Adaptive Real-Time Routing in Polynomial Time

Kunal Agrawal and Sanjoy Baruah
Washington University in Saint Louis
{kunal,baruah}@wustl.edu

***Abstract*—We consider a recently-proposed problem on networks in which each individual link is characterized by two delay parameters: a (usually very conservative) guaranteed upper bound on the worst-case delay, and an estimate of the delay that is typically encountered, across the link. Given a source node, a destination node, and an upper bound on the end-to-end delay that can be tolerated, the objective is to determine routes that typically experience a small delay, while guaranteeing to respect the specified end-to-end upper bound under all circumstances. We show that the prior algorithm that has been proposed for this problem has super-polynomial running time, and derive polynomial time algorithms for solving the problem.**

I. Introduction

A recent paper [1] considered real-time routing problems on graphs in which each edge is labeled with a pair of edge-weights, one denoting the *maximum* delay one could encounter while traversing the edge and the other, an estimate of the delay one *typically* encounters upon traversing the edge. The goal is to travel from a specified source vertex to a specified destination vertex on a path that minimizes typical delay, while simultaneously guaranteeing a specified end-to-end delay bound. Such problems are motivated in [1] by numerous application scenarios in real-time computing: consider, for instance, the autonomous vehicles that play a significant role in many testbeds and exemplars of cyber-physical system (CPS) principles and technologies. Such vehicles often need to traverse a network between locations of interest. While it may be possible to obtain worst-case bounds on the delays that will be encountered across each segment of roadway, making routing decisions solely on the basis of such worst-case delay bounds may result in routes that perform needlessly poorly under typical conditions; one should simultaneously seek to optimize for typical traversal times as well.

Figure 1 shows one such graph with source vertex s and destination t. There are four possible paths from s to t and the following table shows the worst case delay (denoted by c_W) and the typical delay (denoted by c_T) encountered on each path.

Path	$c_W(\cdot)$ **cost**	$c_T(\cdot)$ **cost**
$s \to t$	25	12
$s \to v_1 \to t$	20	14
$s \to v_1 \to v_2 \to t$	30	10
$s \to v_1 \to v_3 \to t$	34	6

We see that the paths are ordered differently from the perspectives of typical delay and guaranteed delay; e.g., the fourth path $\langle s \to v_1 \to v_3 \to t \rangle$ is the best with respect to typical delay while the second path is able to *guarantee* arrival within the shortest duration (20 time units rather than 34). This

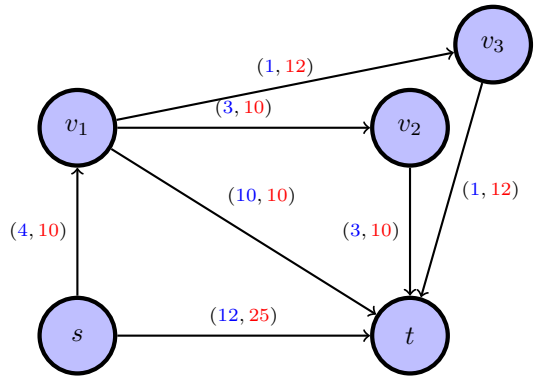

Fig. 1. An example graph. Each edge is characterized by a pair of two weight parameters that are estimates on delay-bounds – the smaller value represents an estimate of the delay that will *typically* be encountered across the edge, while the larger value represents a guaranteed upper bound on such delay.

difference in ordering of path lengths depending upon whether one is looking at typical or worst-case delay complicates the choice of routes. Suppose, for instance, that we would like to travel from s to t rapidly under typical circumstances, but were additionally required to guarantee an end-to-end delay of 25: the first path would be best since only the first two paths are *feasible* (guarantee to get to t within the required 25 time units) and of these the first path has the smaller typical delay.

A distinction was made in [1] between **static** and **adaptive routing strategies**. A static strategy decides the entire route prior to setting out from the source vertex, whereas an adaptive strategy can make decisions on the way. Let us return to our example in Figure 1 where we want to guarantee an end-to-end delay of 25. Consider the following strategy:

> Traverse the edge $s \to v_1$
> **if** (*the actual delay encountered is* ≤ 5 *time units*)
> traverse the path $v_1 \to v_2 \to t$
> **else** traverse the edge $v_1 \to t$

This is an adaptive strategy, since the decision regarding the route to take from v_1 to t is made upon reaching v_1. We can verify that this strategy guarantees the end-to-end delay of 25:

- If the delay encountered on edge (s, v_1) is at most 5 time units, then the total delay from s to t is at most 25 since the maximum delay on $\langle v_1 \to v_2 \to t \rangle$ is $10 + 10 = 20$.
- If, on the other hand, the delay on (s, v_1) is between 5 and 10 time units, taking the direct edge $\langle v_1 \to t \rangle$ guarantees

that the maximum delay from s to t is $10 + 10 = 20$.

Hence this strategy is *correct* in the sense that it respects the end-to-end delay bound. To determine the typical delay associated with this correct strategy, we note that the typical delay on edge (s, v_1) is 4 time units, which implies that upon reaching v_1 we would typically take the path $\langle v_1 \to v_2 \to t \rangle$ for a total typical delay of $4 + 3 + 3 = \mathbf{10}$ time units, in contrast to the 12 time units associated with the optimal static solution, which is the first path in the above table.

Static, fully-adaptive, and semi-adaptive routing. As the above example illustrates, for a given worst-case end-to-end delay bound, *adaptive strategies are generally capable of achieving smaller typical delays than static ones*, by using knowledge of the delay that was experienced across already-traversed edges to make future routing decisions. (We will see in Section VII that static routing strategies may perform arbitrarily poorly in comparison to adaptive ones.) In this paper, we further distinguish between *fully-adaptive* and *semi-adaptive* strategies. A fully-adaptive strategy assumes that upon traversing an edge, the exact actual delay that was encountered while doing so becomes known — the example above illustrates a fully-adaptive strategy. A semi-adaptive strategy, by contrast, makes the weaker assumption that upon traversing an edge it only becomes known whether the actual delay that was experienced exceeds the typical estimate or not – the exact duration of the delay may not be known. An algorithm was presented in [1] for determining optimal fully-adaptive strategies; however, we are not aware of any prior algorithms for determining optimal semi-adaptive strategies.

A fact in favor of static strategies is that their *run-time overhead* tends to be small: at each intermediate vertex only the constant-time operation of looking up the outgoing edge from this pre-computed path needs to be performed. In contrast, fully-adaptive strategies may require non-trivial computations during run-time.

In addition to the per-intermediate-vertex processing time, a routing strategy also typically requires some *pre-runtime computation*. For static strategies, the entire path must be computed in advance. For semi-adaptive and fully-adaptive strategies, one must pre-compute some initial path and then potentially recompute the path to be taken at each vertex based on the information we have so far. Therefore, we can compare different approaches based on *the overall running time*: the sum of the time for computations performed prior to setting out from the source and the times taken at each individual intermediate vertex during run-time. We will see in Section VII that the problem of determining an optimal static route is NP-hard; this implies that we are unlikely to find optimal static routing strategies with overall running time polynomial in the representation of the problem. To our knowledge the computational complexity of optimal adaptive routing has remained open: while the algorithm for determining optimal fully-adaptive routing strategies that was proposed in [1] is easily seen to have polynomial processing time at each intermediate vertex, it is not clear from [1] whether the preprocessing step for that algorithm is a polynomial-time one or not.

Our contributions. We report the following major research findings concerning adaptive routing in this paper.

1) As mentioned above, the overall running time of the adaptive routing strategy presented in [1] was unknown. Our first contribution is to show, in Section IV, that the pre-runtime processing step in this strategy is not, in general, a polynomial-time one; consequently *the overall running time of the algorithm of [1] for determining optimal adaptive routing strategies is not polynomial* in the representation of the problem.
2) We then provide an algorithm for finding optimal semi-adaptive routing strategies whose overall complexity is polynomial in the problem size. In particular, the complexity of pre-runtime computation is the same as that of Dijkstra's algorithm for shortest paths [2], while the runtime complexity at each intermediate vertex is a constant.
3) Our third (and main) contribution is *an algorithm for finding optimal fully-adaptive routing strategies that has overall run-time polynomial* in the representation of the problem and the number of edges in the actual optimal adaptive path realized. (This distinguishes our algorithm from the one in [1] since that has pseudo-polynomial running time regardless of the number of edges in the optimal adaptive path. While it is straightforward to show that the number of edges in the optimal adaptive path is polynomial in problem size for undirected graphs or directed acyclic graphs, we do not yet know whether optimal adaptive paths with polynomially many edges are guaranteed to exist for directed graphs that may contain cycles.)
4) Recall that the typical delay bounds specified for the edges of the graph are *estimates*; what if more accurate estimates become available *after* an optimal routing strategy has been determined (perhaps even after we have traversed some part of the route as dictated by the strategy)? The algorithm of [1] must be re-run (as stated above, we show in this paper that this may take pseudo-polynomial time in the worst case) from scratch; an additional contribution contained in this paper is a result showing that the new algorithm we have derived is able to incorporate such information into future routing decisions in fast polynomial time.

