Hard-Real-Time Routing in Probabilistic Graphs to Minimize Expected Delay

Kunal Agrawal[1], Sanjoy Baruah[1], Zhishan Guo[2], Jing Li[3*], Sudharsan Vaidhun[2]
[1]*Washington University in St. Louis;* [2]*University of Central Florida;* [3]*New Jersey Institute of Technology;*

***Abstract*—This work studies the hard-real-time routing problem in graphs: one needs to travel from a given vertex to another within a hard deadline. For each edge in the network, the worst-case delay that may be encountered across that edge is bounded. As far as this given bound is trustworthy at a very high level of assurance, it must be guaranteed that one will meet the specified deadline. The actual delays across edges are uncertain and the goal is to minimize the total expected delay while meeting the deadline. We propose a comprehensive solution to this problem. Specifically, if the precise a priori estimates of the delay probability distributions are available, we develop an optimal table-driven algorithm that identifies the route with the minimum expected delay. If those estimates are not precise (i.e., unknown or dynamic), we develop an efficient Q-Learning approach that leverages the table-driven algorithm to track the true distributions rapidly, while ensuring to meet the specified hard deadline. The proposed solution suggests a promising direction towards incorporating probabilistic information and learning-based approaches into safety-critical systems without compromising safety guarantees, when it is not feasible to establish the trustworthiness of the probabilistic information at the high assurance levels required for verification purposes.**

***Index Terms*—real-time routing, guaranteed delay bounds, minimize expected delay, reinforcement learning**

I. Introduction

Suppose that you are leaving your home s and going to the airport t — see Figure 1. To not miss the flight, you must complete the trip within an hour, but would like to arrive sooner if possible. There are several alternative routes available, and, for now, you are only given an estimate of the maximum delay and the expected (i.e., average) delay that you are likely to encounter on each segment of the road. Which route should you choose? To begin with, should you leave home via the edge (s, v_1) or (s, v_2)? Given the symmetric nature of the delay characterizations on the road segments in the upper and lower halves of Figure 1, it is evident that, in the absence of further information, these two choices are equivalent. With no loss of generality, let us suppose that you chose (s, v_1). Then, your routing strategy should be as follows:

> Traverse the edge (s, v_1). Upon reaching v_1, determine the delay that you have experienced thus far. If it exceeds 10 minutes, then take the edge (v_1, t) to arrive at the airport within the required $(30+30 = \mathbf{60})$ minutes of leaving home, with an additional expected delay of $\mathbf{30}$ minutes. Else take the path $\langle v_1 \to v_3 \to t \rangle$ to arrive at the airport also in $(10+20+30 = \mathbf{60})$ minutes, with an additional expected delay of $(5+5 = \mathbf{10})$ minutes.

This research was supported, in part, by the National Science Foundation (USA) under Grant Numbers CNS–1948457, CNS–1814739, CNS–1911460, CPS–1932530, CNS–1850851, PPoSS-2028481, OIA–1937833, CCF-1733873 and CCF–1725647, and in part by Northrop Grumman Corporation grant. *Jing Li is the corresponding author.

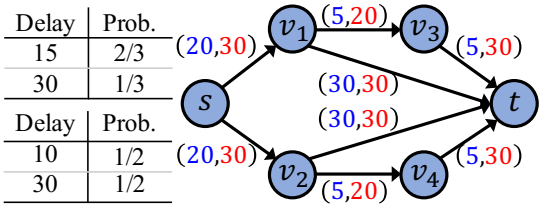


Fig. 1: A routing graph from home to the airport, where blue is the expected delay and red is the maximum delay – all in minutes. Delay bound is one hour (60 minutes).

We can see that this strategy is **safe** — it guarantees to get us to the airport within the specified one-hour "*deadline*" (assuming, of course, that the given maximum delay bounds are correct). But what is the expected (average) duration of the entire trip? That depends on the likelihood of the delay across the edge (s, v_1) exceeding 10 minutes, which cannot be determined based solely upon the provided information; and hence neither can the expected end-to-end duration of the entire trip from the home to the airport.

Now, suppose that in addition to the maximum and expected delays across each edge you were provided with some information regarding the actual *distribution* of delays across the edges, as specified in Figure 1. Specifically, the delay across the edge (s, v_1) is 15 with probability two-thirds and 30 with probability the remaining one-thirds. So the average delay across (s, v_1) is 20. In this case, we will never reach vertex v_1 within 10 minutes of leaving home. Therefore, we must always take the route $\langle s \to v_1 \to t \rangle$ for an end-to-end expected delay of $20 + 30 = \mathbf{50}$ minutes.

On the other hand, the delay across the edge (s, v_2) is either 10 or 30, each with probabilityone-half. Hence, the average delay on (s, v_2) is 20, same as the one on (s, v_1). If we were to leave home via the edge (s, v_2), then there is a probability of one-half that we would reach vertex v_2 in 10 minutes, which means that we would be able to take the path $\langle v_2 \to v_4 \to t \rangle$ for half of the time. The expected end-to-end delay is

$$\frac{1}{2}\big(10 + (5+5)\big) + \frac{1}{2}\big(30+30\big) = \frac{20}{2} + \frac{60}{2} = \mathbf{40} \text{ minutes.}$$

This example illustrates how the knowledge of delay distributions, if available, can be used to make "better" routing decisions — while we cannot distinguish the strategies of leaving home via the edges (s, v_1) or (s, v_2) solely based

on their expected and maximum delays, the knowledge of the actual delay distributions that yield these expected and maximum delays allow us to rate one over the other. When such knowledge is lacking or imprecise, one can only make reasonable "guesses" from experience and exploration, which is in general referred to as "learning."

The problem considered. Real-time routing problems of the kind illustrated in our example above arise in several safety-critical systems, such as autonomous navigation for search-and-rescue, delivery of time-sensitive material across transportation or communication networks. In such problems, there is a hard deadline by which one needs to travel from one specified vertex to another — such a deadline is a safety constraint, that *must* be respected under all circumstances. Subject to this constraint, the *optimization objective* is to minimize the expected traversal time.

Timing guarantees in safety-critical systems must be ensured even under very unlikely circumstances. Hence, only the information that is trustworthy to very high degrees of confidence may be used for assuring safety — in this work, we assume that the worst-case delay bounds are trustworthy at such assurance levels. Additional information that is less trustworthy may also be available. When multiple design options can assure safety, it is reasonable to use such additional information to identify a safe option that is more likely to *optimize* the relevant objectives. In this work, we assume that probability distributions of the actual delays experienced across individual edges constitute such additional information, and study the problem of incorporating such probabilistic information into our algorithms for making routing decisions to optimize performance without compromising safety.

