want to know if it is possible to find a path from u to v. We can also be given update instructions that tell us to insert or remove an edge.

There are two naive ways to write a dynamic algorithm. The first way is to redo preprocessing whenever a update is made. For example, whenever an edge is inserted removed, we can preprocess reachability between all pairs of nodes again to answer queries in constant time. The second way is to do nothing when updating the graph, and compute the result when the query is given. For example, we can merely update the graph when an update instruction is given, and only compute reachability between (u, v) then the query (u, v) is given.

When creating dynamic algorithms, it is sometimes hard to support both insertion and deletion of edges. Therefore, we can create algorithms that only support insertion or deletion. The former kind is called an incremental algorithm while the latter is called a decremental algorithm.

Also, sometimes a dynamic algorithm may take a while when handling certain updates or queries, but perform efficiently in the long run. Therefore, we often look at the amortized runtime when doing complexity analysis.