Organization. The remainder of this paper is organized as follows. Section II provides the formal definition of the problem. Section III has a brief description the algorithm proposed in [1]. Our main technical contributions are presented in Sections IV–VI: Section IV contains an example and a proof that the approach of [1] has super-polynomial run-time on this example; Section V derives a polynomial-time algorithm for optimal semi-adaptive routing; and Section VI derives an optimal fully-adaptive routing that has running time polynomial in the graph size and the number of edges in the actual optimal adaptive path. In Section VII we formally

demonstrate some advantages of adaptive routing strategies over static ones.

II. Model and Definitions

As discussed above, we want to calculate routing strategies that provide low typical delays between a source and a destination in a graph, while simultaneously guaranteeing to arrive at the destination within a specified duration of leaving the source even if some (or all) edges experience delays as large as the worst-case estimates. We represent the network as a directed graph $G = (V, E)$, where the vertices represent locations of interest, and the edges, direct connections between pairs of locations. There are two cost functions on edges: $c_T : E \to \mathbb{N}$ and $c_W : E \to \mathbb{N}$. For any edge $(u, v) \in E$, $c_T(u, v)$ and $c_W(u, v)$ respectively denote *an* estimate *of the typical delay*, and *a* guarantee *of the maximum delay*, that will be encountered across this edge. A *path* p from vertex u to vertex v (sometimes designated as $u \overset{p}{\rightsquigarrow} v$) is a sequence of vertices $\langle u \equiv v_0, v_1, v_2, \ldots, v_k \equiv v \rangle$ such that $(v_{i-1}, v_i) \in E$ for each $i, 1 \leq i \leq k$. The cost parameters $c_T(p)$ and $c_W(p)$ of this path p are defined in the obvious manner:

$$c_T(p) = \sum_{i=1}^{k} c_T(v_{i-1}, v_i) \text{ and } c_W(p) = \sum_{i=1}^{k} c_W(v_{i-1}, v_i)$$

An ***instance*** $I = \langle G, c_T, c_W, s, t, D \rangle$ of our problem is specified by specifying such a graph $G = (V, E)$ with the two cost functions $c_T : E \to \mathbb{N}$ and $c_W : E \to \mathbb{N}$, a source vertex $s \in V$ and a terminal vertex $t \in V$, and an end-to-end delay bound $D \in \mathbb{N}$. Instance $I = \langle G, c_T, c_W, s, t, D \rangle$ is said to be *feasible* if there exists some path p from s to t (i.e., $s \overset{p}{\rightsquigarrow} t$) with $c_W(p) \leq D$; otherwise it is *infeasible*. We seek algorithms that are able to traverse the graph G from the source s to the destination t of any feasible instance with minimum delay under typical circumstances (when no edge exceeds its typical delay estimate), while guaranteeing that the delay is no larger than the delay bound D under even worst-case circumstances.

For static strategies — i.e., the entire path from the source vertex s to the destination vertex t must be completely specified prior to departing from s — the optimization criterion is straightforward: we must identify a path p, $s \overset{p}{\rightsquigarrow} t$, satisfying $(c_W(p) \leq D)$, for which $c_T(p)$ is minimized. (Identifying such paths is easily shown to be NP-hard – see Section VII.)

The optimization criterion is somewhat more difficult to state for adaptive strategies. Let us start with some intuition: An adaptive strategy must never allow the entity traversing the network to reach an *unsafe state* during run-time — that is, it must never traverse an edge which could make it impossible to reach the destination vertex within the specified delay bound of D after leaving the source vertex. In addition, assuming the actual delay on each edge is at most its typical delay, we want to minimize the end-to-end delay. To formalize this intuition, we start with some definitions.

Definition 1 ($\mathcal{W}(v)$; $\alpha : V \to \mathbb{N}$): Consider some instance $\langle G, c_T, c_W, s, t, D \rangle$. For every vertex $v \in V$, $\mathcal{W}(v)$ denotes a shortest path from v to t according to the maximum delay bounds (c_W) on the edges. $\alpha(v)$ denotes the sum of the maximum-delay costs of the edges on path $\mathcal{W}(v)$ — that is, $\alpha(v) = c_W(\mathcal{W}(v))$. □

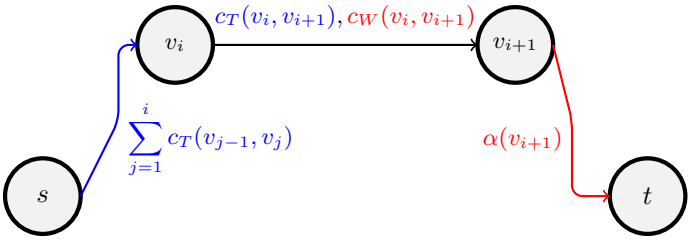

Fig. 2. Correct semi-adaptive routes: if typical delays have been experienced up to v_i, then we can get to t within a duration $\left(\sum_{j=1}^{i} c_T(v_{j-1}, v_j) + c_W(v_i, v_{i+1}) + \alpha(v_{i+1})\right)$ of having left s

Intuitively, if an entity traversing the network reaches vertex v with remaining end-to-end delay of $\alpha(v)$, then it is "safe" in that there exists a path from v to t that allows us to meet the end-to-end deadline.

Now we can define the correctness criterion for adaptive routing strategies. Let us start with semi-adaptive routing.

Definition 2 (Correct semi-adaptive route): Let $p = \langle s \equiv v_0, v_1, v_2, \ldots, v_k \equiv t \rangle$ denote a route of feasible instance $\langle G, c_T, c_W, s, t, D \rangle$ that was taken during some traversal from s to t. This route is said to be a **correct semi-adaptive route** for instance $\langle G, c_T, c_W, s, t, D \rangle$ if

$$\forall\, i : 0 \leq i < k : \left(\sum_{j=1}^{i} c_T(v_{j-1}, v_j) + c_W(v_i, v_{i+1}) + \alpha(v_{i+1}) \leq D\right)$$

□

Here is the intuition (see Figure 2): Say that we have reached vertex v_i without experiencing delay larger than the typical delay on any edge from s to v_i. The first term in the inequality denotes the estimate of the delay so far (since a semi-adaptive strategy does not know the exact delay and must assume that, in the worst case, the delay on every edge so far was exactly its typical estimate). The second term denotes the maximum delay we may experience in traversing the edge (v_i, v_{i+1}), and the third term, the maximum delay we will experience if we were to traverse the path $\mathcal{W}(v_{i+1})$ from v_{i+1} to the destination t. The condition requires that the sum of these terms not exceed the specified end-to-end delay bound; if it is satisfied, we may conclude that it is safe to take the edge (v_i, v_{i+1}) out of vertex v_i since it remains possible upon doing so to get to the destination vertex t within the specified end-to-end delay bound even if the worst-case delay is encountered while traversing this edge and all subsequent edges *en route* to the destination.

An optimal semi-adaptive strategy never reaches an unsafe state while guaranteeing that the typical delay of the path is minimized whenever *the actual delay that is experienced*

across each edge during this traversal does not exceed the typical delay estimate for that edge. Note that in the case where the actual delay on any edge exceeds the typical estimate, all we guarantee is that the end-to-end delay bound will be met. Based on the definitions, for semi-adaptive strategies we observe that for a particular problem instance, an optimal instance needs to compute a single path $\mathcal{P}$ — the path with the smallest typical delay among all correct semi-adaptive paths. During runtime, we traverse $\mathcal{P}$ as long as all edge delays do not exceed their respective typical estimates. As soon as the delay on any edge, say (v_i, v_{i+1}) exceeds its typical estimate, we then then take the path $\mathcal{W}(v_{i+1})$ from v_{i+1} to t.