We emphasize that we do *not* require the offline estimations of the delay distributions to be trustworthy to the same high levels of assurance at which safety must be assured. If these estimations are precise, then we optimally solve the optimization problem while guaranteeing worst-case correctness. If the estimations turn out to be imprecise, our learning algorithm approximates the distributions and the optimal offline solution; while the performance (i.e., the actual end-to-end delay) may be sub-optimal, safety will not be compromised (i.e., we are nevertheless guaranteed to meet the specified deadline).

Contribution. For the above hard-real-time routing problem, we provide a *comprehensive* solution that makes use of the additional probabilistic information to minimize the expected end-to-end delay while always meeting the hard deadline. In particular, when accurate delay distributions (for each edge) are *known a priori*, we develop an *optimal* table-driven solution that finds the route with minimum expected delay offline. We further propose an efficient Q-learning[1] approach [2]–[4] to handle the scenario where the delay distributions are imprecise or dynamic. This approach exploits the structure and properties of our proposed table-driven algorithm to formulate a Q-table[2] representation, rapidly approximate the actual distributions, and optimize the routing decisions online.

Our solution is *comprehensive*. When the estimations on delay distributions are precise and static, the proposed table-driven algorithm serves as a perfect initialization for forming the optimal Q-table of the learning algorithm. If the estimations are imprecise, gradually deviate from the actual distributions, or even unknown, the Q-learning algorithm can quickly adapt to the actual delay distributions; once converged, the resulting Q-table can be reduced back to the optimal offline routing tables.

To our knowledge, this is the **first** work that (1) takes probabilistic information in making optimal decisions in routing problems while providing hard-real-time guarantees; and (2) builds a machine learning approach upon theoretically sound and optimal sub-solutions for such problems. Furthermore, it suggests a novel and promising direction towards the integration of safety versus efficiency considerations in the design and analysis of real-time systems.

Organization. Section II formally states the problem under consideration. In Section III, we briefly discuss related works. Section IV describes the optimal table-driven algorithm for the offline setting, while Section V explains the Q-learning algorithm for the online setting. We conduct extensive experiments in Section VI and conclude in Section VII.

II. Model and Problem Statement

In this work, we study the problem of traveling with a low expected delay from the source vertex to the destination in a network, while guaranteeing to arrive at the destination from the source within a specified deadline. We model this problem as a shortest-path problem on directed graphs as follows. We represent the network as a directed graph $G = (V, E)$, with a (probabilistic) delay function $\Pr$ specified on each edge. The vertices represent locations of interest, and the edges direct connections between pairs of locations. For each edge $(u, v) \in E$, $\Pr_{(u,v)}$ is a discrete probability distribution[3] over a finite sample space of positive integers: i.e., $\Pr_{(u,v)}(x)$ denotes the probability that the delay experienced in traversing the edge (u, v) equals x. We initially assume that $\Pr_{(u,v)}$ is *a priori* known for all $(u, v) \in E$. In Section V, we consider the situation when $\Pr_{(u,v)}$ is unknown or dynamic.

For each edge $(u, v) \in E$, we define the *worst-case delay* of the edge, denoted $d_W(u, v)$, to be a (deterministic) upper bound on the maximum delay that will be encountered

[1]Q-learning is a classical Reinforcement Learning [1] mechanism that seeks to find the best policy that maximizes some formulated "reward" of the problem under consideration.

[2]The policy obtained by Q-learning is represented in the form of a look-up table called *Q-table*, which selects the action that optimizes the "reward" given the current system state. The learning process iteratively explores/samples the system space and updates the Q-table using the observed "reward" until the Q-table converges. Since different Q-table representations (with their reward formulations) lead to different converged policies with different performances on the considered problem, it is crucial to formulate a good Q-table representation.

[3]Our techniques are easily extended to the case where these probability distributions are continuous rather than discrete; however, the computational complexity of the resulting algorithms may be considerably higher.

across (u, v).[4] And we define the *expected delay* of the edge, $d_E(u, v)$, to be the quantity: $\sum_x \left(x \cdot \Pr_{(u,v)}(x)\right)$. Note that even when $\Pr_{(u,v)}(x)$ is dynamic or unknown, we assume that the worst-case delay bound $d_W(u, v)$ is known to hold at a very high level of assurance.

A **path** p from some vertex u to vertex v, designated as $u \overset{p}{\rightsquigarrow} v$, is a sequence of vertices $p = \langle u \equiv v_0, v_1, v_2, \ldots, v_k \equiv v \rangle$ such that $(v_{i-1}, v_i) \in E$ for each $i, 1 \leq i \leq k$. We denote the maximum delay and expected delay of a path as:

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

An instance of the problem that we seek to solve is specified as $\langle G = (V, E), \Pr, s, t, D \rangle$, where

- $G = (V, E)$ is a directed graph of the network;
- For each $(u, v) \in E$, $\Pr_{(u,v)}$ is its delay function, which can be known (Section IV) or unknown (Section V);
- $s \in V$ is the source vertex, and $t \in V$ the destination;
- $D \in \mathbb{N}_{\geq 0}$ is the end-to-end deadline from s to t.

Definition* 1 (feasible instance).** Problem instance $\langle G = (V, E), \Pr, s, t, D \rangle$ is said to be ***feasible if and only if there is a path p in the graph from the source vertex s to the destination vertex t (i.e., $s \overset{p}{\rightsquigarrow} t$) for which $d_W(p) \leq D$. □

Consider a particular traversal from s to t, during which we have reached some vertex u and desire to know which outgoing edge to take from u. We know the actual delays that we have encountered thus far, and hence the remaining end-to-end deadline. However, we do not know, prior to traversing an edge $(u, v) \in E$, the actual delay we will experience on traversing (u, v) — all we know is that this delay is guaranteed to not exceed $d_W(u, v)$ and that it is likely to be drawn[5] from the probability distribution $\Pr_{(u,v)}$. Thus, it is not *safe* to take the edge (u, v), if doing so may lead to a vertex from which there is no path to the destination with a guaranteed cumulative delay not exceeding the remaining deadline. This concept is formalized in the definition of safe edges.

Definition* 2 (safe edge).** Let path $p = \langle s \equiv v_0, v_1, \ldots, v_i \rangle$ denote some path that has been traversed across a feasible instance $\langle G = (V, E), \Pr, s, t, D \rangle$. Upon arriving at v_i, the edge $(v_i, v_j) \in E$ is a ***safe edge if and only if there is some path p' from v_j to the destination vertex t ($v_j \overset{p'}{\rightsquigarrow} t$) such that

$$\left(\sum_{k=1}^{i} \mathit{delay}(v_{k-1}, v_k)\right) + d_W(v_i, v_j) + d_W(p') \leq D. \qquad \square$$

Now, we are ready to formally specify the problem that we seek to solve in this paper.

[4]Note that our results and techniques do not require $\Pr_{(u,v)}(d_W(u, v)) > 0$: it is quite possible that the values of worst-case delay $d_W(u, v)$ are derived using a completely different methodology — one that offers a greater degree of assurance — than the methodology that is used to determine the probability distributions (the $\Pr_{(u,v)}$'s). If this is the case, it may be that $\Pr_{(u,v)}(d_W(u, v)) = 0$.

[5]We use "*likely to be*", rather than "is", to emphasize that these probability distributions are trusted to a lower assurance level than the worst-case delays.

Problem statement. *For instance $\langle G = (V, E), \Pr, s, t, D \rangle$ that is feasible, determine a strategy for traversing the graph from s to t that makes the following guarantee: at each vertex along the path, it chooses the outgoing safe edge that results in the **minimum expected delay** from that vertex to t. (Note that, as a consequence, the strategy guarantees a **safe path** with the minimum expected delay from s to t.) In the **offline** version, we assume* Pr *is given and accurate, while in the **online** version,* Pr *is dynamic or unknown.* □

By "*minimum expected duration*", we are applying a **frequentist** interpretation of probability [5]. We seek to obtain a strategy for traversing the graph guaranteeing that, *if we were to repeatedly travel from s to t, then our strategy would, on average, have the minimum delay to get from s to t, while always ensuring safety (i.e., only taking safe edges).*

III. Related Work

There is an extensive body of research on the problem of finding the shortest paths in graphs (see, e.g., [6] for a survey); such shortest-path algorithms have formed the bases of several real-time routing algorithms. Most, however, deal with the situation where there is only a single numerical estimate of the delay across individual edges (rather than a probability distribution for such delay). There is some prior work on probabilistic shortest-path algorithms (e.g., [7]–[9]) and stochastic shortest-path algorithms (e.g., [10]–[15]); however, the probabilistic or stochastic models assumed in these prior works are not suitable for the problem we seek to solve. Amongst other mismatches, to our knowledge, no prior work incorporates the "hard" end-to-end deadline that is a fundamental part of our problem.

The prior work that is most related to ours assumed a model that characterizes each edge e by a worst-case delay (the $d_W(e)$ parameter of our model) and a *typical-case* delay $d_T(e)$, which is a less pessimistic delay bound for all "typical" (i.e., non-pathological) run-time conditions [16], [17]. Such typical-case parameters could be obtained using, e.g., generalizations of the typical-case analysis methodology proposed by Quinton et al. [18]. However, the performance objective considered in [16], [17] is very different from the one we seek here: whereas we seek to identify safe paths that minimize the expected (i.e., average) delay, they aim to identify safe paths that *minimize the maximum delay under all typical conditions.*

Very recently, a Q-learning based approach has been proposed [19], [20]; by designing proper reward functions that reflect the expected delay, this approach explores the safe action space to identify routing decisions that maximize the cumulative reward. This approach does not assume any prior knowledge of the delay distributions. However, the proposed model must be re-trained when the deadline changes even very slightly (please refer to Section V for details).

IV. Table-Driven Algorithm for Offline Routing

This section presents our algorithm that optimally solves the problem described in Section II. In Section IV-A, we explain the design of our algorithm by illustrating its execution upon

(parts of) the example in Figure 1. The pseudo-code form of the algorithm is provided in Section IV-B. Finally, we establish its correctness and prove its optimality in Section IV-C.

A. Algorithm Description

Recall that a problem instance is characterized by $\langle G = (V, E), \Pr, s, t, D \rangle$, denoting that we are to travel from vertex s to vertex t in the graph G within an end-to-end deadline of D, and the delay encountered across each edge $e \in E$ is as specified in the discrete probability distribution $\Pr_e$.

Our algorithm has the following behavior. It first constructs certain data structures for each vertex $v \in V$, which will be used when traversing the graph from the source to the destination. The data structures answer the following questions: *If the remaining time of the end-to-end deadline upon arriving at vertex v is Δ, then which outgoing edge from v should be taken in order to minimize the expected delay of the remaining traversal to the destination t, and what is this expected delay?* That is, at each vertex v these data structures enable efficient computation of two functions: $\pi_v : \mathbb{N} \to V$ and $\xi_v : \mathbb{N} \to \mathbb{R}$, denoting that if one arrives at v with a remaining deadline Δ then one should depart along the edge $(v, \pi_v(\Delta))$, and the expected remaining delay to arrive vertex t is $\xi_v(\Delta)$.

Specifically, the data structure at each vertex $v \in V$ can be thought of as a routing table of 3-tuples (d, π, e) in which all the first components – the d's – are distinct. The presence of a particular 3-tuple (d_o, π_o, e_o) in the table denotes that it is possible to travel from v to t with expected delay e_o while guaranteeing a worst-case delay of d_o, and the edge (v, π_o) is the outgoing edge from v that one should take to achieve this.

After constructing these routing tables in the pre-processing phase, they are used during run-time to compute the quantities $\pi_v(\Delta)$ (routing decision at the next step) and $\xi_v(\Delta)$ (expected delay to reach the final destination) upon arriving at a vertex v with a remaining end-to-end deadline Δ as follows.

- **If** Δ is smaller than the smallest d of any (d, π, e) in the table at vertex v,
- **Then** we return $\pi_v(\Delta) = -$ and $\xi_v(\Delta) = \infty$. This denotes failure: we are unable to find a path from v to t (hence the expected delay of such a path is ∞) that guarantees a worst-case delay no larger than Δ.
- **Else** we identify the unique 3-tuple in the table at vertex v with the largest value of d that is $\leq \Delta$. More specifically, we identify the 3-tuple (d', π', e') in the table at vertex v such that (i) $d' \leq \Delta$; and (ii) there is no 3-tuple (d'', π'', e'') in the table at vertex v such that $(d' < d'' \leq \Delta)$.
- Our algorithm maintains the tuples in sorted order according to the first component – the d values – to allow us to apply a binary search to identify the relevant tuple.

Having identified this 3-tuple (d', π', e'), we return $\pi_v(\Delta) = \pi'$ and $\xi_v(\Delta) = e'$.

Before illustrating how to construct the lookup table, we first illustrate the use of these tables in Example IV.1 below.

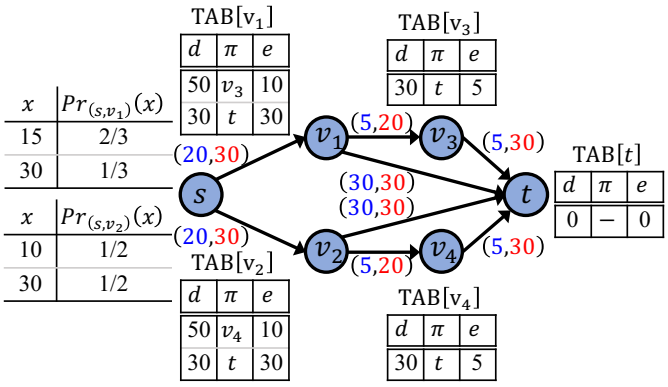


Fig. 2: Illustration of the RELAX operation.