The correctness condition of a fully-adaptive route is similar to semi-adaptive correctness, except that it considers the actual delay encountered on each edge. Let $\mathit{delay}(u, v)$ denote the actual delay that was experienced on edge (u, v) in a particular traversal.[1]

Definition 3 (Correct adaptive route): Let $p = \langle s \equiv v_0, v_1, v_2, \ldots, v_k \equiv t \rangle$ denote a route of feasible instance $\langle G, c_T, c_W, s, t, D \rangle$ that was taken during some traversal from s to t. This route is said to be a **correct adaptive route** for instance $\langle G, c_T, c_W, s, t, D \rangle$ if

$$\forall\ i : 0 \leq i < k : \left(\sum_{j=1}^{i} \mathit{delay}(v_{j-1}, v_j) + c_W(v_i, v_{i+1}) + \alpha(v_{i+1}) \leq D \right)$$

An optimal adaptive strategy is one that minimizes, at every intermediate vertex v_i, the sum of the actual delay encountered so far from s to v_i and the typical delay of the future path from v_i to t while guaranteeing correctness. Since the optimality and correctness depends on the actual delays experienced during the traversal, it is not sufficient to compute a single path — the correct optimal path may change based on how much delay has been experienced so far in a particular traversal.

III. The approach of [1]

As mentioned above, the problem we are studying in this paper was introduced in [1]. Given a feasible instance $\langle G, c_T, c_W, s, t, D \rangle$, the algorithm of [1] uses a dynamic-programming approach derived from the Bellman-Ford shortest-paths algorithm [3] to construct a *lookup table* at each vertex in the graph in a pre-processing phase before setting out from s. For our purposes the details of this pre-processing algorithm are unnecessary; it suffices to state that the lookup table at each vertex u contains entries of the form (d, v) where $d \in \mathbb{N}$ and (u, v) is an edge in the graph. The presence of the entry (d, v) at the vertex u has the following interpretation:

> *Of all the correct adaptive routes $p, u \overset{p}{\rightsquigarrow} t$, of the instance $\langle G, c_T, c_W, u, t, d \rangle$ in which the actual delay encountered across each edge e is exactly equal to the estimated typical delay $c_T(e)$ of that edge, the one for which $c_T(p)$ is minimized has (u, v) as its first edge.*

That is, the outgoing edge (u, v) should be taken in order to travel from u to the destination vertex t in the shortest time under typical circumstances, while simultaneously guaranteeing to arrive at t within a duration d of leaving v.

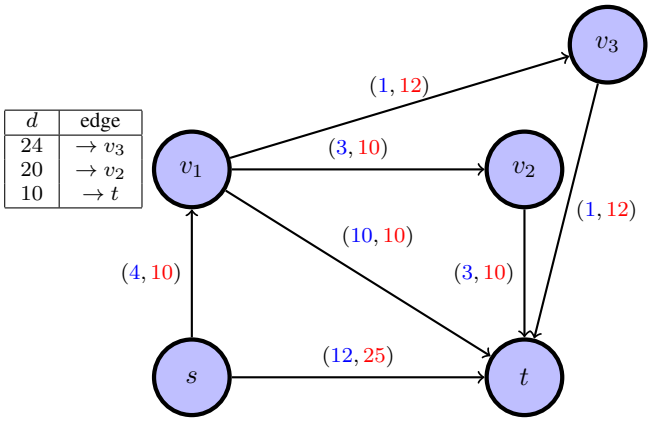

Fig. 3. The lookup table at vertex v_1 that is generated by the algorithm of [1] on the example instance that is depicted in Figure 1. Upon reaching vertex v_1, an entity would leave on the edge (v_1, v_3) if the remaining duration of its end-to-end delay bound is ≥ 24, on the edge (v_1, v_2) if this remaining duration is smaller than 24 but ≥ 20, and on the edge (v_1, t) if this remaining duration is less than 20 but ≥ 10. If the remaining duration is < 10 then no route exists that guarantees to reach the destination vertex t on time.

The lookup table at the vertex v_1 for the graph in Figure 1 is depicted in Figure 3 – its use is explained in the caption of the figure.

It was shown in [1] that (i) the execution time of the pre-processing algorithm to constructs these lookup tables is polynomial in the cumulative size (i.e., number of rows) of the lookup tables on all the vertices; and (ii) at each intermediate vertex, the time to decide which edge to follow next is logarithmic in the number of rows of the lookup table at the vertex. In Section IV we will show that the number of rows in the lookup table at a vertex may be larger than polynomial in the representation of the instance; it therefore follows that the preprocessing time (and consequently, also the overall running time) of this approach is not polynomial.

IV. The Approach of [1] has Super-Polynomial Time-Complexity

We consider the main result of this section —that the lookup tables at individual vertices may contain super-polynomially many rows— to be very counter-intuitive; our expectation was that the number of rows in the lookup table at a vertex would not exceed the out-degree of (number of outgoing edges from) the vertex. To be more specific, we did not expect that the same out-going edge from a vertex would turn out to be the best one to take for non-overlapping ranges of the remaining worst-case delay bound. However, our intuition turned out to

[1] We assume that $\mathit{delay}_j \leq c_W(v_{i-1}, v_i)$ for all j: we are not required to meet the end-to-end deadline if this condition is violated.

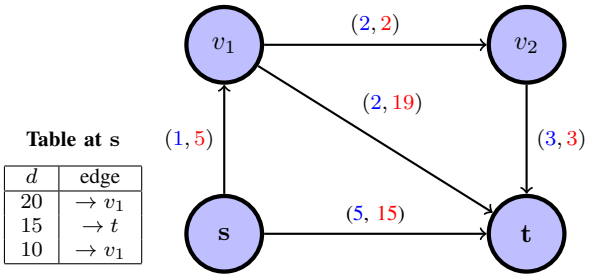

Fig. 4. An example used in Section IV, illustrating the intuition that large look-up tables may be necessary.

be incorrect; Figure 4, shows an example graph along with the lookup table at the vertex **s**:

- If the end-to-end delay bound is at least 20 time units, then the edge $\mathbf{s} \to v_1$ should be taken, since it is safe and under typical conditions the path $\mathbf{s} \to v_1 \to \mathbf{t}$ yields a typical delay of $1+2=\mathbf{3}$.
- If the end-to-end delay bound is in the range $[10, 15)$, then the only correct option is $\mathbf{s} \to v_1 \to v_2 \to \mathbf{t}$, for a typical delay of $1+2+3=\mathbf{6}$.
- However, if the delay bound is in the range $[15, 20)$, then the direct edge $\mathbf{s} \to \mathbf{t}$ yields a typical delay of **5** (while the alternative path, $\mathbf{s} \to v_1 \to v_2 \to \mathbf{t}$, has typical delay of $1+2+3=6$).

Thus, the edge $\mathbf{s} \to v_1$ is optimal for the non-overlapping range of delay-bound values $[10, 15)$ and $[20, \infty)$, while the edge $\mathbf{s} \to v_1$ is optimal for interval $[15, 20)$ that lies between these two ranges. Consequently the number of rows in the lookup table exceeds the number of outgoing edges.

We now generalize the example above to carry the intuition through to its logical conclusion.[2] Figure 5 shows an example graph for which the table size at a vertex is actually super-polynomial in the number of vertices of the graph. The graph consists of $\sqrt{n}$ layers numbered from 0 to $\sqrt{n}-1$. Layer i consists of one *switching node* u_i and $\sqrt{n}$ *fan out* nodes $v_{i,0}$ to $v_{i,\sqrt{n}-1}$. The switching node u_i is connected to the fan out node $v_{i,j}$ as follows: $c_T(u_i, v_{i,j}) = j \times \sqrt{n}^i$ and $c_W(u_i, v_{i,j}) = k_i - c_T(u_i, v_{i,j})$ where $k_i = 2(\sqrt{n}^{i+1} - \sqrt{n}^i)$. For instance, in layer 0, we have $c_T(u_0, v_{0,j}) = j$ and $c_W(u_0, v_{0,j}) = 2\sqrt{n} - 2 - j$. All the fan out nodes of layer i are connected to the switching node of layer $i+1$ with both typical and worst case costs of 0. The fan out nodes of the last layer are connected to the sink t with edges of weight $(0, 0)$.

Intuition for large tables. We will show that switching nodes have large tables. In particular, we will show that the table at switching node u_i can have size $\sqrt{n}^{\sqrt{n}-i}$. We first state some observations that are clear from the figure. The following observation follows from the fact that there are $\sqrt{n}$ layers and there are $\sqrt{n}$ independent paths in each layer.

[2]This generalization is quite technical and detailed, and the remaining sections of this paper do not assume an understanding of these details; it may therefore safely be skipped at a first reading.