***Example* IV.1.** Figure 2 shows the lookup tables computed by our algorithm for the example of Figure 1 for all the vertices except s. Recall that the deadline D is assumed to be 60.

(1) If we were to leave s along the edge (s, v_1) and arrive at v_1 after encountering a delay of 15, then we have a remaining deadline of $(60 - 15) = 45$. The tuple in the lookup table at vertex v_1 that has the largest d-parameter ≤ 45 is the tuple $(30, t, 30)$; hence, we are directed along the edge (v_1, t), and expect to experience a remaining delay of **30** to arrive at t.

(2) If (s, v_1) has taken a delay of 30, our remaining deadline is 30. The same tuple $(30, t, 30)$ has the largest $d \leq 30$, so we again take (v_1, t) with an expected remaining delay of **30**.

(3) Suppose we had instead chosen to traverse the edge (s, v_2) and experienced a delay of 30. Our remaining deadline is 30. The tuple $(30, t, 30)$ in the lookup table at v_2 has the largest d-parameter ≤ 30, so we are directed along (v_2, t) with an expected remaining delay of **30**.

(4) Finally, suppose we had traversed (s, v_2) and experienced a delay of 10. Our remaining deadline at v_2 is $(60 - 10) = 50$. The tuple $(50, v_4, 10)$ has the largest $d \leq 50$ in the lookup table at v_2; hence, we can take the edge (v_2, v_4) with a smaller expected delay of **10** (rather than 30 in the three cases above). □

In the remainder of this section, we motivate and describe the algorithm we are proposing for the construction of these tables. We start with an **overview** of our approach as follows.

Let TAB$[v]$ denote the lookup table to be constructed at the vertex v, for all $v \in V$. **§1.** We first initialize these tables using an *upper bound* on the expected delay $\xi_v(d)$ for any $d \in \mathbb{N}_{\geq 0}$. **§2.** Next, we repeatedly choose an edge $(u, v) \in E$ and use the information that is available in TAB$[v]$ to *update* TAB$[u]$ such that the upper bounds on $\xi_u(d)$ are reduced — closer to their actual values. **§3.** Our algorithm terminates when no update can be made to change TAB$[u]$ for any $u \in V$.

Note that many well-known (deterministic) shortest-path algorithms on graphs, including the Bellman-Ford and Dijkstra algorithms [21], [22], are centered on this same procedure. The operation of updating upper bounds on shortest path-length estimates at the vertex u based on the presence of the edge (u, v) and upper bounds on shortest path-length estimates at vertex v is referred to in the shortest-paths literature as

d	Expected Delay
80	$\frac{2}{3}(15+10)+\frac{1}{3}(30+10)=30$
65	$\frac{2}{3}(15+10)+\frac{1}{3}(30+30)=36\frac{2}{3}$
60	$\frac{2}{3}(15+30)+\frac{1}{3}(30+30)=50$
45	$\frac{2}{3}(15+30)+\frac{1}{3}(30+\infty)=\infty$

TABLE I: $\textsc{relax}(s, v_1)$

d	Expected Delay
80	$\frac{1}{2}(10+10)+\frac{1}{2}(30+10)=30$
60	$\frac{1}{2}(10+10)+\frac{1}{2}(30+30)=40$
40	$\frac{1}{2}(10+30)+\frac{1}{2}(30+\infty)=\infty$

TABLE II: $\textsc{relax}(s, v_2)$

d	π	e

$\Rightarrow$

d	π	e
80	v_1	30
65	v_1	$36\frac{2}{3}$
60	v_1	50

$\Rightarrow$

d	π	e
80	v_1	30
65	v_1	$36\frac{2}{3}$
60	v_2	40

TABLE III: $\textsc{tab}[s]$ after initialization $\Rightarrow$ $\textsc{tab}[s]$ after relaxing (s, v_1) $\Rightarrow$ $\textsc{tab}[s]$ after relaxing (s, v_2)

relaxation (on) the edge (u, v). Therefore, we also refer to this operation as the relaxation on the edge.

We now elaborate on the above steps of our algorithm.

§1. Initialize the tables. The table $\textsc{tab}[t]$ at the destination vertex t is initialized to contain the single tuple $(0, -, 0)$, denoting that for all delay bounds ≥ 0 there is a path with an expected delay 0 from t to itself, requiring us to take no outgoing edge. The tables at all the other vertices are initially empty, indicating that the best currently-known upper bound on expected delay from that vertex to t is ∞.

§2. RELAX edges and update the tables. Suppose that we are at the vertex u during some traversal from s to t and have a remaining duration d of the specified end-to-end deadline D. If we were to encounter a delay x whilst traversing the edge (u, v), then on reaching vertex v we would have a remaining duration $(d - x)$ of the end-to-end deadline. By the definition of functions π_v and ξ_v, it follows that the outgoing edge we should subsequently take out of vertex v is $\pi_v(d-x)$, and (the upper bound on) the expected delay of the remaining traversal from v to the destination t is $\xi_v(d - x)$. Focusing upon the latter piece of information, we may conclude that should we choose to take the edge (u, v) when we were at vertex u with a remaining end-to-end deadline d, an upper bound on the expected delay of traversing from u to t is given by

$$\sum_{\{x \mid \Pr_{(u,v)}(x)>0\}} \Pr_{(u,v)}(x) \times (x + \xi_v(d - x)) \qquad (1)$$

We illustrate this formula on our running example (Figure 2).

***Example* IV.2.** Now we wish to compute the expected delay in traveling to t within a deadline of 60, if we were to leave s along the edge (s, v_2). From the probability distribution $\Pr_{(s,v_2)}$ depicted in Figure 2, the actual delay encountered across this edge must be one of the two values: **(i)** 10 (with probability $\frac{1}{2}$); or **(ii)** 30 (with probability $\frac{1}{2}$).

(i) If the actual delay on (s, v_2) is 10, then at vertex v we have a remaining deadline of $(60 - 10) = 50$; the expected further delay in getting to t is given by $\xi_v(50)$, which equals **10** (due to the presence of the 3-tuple $(50, v_4, 10)$ in $\textsc{tab}[v_2]$).

(ii) If the actual delay is 30, however, at vertex v we have a remaining deadline of only $(60-30) = 30$. In this case, the expected further delay becomes $\xi_v(30)$, which equals **30**.

Thus, the expected delay if we were to traverse the edge (s, v_2) is $\Pr_{(s,v_2)}(10) \times (10 + \xi_{v_2}(50)) + \Pr_{(s,v_2)}(30) \times (30 + \xi_{v_2}(30)) = \frac{1}{2} \times (10 + 10) + \frac{1}{2} \times (30 + 30) = \mathbf{40}$.

It is important to note the following observation from this example. The computation above was for $d = 60$, but *all the computation steps would remain the same for all values of d in the range* $[60, 80)$: for all such values of d, the options available to us on reaching v are identical to the enumerated cases **(i)** and **(ii)**. That is, if the actual delay encountered is 15 then we have the option of taking the edge (v_1, v_2), while if the actual delay is 30 then we must take the edge (v_1, t). □

The above observation yields the insight that helps us decide which values of d we need to consider in constructing the table $\textsc{tab}[u]$ when relaxing an edge (u, v).

1) Let $\mathcal{X} = \{x \mid \Pr_{(u,v)}(x) > 0\}$
2) Let $\mathcal{D} = \{d' \mid (d', \pi', e') \in \textsc{tab}[v]\}$
3) Consider all (distinct) integers of the form $x + d$ where $x \in \mathcal{X}$ and $d \in \mathcal{D}$, ordered in increasing order in a list.
4) Let d_i and d_{i+1} denote two contiguous integers in this sorted list. The probabilities computed according to Eq. (1) is the same for all $d \in \{d_i, d_i+1, \ldots, d_{i+1}-1\}$.

Hence, when relaxing on the edge (u, v), we need to use Eq. (1) to compute expected delays for only those values of d that are of the form $d = x + d'$ where x is a delay that has a non-zero probability of occurring while traversing the edge (u, v) and d' is the first component of some 3-tuple in $\textsc{tab}[v]$.

***Example* IV.3.** Suppose we were relaxing the edge (s, v_1) with the tables in Figure 2. We apply the above insight:

1) $\mathcal{X} = \{15, 30\}$, $\mathcal{D} = \{30, 50\}$
2) Hence, the sorted list of distinct integers is
$$\langle 15{+}30 = \mathbf{45}, 30{+}30 = \mathbf{60}, 15{+}50 = \mathbf{65}, 30{+}50 = \mathbf{80} \rangle$$
3) Using Eq. (1) on the four values of d, we get the Table I.

Similarly, upon relaxing the edge (s, v_2), the values of d that we need to consider are $10{+}30 = \mathbf{40}$, $30{+}30 = \mathbf{60}$, $30{+}30 = \mathbf{60}$, and $30 + 50 = \mathbf{80}$. Remove the duplicate 60, we are left with three values for d. Using Eq. (1), the expected delays for these values of d are shown in Table II. □

MERGE the results of a RELAX operation. As we have seen above, relaxing along an edge (u, v) yields information about the expected delay that will be encountered if we were to travel along that edge, for different values of the remaining end-to-end deadline. This information, once obtained, is *merged* in with the current lookup table $\textsc{tab}[u]$ at the vertex u, thereby updating $\textsc{tab}[u]$. We illustrate this merge operation on our running example; pseudo-code detailing precisely how this is performed is provided in Figure 3 (Section IV-B).

***Example* IV.4.** Here, we illustrate the process of relaxing edges for $\textsc{tab}[s]$, by going through the steps of first relaxing the edge (s, v_1) and then (s, v_2) for the state in Figure 2.

A. Recall that TAB$[s]$ is initialized to contain no 3-tuples.

B. Suppose that we first relax the edge (s, v_1) with the current TAB$[v_1]$ shown in Figure 2, and computed (d, e) ordered pairs stored in Table I. Merging these ordered pairs into the initially empty TAB$[s]$, we obtain the updated TAB$[s]$ in Table III. To understand why these updates are appropriate (and adequate), let us consider what the presence of an ordered pair (d, e) in Table I indicates. It indicates that if we were at s and had a remaining deadline of d, then choosing to travel along (s, v_1) would yield an expected delay of e. We, therefore, insert the 3-tuple (d, v_1, e) into TAB$[s]$ (except for the last ordered pair $(45, \infty)$ in Table I, for which $e = \infty$ indicates that there is no safe path guaranteeing a delay bound of 45 along this edge).

C. Suppose that we relax the edge (s, v_2) next. In the following, we will individually consider the effect of merging each of the two (d, e) ordered pairs depicted in Table II. Again, the last ordered pair, $(40, \infty)$, need not be merged.

1) The ordered pair $(d = 80, e = 30)$**.** This indicates that there is a path from s to t with maximum delay bound 80 and expected delay 30 that leaves vertex s along the edge (s, v_2). Since this path is no better than the already-known path with maximum delay 80 and expected delay 30 leaving vertex s along the edge (s, v_1) that is asserted by the presence of the 3-tuple $(80, v_1, 30)$ in TAB$[s]$, we do nothing.

2) The ordered pair $(d = 60, e = 40)$**.** The presence of this ordered pair indicates that there is a path from s to t with maximum delay bound 60 and expected delay 40 that leaves s along (s, v_2). Lookup table TAB$[s]$ already has a 3-tuple $(60, v_1, 50)$ with d-value 60; however, this pre-existing 3-tuple asserts the presence of a path of maximum delay bound 60 and expected delay *50* through v_1. Since $40 < 50$, the new path (through v_2) is superior to the previous one, and the $(60, v_1, 50)$ in TAB$[u]$ is ***replaced*** with the 3-tuple $(60, v_2, 40)$.

The lookup table TAB$[s]$ computed after both relax operations (and subsequent merging) is in the right of Table III. □

***Observation* 1.** Note that it is not necessary to generate all the (d, e) tuples by RELAX for an edge (u, v) before they are MERGEd into TAB$[u]$ — it is equally reasonable to merge each tuple once it is generated, which is what we do in Figure 3.

§3. Detect Termination. Observe that relaxing an edge (u, v) obtains *all* the relevant updates in TAB$[u]$ that can be calculated from TAB$[v]$. Hence, having performed this relaxation operation once, there is no point in performing it again *unless the information in* TAB$[v]$ *has changed*. We describe a simple method, based upon this observation, for selecting the order in which edges are relaxed and for determining termination. The method is centered upon maintaining a list Q of vertices v whose current TAB$[v]$ values have not been used to update the TAB$[u]$ values for all the edges $(u, v) \in E$. If the graph $G = (V, E)$ is a directed acyclic graph (DAG), Q is initialized to contain all the vertices in V in reverse topological order. If G is not a DAG, then Q is initialized to contain only the destination vertex t (since the only non-empty table upon initialization is TAB$[t]$). The entire lookup table generation consists of three iterative steps until Q is empty:

1) Remove the vertex v from the front of Q;
2) Relax each edge (u, v) that leads to this vertex v;
3) Add the vertex u to Q if the process of relaxing the edge (u, v) caused a change in TAB$[u]$. We note that for DAGs, a vertex that has been removed from Q is never subsequently added back again by definition of reverse topological order (since by definition of reverse topological order it follows that there can be no edges from vertices in Q to vertices that have previously been removed from Q).

```
GENERATETABLES(⟨G = (V, E), Pr, s, t, D⟩)
 1  for each vertex v ∈ V: TAB[v] ← empty
 2  TAB[t] = (0, −, 0)
 3  if G = (V, E) is a Directed Acyclic Graph
 4       Q ← all vertices in V (in reverse topo. order)
 5  else Q ← only the vertex t
 6  repeat
 7       Remove a vertex v from the front of Q
 8       for each edge (u, v) ∈ E that leads into v
 9            RELAX(u, v)
10            if (TAB[u] changes and the graph is cyclic)
11            then insert u in Q
12  until Q is empty
```

```
RELAX(u, v)
1  for each d_o such that (d_o, π_o, e_o) ∈ TAB[v]
2       for each x_o such that Pr_(u,v)(x_o) > 0
3            d = d_o + x_o
4            e = Σ_{x | Pr_(u,v)(x)>0} Pr_(u,v)(x) (x + ξ_v(d − x))
5            if (e < ∞) // e == ∞ implies no safe path
6            then MERGE(d, e, u, v)
```

```
MERGE(d, e, u, v) // Merge (d, e) into TAB[u]
1  if no 3-tuple in TAB[u] has first component equal to d
2       then Add the 3-tuple (d, v, e) to TAB[u]
3  else let (d, π′, e′) denote the 3-tuple in TAB[u]
       with first component equal to d
4       if (e < e′) //Find new edge with smaller e
5            then Replace this 3-tuple with (d, v, e)
6  Remove all 3-tuples (d′, π′, e′) from TAB[u] with
       d′ > d and e′ ≥ e
```

Fig. 3: Pseudo-code of the pre-processing algorithm.

B. Pseudo-Code Representation

Figure 3 gives the pseudo-code of the pre-processing algorithm. It starts with the procedure GENERATETABLES(), which (i) initiates all the lookup tables (Lines 1-2) and the list Q of vertices (Lines 3-5); and then (ii) repeatedly (Lines 6-12)

removes a vertex from Q and relaxes all the edges leading into this vertex, until termination is detected (Line 12).

The procedure RELAX(u, v) iterates through all the delay bound values d_o in TAB$[v]$ at vertex v; for each d_o, it iterates through all values of x, which represents the possible actual delays that may be experienced on edge (u, v), to determine all values of d that may need to be updated in TAB$[u]$ at vertex u; for each d, the corresponding expected delay e is computed using Equation (1) (Line 4). Each such (d, v, e) (if needed) is merged into the lookup table TAB$[u]$ (Line 6).

The procedure MERGE(d, e, u, v) performs the merging of the newly "discovered" tuple (d, v, e) into TAB$[u]$ (Lines 2 and 5); if doing so renders some pre-existing tuples in TAB$[u]$ redundant, the redundant tuples are removed (Lines 6).

C. Properties: Correctness and Run-Time Efficiency

To prove the algorithm is correct, we first argue that the pre-processing step correctly computes the 3-tuples for each vertex and it converges. We then prove that the path taken by the runtime algorithm minimizes the expected delay and is safe — it never exceeds the end-to-end deadline. In this subsection, instead of saying that we "correctly compute the tuples", we will say that we "correctly compute $\xi_v(d)$ for all values of v and d" for better clarity. Since $\xi_v(d)$ is computed from the tuples stored in TAB$[v]$, these are equivalent statements.

We first define a couple of random variables. Random variable $\zeta_{u,v}(d)$ denotes the actual delay in going from vertex u to t using our runtime algorithm, when the maximum delay allowed from u to t is d *and* the first edge we take is the edge (u, v). Random variable $\zeta_u(d)$ denotes the delay from u to t using our runtime algorithm, if the maximum allowed delay from u to t is d. According to MERGE, $E[\zeta_u(d)]$ selects the minimum $E[\zeta_{u,v}(d)]$ among all outgoing edges (u, v) from u.

$$E[\zeta_u(d)] = \min_{(u,v)\in E}\{E[\zeta_{u,v}(d)]\} \tag{2}$$

Note that, in general, expectation does not distribute over minimums; however, when at vertex u, our runtime algorithm always chooses the edge that provides the minimum expected delay for a given value of d. So the above equation is correct.

We want to show that, after the pre-processing step, for all u and d, we have $\xi_u(d) = E[\zeta_u(d)]$ — that is, if TAB$[u]$ contains a tuple (d', π, e) where $d' \le d$ and there is no $d' < d'' \le d$ then $\xi_u(d) = e = E[\zeta_u(d)]$. We first show the correctness of Eq. (1) and, therefore, the RELAX operation.

***Lemma* 1.** *If we have an edge (u, v), then*

$$E[\zeta_{u,v}(d)] = \sum_{\{x|\ \Pr_{(u,v)}(x)>0\}} \Pr_{(u,v)}(x) \times (x + E[\zeta_v(d-x)]) \tag{3}$$

Proof. Consider the definition of expectation: $E[\zeta_{u,v}(d)] = \sum_{\text{all } y} Prob\{\zeta_{u,v}(d) = y\} \cdot y$. Recall that $\Pr_{u,v}(x)$ is the probability that the delay experienced on edge (u, v) is x and $Prob\{\zeta_v(d-x) = y-x\}$ is the probability that the delay experienced from v to t is $y - x$, while remaining safe (guaranteeing that the delay cannot exceed $d - x$) — since delays experienced on different edges are independent, these two events are independent. Therefore, for any value of y

$$Prob\{\zeta_{u,v}(d) = y\} = \sum_{\{x|\Pr_{(u,v)}(x)>0\}} \Pr_{(u,v)}(x) \cdot Prob\{\zeta_v(d-x) = y-x\}$$

We can substitute this in the definition of expectation:

$$E[\zeta_{u,v}(d)] = \sum_{\text{all } y} Prob\{\zeta_{u,v}(d) = y\} \times y$$

$$= \sum_{\text{all } y} \sum_{\{x|\Pr_{(u,v)}(x)>0\}} \Pr_{(u,v)}(x) \cdot Prob\{\zeta_v(d-x) = y-x\} \cdot y$$

$$= \sum_{\{x|\Pr_{(u,v)}(x)>0\}} \Pr_{(u,v)}(x) \sum_{\text{all } y-x} Prob\{\zeta_v(d-x) = y-x\} \cdot (y - x + x)$$

$$= \sum_{\{x|\Pr_{(u,v)}(x)>0\}} \Pr_{(u,v)}(x)\,(E[\zeta_v(d-x)] + x)$$

where the last equation is by the definition of expectation. □

Corollary 1 follows, since the RELAX operation uses Eq. (1).

***Corollary* 1.** *For any edge (u, v),* RELAX *correctly computes $e = E[\zeta_{u,v}(d)]$ if all the relevant table values for v have already been computed correctly — that is, if, for all $\{x|\ \Pr_{(u,v)}(x) > 0\}$, we have $\xi_v(d-x) = E[\zeta_v(d-x)]$.*

We show that while the pre-processing algorithm is running, all tables store over-approximations of expected delays.