Observation 1: The number of paths from u_i to u_j is $\sqrt{n}^{j-i}$. Therefore, the number of paths from u_0 to t is $\sqrt{n}^{\sqrt{n}}$.

The following observation says that the edges are well-behaved in that their typical delays are not larger than their worst case delays.

Observation 2: For each edge, we have $c_W(e) \geq c_T(e)$.

Proof: Edges from fanout nodes to switching nodes have both c_W and c_T equal to 0 — so this is trivially true. Consider any layer i and the edges from its switching node to fanout nodes. The edge with the largest c_T and the smallest c_W is the edge $(u_i, v_{i,\sqrt{n}-1})$. For this edge, $c_T(u_i, v_{i,\sqrt{n}-1}) = (\sqrt{n}-1)(\sqrt{n}^i)$ and $c_W(u_i, v_{i,\sqrt{n}-1}) = k_i - (\sqrt{n}-1)(\sqrt{n}^i) = 2(\sqrt{n}^{i+1} - \sqrt{n}^i) - (\sqrt{n}^{i+1} - \sqrt{n}^i) = c_T(u_i, v_{i,\sqrt{n}-1})$. For all other edges in the layer, $c_T < c_W$. □

This observation states the relationship between typical and worst case delays of paths between switching nodes.

Observation 3: For any path p that goes from u_0 to u_j, $c_W(p) = \sum_{\ell=1}^{i-1} k_i - c_T(p) = 2\sqrt{n}^i - 2 - c_T(p)$. Therefore, for any path from u_0 to t, we have $c_W(p) = 2\sqrt{n}^{\sqrt{n}} - 2 - c_T(p)$.

We can use these observations to understand the intuition as to why the switching nodes need large tables. Let us consider the particular case of u_0

1) Each path from u_0 to t has a distinct typical delay between 0 and $\sqrt{n}^{\sqrt{n}} - 1$ (proven formally in Lemma 1).
2) For any path from u_0 to t, Observation 3 tells us that the worst case delay is $c_W(p) = \sum_{\ell=1}^{i-1} k_i - c_T(p)$ where $\sum_{\ell=1}^{i-1} k_i = 2\sqrt{n}^{\sqrt{n}} - 2$. Therefore, all worst case delays are also distinct and take values between $\sqrt{n}^{\sqrt{n}} - 1$ and $2\sqrt{n}^{\sqrt{n}} - 2$. But their ordering is opposite of typical delays — for any pair of paths p_1, p_2, if $c_T(p_1) > c_T(p_2)$, then $c_W(p_1) < c_W(p_2)$.
3) On careful observation, we see the following for paths from u_0 to t: a path p with $c_T(p) = j \mod \sqrt{n}$ must go through $v_{0,j}$ since all typical weights except the weights of fan out edges in the first layer are multiples of $\sqrt{n}$. Therefore, the first edge on the path from u_0 to t decides the remainder when the path weight is divided by $\sqrt{n}$.
4) Now let us consider the lookup table at u_0. If the remaining end-to-end delay bound is smaller than $\sqrt{n}^{\sqrt{n}} - 1$, then this graph is infeasible. If the remaining end-to-end delay bound is $\sqrt{n}^{\sqrt{n}} - 1$, this packet is forced to take the bottom path through $v_{0,\sqrt{n}-1}, v_{1,\sqrt{n}-1}, \ldots$ in Figure 5 with the typical delay bound of $\sqrt{n}^{\sqrt{n}} - 1$ since this is the only path with the required worst case delay bound. On the other hand, if the remaining end-to-end delay bound at u_0 is at least $2\sqrt{n}^{\sqrt{n}} - 2$, then the packet can take the top path through $v_{0,0}, v_{1,0} \ldots$ for the typical delay bound of 0. All end-to-end delays between these quantities, there is a unique path that has that particular worst case bound.
5) Recall that paths with smaller worst case delay have larger typical delays. For any particular remaining end-to-end delay bound of Y at u_0, to get the smallest typical delay, we should choose the path p such that $c_W(p) = Y$ and

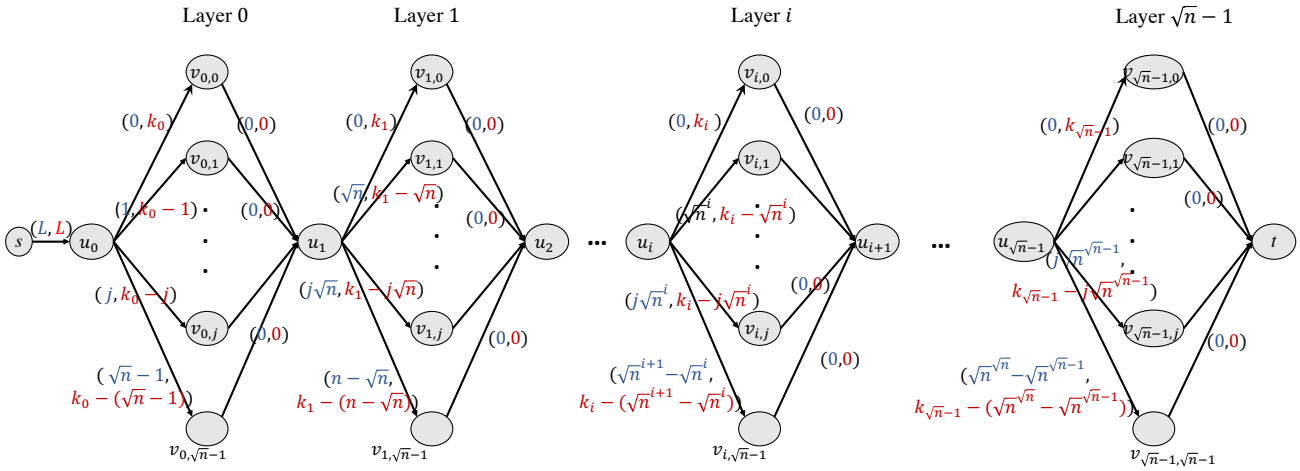

Fig. 5. The example to show that we need pseudopolynomial size tables. The example is a layered graph with $\sqrt{n}$ layers where each layer has a switching node u_i and $\sqrt{n}$ fan-out nodes. Each edge is labeled with its typical and worst-case delays as c_T, c_W.

therefore, $c_T(p) = 2\sqrt{n}^{\sqrt{n}} - 2 - Y$. This path p will go through vertex $v_{0,j}$ if $c_T(p) = j \mod \sqrt{n}$.

To be even more concrete of when we might store a large table, consider the case where the source node is connected to u_0 with an edge with $c_T(s, u_0) = c_W(s, u_0) = L = \sqrt{n}^{\sqrt{n}} - 1$ and the end-to-end delay deadline between s and t is $D = 2\sqrt{n}^{\sqrt{n}} - 2$. Therefore, the packet may experience any delay between 0 and L on the link (s, u_0) and therefore, its remaining end to end budget can be anywhere between $D - L = \sqrt{n}^{\sqrt{n}} - 1$ and $D = 2\sqrt{n}^{\sqrt{n}} - 2$. Table I shows the $\sqrt{n}^{\sqrt{n}}$ table we must store at u_0 in this case. The left column d is the maximum worst case delay we can tolerate and the second column indicates the edge it must take to get the smallest typical delay given this worst case delay. As the third column, (just for illustration) the table also shows the expected typical delay for the particular value of d.

This table has $\sqrt{n}^{\sqrt{n}}$ rows at u_0. Since the total number of nodes in the network is $O(n)$ ($n + \sqrt{n} + 2$ to be exact), the table is super-polynomial in the size of the network. It is still pseudo-polynomial since it can not be larger than D or the sum of all the worst-case delays. All switching nodes are similar — u_i must store $\sqrt{n}^{\sqrt{n}-i}$ rows since there are $\sqrt{n}^{\sqrt{n}-i}$ paths from u_i to t.

d	edge	expected $c_T(p)$
$D(= 2\sqrt{n}^{\sqrt{n}} - 2)$	$\to v_{0,0}$	0
$D - 1$	$\to v_{0,1}$	1
$D - 2$	$\to v_{0,2}$	2
$\vdots$	$\vdots$	$\vdots$
$D - (\sqrt{n} - 1)$	$\to v_{0,\sqrt{n}-1}$	$\sqrt{n} - 1$
$D - \sqrt{n}$	$\to v_{0,0}$	$\sqrt{n}$
$D - (\sqrt{n} + 1)$	$\to v_{0,1}$	$(\sqrt{n} + 1)$
$\vdots$	$\vdots$	$\vdots$
$D - (2\sqrt{n} - 1)$	$\to v_{0,\sqrt{n}-1}$	$2\sqrt{n} - 1$
$\vdots$	$\vdots$	$\vdots$
$D - (\sqrt{n}^{\sqrt{n}} - \sqrt{n})$	$\to v_{0,0}$	$\sqrt{n}^{\sqrt{n}} - \sqrt{n}$
$D - (\sqrt{n}^{\sqrt{n}} - \sqrt{n} + 1)$	$\to v_{0,1}$	$\sqrt{n}^{\sqrt{n}} - \sqrt{n} + 1$
$\vdots$	$\vdots$	$\vdots$
$D - (\sqrt{n}^{\sqrt{n}} - 1)$	$\to v_{0,\sqrt{n}-1}$	$(\sqrt{n}^{\sqrt{n}} - 1)$

TABLE I
THE LOOKUP TABLE AT u_0 FOR THE EXAMPLE SHOWN IN FIGURE 5

Rigorous proof for large tables. We now do the rigorous proof. Again, relatively obvious facts are stated as observations.

Observation 4: For any path p between u_i and u_j, $c_T(p)$ is a multiple of $\sqrt{n}^i$.

Proof: All edge weights between u_i and u_j are products of $\sqrt{n}^i$. Therefore, path weights must also be multiples of $\sqrt{n}^i$. □

This key lemma shows that all paths have unique weights.

Lemma 1: No two paths between u_i and u_j (for all i and all $j > i$) have the same typical weight (therefore worst-case weight). In particular, there are $\sqrt{n}^{j-i}$ paths between u_i and u_j and each of these paths has $c_T(p) = x\sqrt{n}^i$ for different values of x between 0 and $\sqrt{n}^{j-i} - 1$.

Proof: Proof is by induction. As a base case, consider a single layer. From Figure 5, for all i, we see that the $\sqrt{n}$ paths between u_i and u_{i+1} have unique typical delays that take values $0, \sqrt{n}^i, 2\sqrt{n}^i, ..., (\sqrt{n} - 1)\sqrt{n}^i$.

Assume for inductive hypothesis that the statement is true for paths between u_i and u_j. We must now prove it for paths between u_i and u_{j+1}. Note that the $\sqrt{n}$ paths between u_j and u_{j+1} have unique weights and they take values $0, \sqrt{n}^j, 2\sqrt{n}^j, ..., (\sqrt{n} - 1)\sqrt{n}^j$ (from base case).

From the inductive hypothesis, each path from u_i to u_j takes a unique weight among $0, \sqrt{n}^i, 2\sqrt{n}^i, ..., (\sqrt{n}^{j-1} - 1)\sqrt{n}^i$. Consider path $p_x^{i,j}$ with weight $x\sqrt{n}^i$ ($0 \le x \le \sqrt{n}^{j-i} - 1$) and path $p_y^{j,j+1}$ with weight $y\sqrt{n}^j$ ($0 \le y \le \sqrt{n} - 1$). If we

concatenate these paths, we get a path of length $x\sqrt{n}^i + y\sqrt{n}^j$. Since $x \leq \sqrt{n}^{j-i} - 1$, the first term $x\sqrt{n}^i \leq \sqrt{n}^j - \sqrt{n}^i$. Therefore, the sum of the two terms are unique for all values of x and y.

In addition, by Observation 4, all these paths lengths must be multiples of $\sqrt{n}^i$. Finally, for the largest values of x and y, we have $x\sqrt{n}^i + y\sqrt{n}^j = \sqrt{n}^{j+1} - \sqrt{n}^i = (\sqrt{n}^{j+1-i} - 1)\sqrt{n}^i$. Therefore, path from u_i to u_{j+1} has $c_T(p) = x\sqrt{n}^i$ for different values of x between 0 and $\sqrt{n}^{j+1-i} - 1$. □

Corollary 1: There are $\sqrt{n}^{\sqrt{n}}$ paths between u_0 and t and they all have unique typical weights between 0 and $\sqrt{n}^{\sqrt{n}}$.

Lemma 2: If we consider the paths from u_0 to t — all paths with typical weight $j \mod \sqrt{n}$ go through $v_{0,j}$ for all $0 \leq j \leq \sqrt{n} - 1$.

Proof: From Lemma 1, paths from u_1 to t have weights $0, \sqrt{n}, 2\sqrt{n}, ...,$. These values are all $0 \mod \sqrt{n}$. Therefore, in order to get a weight of $j \mod \sqrt{n}$, we must add a path of weight j between u_0 and u_1 and the only way to do that is to go through $v_{0,j}$. □

Now say a packet X delay on edge (s, u_0) and has the remaining end-to-end delay bound of $Y = D - X$, where X can take any value between 0 and L. Therefore, at u_0 it must decide what path to take to get the minimum typical delay.

Lemma 3: If we need the worst case delay from u_0 to t to be less than some quantity $Y = D - X$ for $0 \leq X \leq L$, then we should take the path which will go though fan out node $v_{0,j}$ if $X = j \mod \sqrt{n}$.

Proof: Observation 2 tells us that a path with worst case delay $c_W(p)$ has average delay $c_T(p) = 2\sqrt{n}^{\sqrt{n}} - 2 - c_W(p)$. Therefore, among all paths that have worst case delay at most $D - X$, the one with the smallest typical delay would be the one with typical delay $c_T(p) = 2\sqrt{n}^{\sqrt{n}} - 2 - D - X = X$ (since $D = 2\sqrt{n}^{\sqrt{n}} - 2$). Therefore, this packet should take the path with average delay X, which by Lemma 2 goes through $v_{0,j}$ if $X = j \mod \sqrt{n}$. □

Therefore, to get the best typical delay, we need a lookup table at u_0 that is as large as $X + 1$, which is $\sqrt{n}^{\sqrt{n}}$ in this case. Note that we can modify this example and change the number of layers and the number of fan-out nodes. For any value of ϵ, we can have n^ϵ fan-out nodes and $n^{1-\epsilon}$ layers with the total number of nodes still $O(n)$. (For our example, we have picked $\epsilon = 1/2$.) This will give us $(n^\epsilon)^{n^{1-\epsilon}}$ unique paths. Given a table of size $(n^\epsilon)^{n^{1-\epsilon}}$, at runtime, when we are node u_0, we need $n^{1-\epsilon} \lg n$ time to search for the best path using binary search. Therefore, the runtime overhead at intermediate nodes can be arbitrarily close to $n \lg n$ (at least at u_0). In Section VI, we will see an algorithm which takes polynomial pre-processing time and the runtime overhead at intermediate nodes is only slightly larger.

V. An Optimal Semi-Adaptive Algorithm

Recall from Section II that for semi-adaptive strategies, upon traversing an edge e the actual delay experienced in doing so is not revealed — we only know whether this actual delay exceeded the typical delay estimate $c_T(e)$ or not. Hence, an optimal strategy must minimize the typical delay encountered if no delay exceeds its typical estimate while ensuring that meeting the end-to-end delay bound never becomes impossible along the way. As mentioned in Section II, determining such an optimal strategy is equivalent to determining the single "safe" path $\mathcal{P}$ with minimum typical delay. I.e., the execution of the strategy proceeds as follows:

1) Prior to setting out from the source vertex, we will identify a path $\mathcal{P}$ from the source to the destination vertex – the manner in which we will do this is described below.
2) We then set out from the source vertex along this identified path $\mathcal{P}$. We continue to follow this path so long as the actual delay we encounter while traversing each edge does not exceed the typical delay estimate for that edge. If we reach the destination vertex, we are done.
3) Otherwise, let (u, v) denote the first edge in the path for which the actual delay encountered exceeds $c_T(u, v)$. We then travel along a potentially different path $\mathcal{W}(v)$ from v to the destination vertex t, that was also pre-calculated in a manner discussed below.

Given a problem instance $I = \langle G, c_T, c_W, s, t, D \rangle$, we wish to find this path $\mathcal{P}$ such that we are guaranteed to reach the destination t within a duration D of leaving the source vertex s, so long as the delay on any edge e does not exceed the worst case guaranteed delay of $c_W(e)$. In addition $\mathcal{P}$ is a shortest path (in terms of typical delays) among all such safe paths.