***Lemma* 2.** *While the pre-processing algorithm is executing, $\forall v, d$: (1) $\xi_v(d)$ monotonically reduces; (2) $\xi_v(d) \ge E[\zeta_v(d)]$.*

Proof. We will induct on the order in which the algorithm relaxes edges. In the initialization step, we set up TAB$[t]$ such that $\xi_t(d)$ are correct and all other vertices have $\xi_v(d) = \infty \ge E[\zeta_v(d)]$ for all d. If RELAX calculates for a tuple (d, π, e), MERGE always keeps the smaller value of e for any value of d. In particular, if there is no tuple (d', π, e') in TAB$[v]$ such that $d' \le d$, then old $\xi_v(d)$ is implicitly ∞ and the new tuple is stored. If there is a relevant tuple, then MERGE keeps the tuple with the smaller e. Therefore, after any relaxation and merging step, the values of $\xi_v(d)$ cannot increase.

We prove the second statement by contradiction. Suppose after RELAX(u, v) and its corresponding merges, we have $\xi_u(d) < E[\zeta_u(d)]$ and this is the first such violation. Before this step, by assumption, we have $\xi_v(d) \ge E[\zeta_v(d)]$. Observing Line 4 in RELAX and Corollary 1, we have $e \ge E[\zeta_{u,v}(d)] \ge E[\zeta_u(d)]$ (Eq. (2)). Again, by assumption, we have $\xi_u(d) \ge E[\zeta_u(d)]$ before the merge step. The merge step sets $\xi_u(d) = \min\{\xi_u(d), e\}$. Since both quantities in the minimum are lower bounded by $E[\zeta_u(d)]$, we cannot get the new value of $\xi_u(d) < E[\zeta_u(d)]$. Hence, a contradiction. □

1) Correctness and running time of pre-processing when G is a DAG. We now prove that the algorithm is correct.

***Theorem* 1.** *The pre-processing algorithm terminates after $|E|$ relaxations and the final values $\xi_v(d) = E[\zeta_v(d)]$, $\forall v, d$.*

Proof. We prove by induction on vertices of the DAG in the reverse topological order $S = \langle t \equiv v^0, v^1, ..., v^n \equiv s \rangle$ — the order where the vertices are relaxed in. Our inductive hypothesis is that after all edges (u, v_k) are relaxed (when popping v_k from Q), we have $\xi_{v^i}(d) = E[\zeta_{v^i}(d)]$, $\forall i \leq k$.

The base case $v^0 = t$ is trivial, as $\xi_{v^0}(d) = E[\zeta_{v^0}(d)] = 0$. Say that this is true for vertex v^k. We now show it for vertex $u \equiv v^{k+1}$. Due to reverse topological sorting, all edges (u, v) go to vertices that are before u in S — that is $u \equiv v_i$ where $i < k + 1$. Therefore, by our hypothesis, for all edges (u, v), we know that $\xi_v(d')$ is correct for all d'. Therefore, from Corollary 1, we see that when we relax edge (u, v), all $E[\zeta_{u,v}(d)]$ are correctly computed for all d. The merge procedure keeps the smallest value of e for every d. Therefore, by Eq. (2), after all the merges, $\xi_v(d) = E[\zeta_v(d)]$ for all d. □

Since every edge is relaxed at most once, the total number of calls to RELAX is $|E|$. However, the actual running time of the algorithm is not bounded by any polynomial in the problem size. Specifically, the time taken by a relaxation is $\tau \times |\{x|\Pr_{(u,v)}(x) > 0\}|^2$, where τ is the number of tuples at u and $|\{x|\Pr_{(u,v)}(x) > 0\}|$ is the number of distinct delays with non-zero probabilities at (u, v).

***Observation* 2.** Theorem 1 proves that the time complexity for the pre-processing algorithm is pseudo-polynomial in terms of network size. Note that even in the case where there are no probabilities and the graph only has one typical value and one worst-case value, one can construct DAGs where the number of entries in the table is not a polynomial in the network size [17]. Thus, we cannot hope to bound the running time by a polynomial.

2) Correctness and running time for pre-processing when G has cycles. Proving correctness and convergence is more complicated for graphs with cycles, since we no longer have any order that allows us to relax an edge (u, v) only after all values of $\xi_v(d)$ are correct. Instead, we first observe that the algorithm terminates, since, for a particular value of maximum delay d and a vertex v, Lemma 2 states that our estimate (upper bound) on the expected delay $\xi_v(d)$ from v to t only decreases. In addition, we only have a finite number of values of d for each vertex (0 to D), since the value of d is an integer.

Let T denote the number of final tuples (in the tables of all vertices) that were computed by the pre-processing algorithm once it terminates. The following theorem bounds the number of relaxations needed for termination and also proves that the final tuples are correct when the algorithm terminates.

***Theorem* 2.** *The pre-processing algorithm terminates after at most $|TE|$ relaxations and the final values are correct.*

Proof. We first define an auxiliary graph Γ (for analytical purposes only) based on the final T tuples that were computed by the pre-processing algorithm once it terminates. We label a tuple (d, π, e) at vertex u in G as u_d and add a vertex for each u_d in the auxiliary graph Γ. To define edges in Γ, consider the call to RELAX that led to the final tuple u_d. That is, at some point, some edge (u, v) was relaxed, which led to the creation of tuple (d, v, e) in TAB$[u]$. We add an edge from u_d to $v_{d'}$ if $v_{d'}$ was involved in computing u_d.

The proof follows from the following observations: (1) The pre-processing algorithm terminates; (2) The auxiliary graph Γ is a DAG, so we can do a reverse topological sort S and reason about when a particular tuple in Γ will be computed correctly, similar to the proof of Theorem 1; and (3) After $(k-1)E$ relaxations, we have the correct values of all tuples represented by all vertices up to $v^{k-1} \in S$. Thus, after at most TE relaxations, all tuples in Γ converge to their final correct values. We omit the details due to the space limit. □

3) Runtime Safety. We now show that the runtime behavior is correct using the correctly computed tables for all vertices.

***Theorem* 3.** *At run-time, we start at s with an instantiated delay of D. If the problem instance is feasible, then the runtime algorithm always reaches t within the maximum delay bound.*

Proof. For any path, the average delay is at most the maximum delay. Since the problem instance is feasible, there is a path p from s to t with the maximum delay at most D. Since our algorithm correctly computes the expected delays for each value of the maximum delay, it will compute $\xi_s(D)$ to be smaller than ∞ — therefore, it will not declare failure at s.