Before we describe the algorithm, we need a few additional definitions. Recall from Definition 1 that for any vertex v, $\alpha(v)$ denotes the smallest value of the maximum delay we may experience in getting from v to the destination vertex t, and $\mathcal{W}(v)$ denotes a path on which this delay may be experienced. We point out that the $\alpha(v)$ values, as well as the paths $\mathcal{W}(v)$ that define them, may be determined very efficiently using standard shortest-path algorithms such as the one in [2], for which implementations with $O(|E|+|V| \log |V|)$ running time are known [4].

The next concepts we seek to define are inter-dependent; hence they are defined concurrently.

Definition 4 ($\beta : V \to \mathbb{N}$; useful edges; $\mathcal{T} : V \to \mathbb{N}$): For every vertex $v \in V$, define $\beta(v)$ as follows:

$$\begin{aligned} \beta(s) &= 0 \quad \text{(Recall that } s \text{ denotes the source vertex)} \\ \beta(v) &= \text{minimum sum of typical-delay costs of the edges on a path from } s \text{ to } v \text{ in which all the edges are } \underline{\text{useful}} \end{aligned}$$

where an edge (x, y) is defined to be **useful** if and only if

$$\beta(x) + c_W(x, y) + \alpha(y) \leq D$$

We let $\mathcal{T}(v)$ denote the path from s to v consisting only of useful edges, that realizes the typical delay of $\beta(v)$. □

We will now show that the path $\mathcal{P}$ that we wish to identify prior to setting out from s is the path $\mathcal{T}(t)$. In the following two lemmas, we show that (i) the choice of $\mathcal{T}(t)$ as $\mathcal{P}$ guarantees that we will always reach the destination within D

time of leaving the source if we follow the procedure described above; and (ii) $\mathcal{T}(t)$ is the shortest such safe path.

Lemma 4: The choice of $\mathcal{T}(t)$ as $\mathcal{P}$ guarantees that we will reach the destination within a duration D of leaving the source vertex s.

Proof: Let us consider the path taken in an individual instance. First consider the case where the delay experienced across no edge on the path exceeded the typical delay estimate of that edge. Then the algorithm will take the path $\mathcal{P}$ with cost $\beta(t)$. Say that the last edge on the path was (v,t). By definition of $\mathcal{P}$, it is a useful edge. Therefore, we have $\beta(t) = \beta(v) + c_T(v,t) \leq \beta(v) + c_W(v,t) \leq D$ by definition of useful edges.

Next, consider the case where the delay on some edge e exceeds the typical estimate $c_T(e)$, and (u,v) denote the first such edge. Then, according to the procedure outlined above, the overall path taken is the prefix of $\mathcal{P}$ until v and then $\mathcal{W}(v)$. Therefore, the delay experienced is the at most the sum of the typical delay until u, $c_W(u,v)$ and $\alpha(v)$. Since any prefix of a shortest path is also a shortest path, the delay experienced until u is $\beta(u)$. Therefore, the total delay is $\beta(u) + c_W(u,v) + \alpha(v)$ which is at most D since (u,v) is a useful edge by virtue of being on path $\mathcal{T}(t)$. □

Lemma 5: Given the form of the solution where we take a single path $\mathcal{P}$ until some edge exceeds the typical delay, choosing any path with shorter typical delay than $\mathcal{T}(t)$ as the choice for $\mathcal{P}$ is unsafe. That is, if we chose a shorter path as $\mathcal{P}$, then the deadline may not be met.

Proof: $\mathcal{T}(t)$ is the shortest path from s to t using only useful edges. Therefore, any shorter path must use at least one edge which is not useful. Consider such a path X which uses a non-useful edge, and let (u,v) denote the first such edge on X. That is, all edges on path X from s to u are useful. By definition, since (u,v) is not useful, $\beta(u) + c_W(u,v) + \alpha(v) > D$. We now show a circumstance which results in a violation of the end-to-end delay bound if we choose X as $\mathcal{P}$. The edges on the prefix of the path until u will experience typical delays, causing us to stay on X and incurring a delay of $\beta(u)$ since all edges on the path from s to u are useful. At this point, the algorithm will choose the edge (u,v) which will experience the worst-case delay $c_W(u,v)$. After this, all edges on whichever path is chosen will also experience its worst-case delay — hence we can not get from v to t in time less than $\alpha(v)$. Therefore, the total delay experienced is $\beta(u) + c_W(u,v) + \alpha(v) > D$ thereby violating the delay bound. □

From Lemmas 4 and 5, we conclude that it is sufficient to find $\mathcal{T}(t)$ in order to solve the optimal semi-adaptive routing problem. Figure 6 provides an algorithm to calculate $\beta(t)$ and $\mathcal{T}(t)$; in the remainder of this section we prove its correctness and derive bounds on its running time. Before doing so, we first make an obvious observation.

Observation 5: Each edge is (u,v) is considered for relaxation exactly once — when the vertex u (the originating vertex of the edge) is extracted from Q. □

Correctness of the algorithm of Figure 6 is proved in the following theorem.

```
1  for each vertex v ∈ V
2      DIST(v) = ∞
3  DIST(s) = 0
   // Q is a priority queue prioritized by the DIST(·) values
4  for each vertex v ∈ V
5      Q.INSERT(v)
6  while Q is not empty
7      u = Q.EXTRACTMIN()
8      for each edge (u, v) such that v ∈ Q
9          if ( (DIST(u) + c_W(u, v) + α(v) ≤ D) and
               (DIST(u) + c_T(u, v) < DIST(v)) )
               // "relax" this edge
10             DIST(v) = DIST(u) + c_T(u, v)
11             PARENT(v) = u
```

To find $\mathcal{T}(t)$, we trace back the parent pointers from the destination node t until we reach the source.

Fig. 6. Pseudo-code for optimal semi-adaptive algorithm. It calculates the path $\mathcal{T}(t)$. The final values of DIST(u) for all nodes u represent $\beta(u)$.

Theorem 1: For the algorithm of Figure 6

1) DIST(u) is always an upper bound on the $\beta(u)$ (i.e., DIST$(u) \geq \beta(u)$ at all times);
2) when a node u is extracted from Q, DIST$(u) = \beta(u)$; and
3) an edge is relaxed if and only if it is useful.

Proof: We will prove this by induction on the number of iterations of the **while** loop. As the base case, we know that DIST$(s) = 0$ and DIST$(u) = \infty$ for all other u — therefore, in the beginning Property 1 is satisfied. The first node extracted by the algorithm is s and DIST$(s) = \beta(u) = 0$ at this time — satisfying Property 2. Finally, consider an outgoing edge (s,v) from s. This edge is relaxed iff DIST$(s) + c_W(s,v) + \alpha(v) \leq D$. Therefore, these edges are relaxed iff they are useful, satisfying Property 3.

We will now induct on the extraction of vertices and relaxation of edges. As the inductive hypothesis (IH), assume that right before vertex u is extracted, all three properties were satisfied for all vertices and edges. We must now show that until the next vertex is extracted, all properties are still satisfied.

We first show that Property 2 holds about u when u is extracted. Let us assume, for a contradiction, that DIST$(u) \neq \beta(u)$ when u is extracted. By the IH on Property 1, we know that, right before extraction, we have DIST$(u) \geq \beta(u)$. Since DIST(u) never increases, and we assumed that DIST$(u) \neq \beta(u)$, we must have DIST$(u) > \beta(u)$ when u is extracted.

Consider path $\mathcal{T}(u)$ — the actual shortest path from s to v consisting of only useful edges. This path has length $\beta(u)$.

On this path, consider the first node, say x, that is still in Q, and the node immediately before x on the path, say w. Therefore, w is no longer in Q and was extracted during some previous iteration. Hence, by the inductive hypothesis and Property 2, DIST$(w) = \beta(w)$. Now consider edge (w,x).

This edge was considered for relaxation when we extracted w. Since $\mathcal{T}(u)$ consists of only useful edges, this edge is useful. Therefore by IH on Property 3, this edge was relaxed when w was extracted.

When this edge was relaxed, the algorithm set $\text{DIST}(x) = \text{DIST}(w) + c_T(w,x) = \beta(w) + c_T(w,x)$. Since $\mathcal{T}(u)$ is the shortest path from s to u with only useful edges, and all edge weights are positive, $\mathcal{T}(u)$'s prefix from s to x is $\mathcal{T}(x)$ (by what is commonly referred to as the *inclusion property* of shortest paths: every sub-path of a shortest path is a shortest path between its first and last vertices). Therefore, after edge (w,x) is relaxed $\text{DIST}(x) = \beta(x)$.

Therefore, if $x = u$, we have a contradiction since we assumed that $\text{DIST}(u) > \beta(u)$ when u is extracted and $\text{DIST}(u)$ never increases. If $x \neq u$, then x comes before u on $\mathcal{T}(u)$. Therefore, we know that $\text{DIST}(x) = \beta(x) \leq \beta(u) < \text{DIST}(u)$. Since x and u are both in Q at this point, we would extract x before we extract u— which is also a contradiction since we are currently extracting u before x Therefore, when a vertex u is extracted, $\text{DIST}(u) = \beta(u)$ (Property 2)

Now consider Property 3. Edge (u,v) is relaxed when u is extracted if $\text{DIST}(u) + c_W(u,v) + \alpha(v) \leq D$ which implies that $\beta(u) + c_W(u,v) + \alpha(v) \leq D$. This is the definition of usefulness — therefore, the edge will be relaxed iff it is useful.

For property 1, consider a vertex v. $\text{DIST}(v)$ is the typical length of the path that is realized by the current parent pointers from s to v. Before u was extracted, $\text{DIST}(v) \geq \beta(v)$ by IH. $\text{DIST}(v)$ changes only if we relaxed the edge (u,v). If edge (u,v) is relaxed, the algorithm sets $\text{DIST}(v) = \text{DIST}(u) + c_T(u,v) = \beta(u) + c_T(u,v)$. We know that the entire path $\mathcal{T}(u)$ that realizes $\beta(u)$ has only useful edges and (u,v) is useful. Therefore, this new $\text{DIST}(v)$ which is the length of the path $\mathcal{T}(u)$ extended by (u,v) also only consists of only useful edges. Therefore, $\text{DIST}(v)$ is the length of the path $\mathcal{T}(u) \cup (u,v)$, which is a path from s to t consisting of useful edges. Since $\beta(v)$ is the length of the shortest path from s to v consisting of useful edges, we must have $\text{DIST}(v) \geq \beta(v)$. □

Therefore, after the algorithm completes, $\text{DIST}(t) = \beta(t)$ and therefore, the parent pointers identify the path $\mathcal{T}(t)$. Recall that Lemmas 4 and 5 guarantee that $\mathcal{T}(t)$ is the optimal semi-adaptive path. We now prove the performance bound.

Theorem 2: The pre-processing time is $O(E + V \lg V)$.

Proof: First, we calculate $\mathcal{W}(v)$ for all vertices v by running Dijkstra's algorithm from t using the worst case delays for running time of $O(E + V \lg V)$. The algorithm consists of $|V|$ INSERT operations, $|V|$ EXTRACTMIN operations and E DECREASEKEY operations. DECREASEKEY operations occur when an edge is relaxed and $\text{DIST}(v)$ is changed. According to Observation 5 each edge is relaxed exactly once, for at E DECREASEKEY operations overall. Just as for normal Dijkstra's algorithm, we can use Fibonacci Heaps to implement the priority queue with amortized costs of $O(1)$, $O(\log n)$, and $O(1)$ respectively for INSERT, EXTRACTMIN and DECREASEKEY respectively [4], giving us the total running time of $O(E + V \lg V)$. □

In addition, at each intermediate vertex, the runtime overhead is constant for the overall running time of $O(E+V \lg V)$.

VI. Optimal Adaptive Routing in Polynomial Time

As mentioned in Section III, the pre-processing time for the adaptive routing approach of [1] is polynomial in the size of the lookup tables, and the running time overhead at each intermediate vertex is logarithmic in the size of the lookup table at that vertex. The result of Section IV shows that lookup table sizes (and hence, the pre-processing time) can be super-polynomial; therefore, the per-vertex run-time overhead during traversal can be close to linear in the representation of the instance.

We saw in Section V that the optimal semi-adaptive algorithm has polynomial pre-processing time as well as constant overhead at runtime per vertex. We will now extend this algorithm to get an optimal fully-adaptive algorithm which has the following properties:

1) the pre-processing phase is identical to the semi-adaptive algorithm and has polynomial running time.
2) the running time at each intermediate vertex during traversal is also polynomial in the representation of the instance (in this regards it is hence slower than the semi-adaptive algorithm which has constant per-vertex complexity, but comparable to the approach of [1]); and
3) the end-to-end delays are identical to the ones that are obtained by the approach of [1]and hence optimal.

We now proceed to describe our proposed approach. We start out with an informal observation that we will formalize and prove a bit later (as Lemma 7): *the first edge taken by an optimal semi-adaptive algorithm is also the first edge taken by the optimal fully-adaptive algorithm (on the same instance)*. This observation is not surprising: informally speaking, this happens because no additional ("on-line") information, in the form of the actual delays experienced upon traversing an edge, has been revealed prior to leaving the source vertex s, and both approaches are making the best use of the same pre-runtime information to choose the same edge by which to leave s.

Of course, the actual delay encountered while traversing this first edge may be different from its estimated typical delay. Let (s, v_1) denote this first edge out of the source vertex s, and let $\textit{delay}(s, v_1)$ denote the actual delay that was encountered across this edge (s, v_1). If $\textit{delay}(s, v_1) \neq c_T(s, v_1)$, the route $v_1 \rightsquigarrow t$ taken by the a semi-adaptive algorithm may differ from the one taken by an optimal adaptive algorithm, since the fully-adaptive algorithm takes account of the magnitude of the difference between $\textit{delay}_1$ and $c_T(s, v_1)$ in order to inform its future routing decisions, while a semi-adaptive algorithm does not have this information.

The crucial observation that underpins our fully-adaptive approach is this: upon having reached the vertex v_1, the value of $\textit{delay}(s, v_1)$ becomes known, and it can hence be concluded that there is a duration $(D - \textit{delay}(s, v_1))$ remaining of the end-to-end delay bound. We can therefore look upon the problem

Let $\langle G, c_T, c_W, s, t, D\rangle$ denote a feasible instance we seek to solve. That is, we seek to travel from the source s to the destination t with minimum delay subject to a worst-case delay bound D

1 $source = s$
2 $D_o = D$
3 Call semi-adaptive algorithm on $\langle G, c_T, c_W, source, t, D_0\rangle$
4 Say (v_0, v_1) is the first edge of the returned path.
5 Traverse this edge; Say actual delay is $delay(v_0, v_1)$
6 **if** $(v_1 \equiv t)$ **exit** // Destination has been reached
7 **if** $delay_1 > c_W(v_0, v_1)$ **then exit**
// Worst-case delay bound on edge exceeded
8 $D_o = D_o - delay_1$ // Remaining end-to-end delay
9 $source = v_1$ // The new source vertex
10 **goto** Line 3

Fig. 7. Pseudo-code representation of an optimal adaptive routing strategy that has polynomial time-complexity

of getting from v_1 to the designated destination t as a *fresh instance* of the problem:

$$\Big\langle G, c_T, c_W, v_1, t, (D - delay_1)\Big\rangle \tag{1}$$

which requires us to get from this vertex v_1 to t with minimum typical delay while guaranteeing to do so within a worst-case delay of $(D - delay_1)$. And this fresh problem can be solved in polynomial time by applying the semi-adaptive pre-processing algorithm. Using an argument that is essentially identical to the one used to show that the first edge in the paths for semi-adaptive and adaptive algorithms are identical, it follows that the second edge selected by an optimal fully-adaptive algorithm is identical to the first edge selected by a semi-adaptive algorithm on the instance represented in Expression 1 above. Therefore, at every intermediate vertex v_i, we make a call to semi-adaptive algorithm on a new problem instance with the source replaced with v_1 and the end-to-end delay bound replaced with D minus the delay experienced so far.

This idea is represented in pseudo-code form in Figure 7. (The repeated calls to the semi-adaptive algorithm are made in Line 3.) We will now show that this strategy is **correct**:

Lemma 6: The strategy of Figure 7 always generates a correct adaptive route on a feasible instance, provided all actual delays that are experienced during run-time do not exceed the respective worst-case bounds.

Proof: We show that if the initial instance is feasible, then each call to the semi-adaptive algorithm made in Line 3 of Figure 7 is on a feasible instance. This can be shown by induction on the number of times Line 3 is executed: the instance is clearly feasible the first time. Assuming for the induction hypothesis that the instance is feasible at the i'th call, since the actual delay that is encountered upon traversing the edge (Line 5 of the pseudo-code) is no larger than the corresponding worst-case bound (as checked in Line 7) it follows that the instance on which Line 3 is executed for the $(i+1)$'th time will also be feasible. □

Lemma 7 shows that the semi-adaptive algorithm and the optimal fully adaptive algorithm (from [1]) take the same outgoing edge from the source vertex for any feasible instance:

Lemma 7: Let $\mathcal{P} = \langle s \equiv v_0, v_1, v_2, \ldots, v_k \equiv t\rangle$ denote the path generated by semi-adaptive algorithm on a feasible instance $I = \langle G, c_T, c_W, s, t, D\rangle$. Edge (s, v_1) is also the first edge traversed by the approach of [1] on instance I.

Proof: The approach of [1] takes the edge (s, v_1) from s if there is an entry d, v_1 in the lookup table at s such that $d \leq D$ and this is the largest such d. Recall from Section III that an entry (d, v) in the lookup table at a vertex u denotes that the outgoing edge (u, v) is the one that should be taken in order to travel from u to the destination vertex t in the shortest time provided the delay experienced across each edge does not exceed the typical delay estimate for that edge, and we are guaranteed to be able to get to t within a duration d. From the correctness and optimality condition of semi-adaptive algorithms in Section V, this is exactly the path that is identified by the semi-adaptive algorithm. Hence the first edge on both paths is the same. □

Repeated application of Lemma 7, one for each edge that is traversed, immediately yields the following result:

Theorem 3: The $s \overset{t}{\rightsquigarrow}$ path generated by the Algorithm depicted in Figure 7 on a feasible instance $\langle G, c_T, c_W, s, t, D\rangle$ is identical to the path generated by the approach of [1].

Dealing with updates to $c_T(\cdot)$ values

While in our model the $c_W(\cdot)$ values are assumed to represent guaranteed upper bounds on the delays experienced while traversing across edges, the $c_T(\cdot)$ values represent *estimates* of the delays that are typically encountered. We now discuss how all three approaches — the one proposed in [1], the semi-adaptive algorithm, and the one in Figure 7 — could deal with more accurate estimates of these $c_T(\cdot)$ values if they were to become available after the traversing entity has started on its path.

- The approach of [1] would require that the lookup tables at all the vertices be recomputed which may be a non-polynomial time operation (Section III). Realistically speaking, one would probably ignore these fresh estimates, potentially compromising the optimality property.
- Since the semi-adaptive algorithm predetermines the entire route before leaving the source vertex, there are two reasonable strategies for dealing with updated $c_T(\cdot)$ estimates: ignore the updates (thereby compromising the optimality of the obtained route), or re-compute the route from the current location. But this latter alternative is exactly what the fully-adaptive strategy does anyway.
- The fully adaptive approach in Figure 7 only uses the estimated $c_T(\cdot)$ values to compute the *next* hop to take; once this hop has been taken, the subsequent hop is recomputed (again, using the estimated $c_T(\cdot)$ values). Hence if the $c_T(\cdot)$ estimates changed while traversing an edge, the approach of Figure 7 can automatically use the new estimates for the next and all subsequent hops. Therefore, this approach provides, in polynomial time, superior (i.e., smaller end-to-end delay

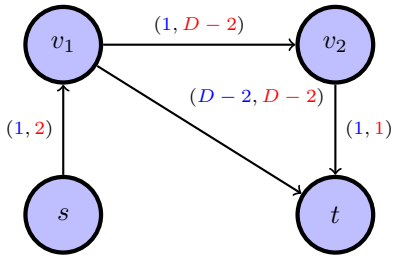

Fig. 8. An example feasible instance upon which static routing may perform arbitrarily poorly in comparison to adaptive routing.

under typical circumstances) routes than the approach of [1] if the approach of [1] does not change its lookup tables upon receiving fresh $c_T(\cdot)$ estimates.

VII. Shortcomings of Static Routing Strateries

In this paper, we have explored adaptive routing strategies, justifying (in Section I) the need to do so by arguing that they are in capable of providing shorter delays under typical conditions than static routes are. In this section, we formally show that static routing can perform poorly in comparison to adaptive routing strategies along both dimensions of comparison: **1.** determining optimal static routes is NP-hard while (as shown in Section VI) optimal adaptive routes can be determined in time polynomial in the problem size and the number of edges in the optimal adaptive path; and **2.** the ratio of the duration of the delay experienced under typical circumstances may be arbitrarily poorer under static routing than under adaptive routing.

§**1. Determining an optimal route/ routing strategy.** We can show that the problem of determining an optimal route is NP-hard by transforming the Restricted Shortest Paths Problem (RSP) to the problem of determining an optimal static route. The RSP is defined [5] as follows:

> ***Given*** a directed graph in which each edge has a *length* and a *transition-time*, a source vertex, a destination vertex, and a transition-time bound T, ***determine*** a path of shortest cumulative length from the source vertex to the destination vertex for which the cumulative transition-time is $\leq T$.

Given an instance of the RSP, we can transform it to a problem of obtaining an optimal static route as follows.

1) The graph, the source, and the destination vertices are the same.
2) Edge lengths become the typical delay estimates while edge transition-times become the maximum delay bounds.
3) The transition-time bound T becomes the end-to-end delay bound.

It is straightforward to show that a static route of minimum cumulative typical delay that has worst-case delay bound no greater than T in the transformed instance is exactly a shortest path solution to the RSP instance.

§**2. Comparing the typical delays that can be obtained.** We now show that for some feasible instances, the ratio of the typical delay obtained by an optimal static algorithm to that obtained by an optimal adaptive algorithm may be arbitrarily large. Figure 8 shows an example graph with source s and destination t with an end-to-end delay not exceeding D. Observe that the only static feasible static route is $\langle s \rightarrow v_1 \rightarrow t \rangle$. The typical delay of this route is is $1 + (D-2) = \mathbf{D-1}$.

Consider now the following adaptive strategy:

1 traverse the edge (s, v_1)
2 **if** the actual delay encountered is ≤ 1 time unit
3 continue along the path $\langle v_1 \rightarrow v_2 \rightarrow t \rangle$
// Worst-case delay: $1 + (D-2) + 1 = \mathbf{D}$
4 **else** take the edge $\langle v_1 \rightarrow t \rangle$
// Worst-case delay: $2 + (D-2) = \mathbf{D}$

This adaptive strategy clearly satisfies the worst-case end-to-end delay bound of D. Under typical conditions it would take the path $\langle s \rightarrow v_1 \rightarrow v_2 \rightarrow t \rangle$; its typical delay therefore $1 + 1 + 1$ or $\mathbf{3}$. The ratio of the typical delay under static routing to the typical delay under adaptive routing is therefore $(D-1)/3$; by making the value of D arbitrarily large, this example illustrates that the performance of a static routing strategy may be arbitrarily poor when compared to an adaptive one.

VIII. Summary and Discussion

We have considered real-time routing problems on networks when specified end-to-end delay bounds need to be met, but the delays that are encountered across individual edges of the network are not known precisely beforehand. We have seen that *adaptive* routing strategies, that are not required to statically pre-determine the entire route to the destination prior to setting out from the source vertex, are in general able to provide superior (i.e., smaller) end to end delays. Here we propose a semi-adaptive polynomial time approach and then generalize it to a fully-adaptive approach. Our fully-adaptive approach has the additional benefit that if fresher estimates of delays that are typically encountered across edges become available, such updates can be seamlessly integrated to maintain optimality without increasing run-time complexity.

Despite its super-polynomial run-time complexity, the approach of [1] seems to integrate the most closely with current practice in real-time routing, which is primarily based on constructing lookup tables at each vertex beforehand and using these tables to make on-line routing decisions. We consider our result in Section IV to be particularly significant from this perspective, since it shows that such table-based routing methods *cannot* yield polynomial-time solutions. As we had stated in Section IV, we were surprised to discover that super-polynomial sized tables are needed for optimal routing: this result ran counter to our intuition. As future work we plan to try and better understand what precise features of a network give rise to this need for inordinately large tables: we believe such understanding may yield useful design guidelines for

building real-time networks. We also plan to study table-lookup based routing approaches from the perspective of approximation: how far removed from optimality would the resulting routes be, if we were restricted to constructing polynomial-sized tables in polynomial time?

ACKNOWLEDGEMENTS

This research was supported, in part, by the the National Science Foundation (USA) under Grant Numbers CNS-1618185, CNS-1911460, CCF-1337218 and CCF-1439062.

REFERENCES

[1] Sanjoy Baruah. Rapid routing with guaranteed delay bounds. In *Real-Time Systems Symposium (RTSS), 2018 IEEE*, Dec 2018.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[3] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.

[5] H.C Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications*, 14(2):191 – 197, 1966.