We now prove, via contradiction, that it never declares failure on any vertex once it leaves s. Assume for contradiction that we transition from a safe to an unsafe state. Say (v, w) is the first edge at which this happens. That is, we were at vertex v with a remaining deadline Δ and $\xi_v(\Delta) = e < \infty$. We then took the edge $\pi_v(\Delta)$ to vertex w and experienced a delay of x on this edge. At this point, the system became unsafe in that $\xi_w(\Delta - x) = \infty$. Consider the 3-tuple in TAB$[v]$ that lets us compute $\xi_v(\Delta)$ — the tuple (d, e, w) such that $d \leq \Delta$ and $e = \xi_v(\Delta)$. The value of e for this tuple was computed by calling RELAX during the pre-processing step (line 4). Hence, for all values of x where x is the delay we can experience on edge (v, w), we had $\xi_w(d-x) < \infty$. Otherwise, we would have computed the value of ∞ for e. Therefore, $\xi_w(\Delta - x) \leq \xi_w(d - x) \leq \infty$, which is a contradiction. □

***Observation* 3.** Note that, as long as there is a path from every vertex v to the destination t in the network, Theorem 1 to 3 can be generalized to instances from any vertex v to t with any instantiated deadline D. In other words, the calculated lookup tables can be directly used at run-time to find the safe path from v to t that has the minimum expected delay.

V. A Q-Learning Based Approach

The effectiveness of the routing tables generated by the algorithm in Section IV depends upon the accuracy of the estimations of the delay probability distributions Pr on the edges: if they are precise, then our solution is optimal with respect to the total expected delay. We now consider situations where the distributions are imprecise, dynamic, or unknown.

Reinforcement Learning [1] is a framework that can gradually learn the best policy — the best action to take at each

system state, with respect to a designed reward function — after some exploration in an uncertain/complicated system. Thus, it naturally fits the online problem well. **Q-learning** [2]–[4] is a Reinforcement Learning mechanism that represents the policy as a **Q-table**. For a problem under consideration, a Q-learning formulation includes the following: (1) the system states $\mathcal{S}$ and valid actions $\mathcal{A}$ at every state; (2) the Q-table that represents the designed reward (cost) function Q as the expected value, called **Q-value**, of taking an action at a state; (3) the training process that iteratively explores/samples the system space and updates the Q-table using the observed reward (cost) of the sample. During run-time, at every state, selecting the action with the maximum (minimum) Q-value in the converged Q-table gives the best policy that leads to the maximum reward (minimum cost) in the formulation. Since different Q-learning formulations lead to different converged policies with different performances on the considered problem, it is crucial to formulate a good Q-table representation that is capable of effective and efficient learning.

Leveraging the insights obtained in the optimal table-driven algorithm in Section IV, we propose a Q-learning formulation for this online problem. We construct our Q-table to approximate the routing tables in Section IV, such that it properly reflects the expected delay to destination and can be directly initialized using the table-driven algorithm. Thus, our complete solution is comprehensive — it seamlessly connects the online Q-learning approach with the offline table-driven one (without the need for a lengthy training process) and can continuously adapt to dynamic changes in the delay distributions.

Q-Learning Formulation. We model the real-time routing problem as a Markov process on the set of states $\mathcal{S} \stackrel{\text{def}}{=} V \times \mathcal{M}$, where V is the set of vertices in the network and $\mathcal{M}$ is the set of possible integer deadlines. At any state $(u, D) \in \mathcal{S}$, there is a set of valid edges that do not violate the safety requirement of the real-time routing problem, i.e., choosing an edge (u, v) in the set will not cause missing the remaining deadline D from u. These valid edges comprise the set of *actions* $\mathcal{A}_{u,D}$:

$$\mathcal{A}_{u,D} = \{v \mid (u, v) \in E, d^u_{\min} \leq D\}$$

where $d^u_{\min} = \min\{d \mid (d, \pi, e) \in \text{TAB}[u]\}$. Every action $v \in \mathcal{A}_{u,D}$ taken at state (u, D) has an associated *cost* $\sigma_{u,v}$, which is independent of the deadline D but depends on the (unknown) delay distribution $\Pr_{(u,v)}$. The *Q-value* of the state-action pair $\langle u, D, v\rangle$ is formulated to represent the expected cost from u to destination t given a remaining deadline D:

$$Q_{u,D,v} = \sigma_{u,v} + \gamma \cdot \min\{Q_{v,D-\sigma_{u,v},w} \mid (v, w) \in \mathcal{A}_{v,D-\sigma_{u,v}}\}$$

where $\gamma \in (0, 1]$ is the discount factor [23] in Q-learning. Note that when $\gamma = 1$ (as the default), the formulated Q-value converges to the expected delay from u to t given D.

Optimal Q-Values. When the delay distributions on edges are known, the routing tables $\text{TAB}[u]$ as calculated in Section IV with an entry (d, π, e) store the expected delay e from a vertex u to destination t via an edge (u, π) with a remaining deadline d. Since the Q-values reflect these expected delays, we can directly use the optimal table-driven algorithm to initialize the Q-values (rather than obtaining them via a learning process):

$$Q_{u,D,v} = \min\{e \mid (d, \pi, e) \in \text{TAB}[u], \pi == v \wedge d \leq D\} \quad (4)$$

The procedure LEARN() in Figure 4 shows the complete pseudo-code to get the optimal Q-values from the optimal routing table. The deadline $d^u_{\min}$ denotes the minimum deadline required to safely transmit the instance to the destination.

```
LEARN()
1  d_max = maxDeadline
2  for each vertex u ∈ V
3      d^u_min = min{d | (d, π, e) ∈ TAB[u]}
4      for each deadline D ∈ [d^u_min, d_max]
5          for each edge (u, v) ∈ E that leads from u
6              Q_{u,D,v} = min{e | (d, π, e) ∈ TAB[u],
                               π == v ∧ d ≤ D}
```

```
ADAPT(D, ε, α)
1  done = False, u = s //deadline D of the instance
2  while not done
3      A_{u,D} = {v | (u, v) ∈ E ∧ d^v_min ≤ D}
4      for each edge (u, v) ∈ E that leads from u
5          if v ∉ A_{u,D}: P(v|u) = 0
6          else P(v|u) = ε/|A_{u,D}|
7      v = arg min_w {Q_{u,D,w} | ∀(u, w) ∈ E}
8      P(v|u) = P(v|u) + 1 − ε
9      Next vertex v is sampled from distribution P(v)
10     σ_{u,v} is the actual cost of traversing (u, v)
11     Q_{u,D,v} = (1 − α) · Q_{u,D,v} + α · σ_{u,v}
           + α · min{Q_{v,D−σ_{u,v},w} | (v, w) ∈ A_{v,D−σ_{u,v}}}
12     D = D − σ_{u,v}, u = v
13     if v == t: done = True
```

Fig. 4: Q-Learning procedures

Updating Q-Values. When the delay distribution is imprecise or dynamic, the optimal Q-values change over time. On every iteration of an instance through the network, the incurred cost σ varies and is used to update the Q-values. Using a temporal-difference learning algorithm [3], the Q-value is updated via:

$$Q_{u,D,v} = (1 - \alpha) \cdot Q_{u,D,v} + \alpha \cdot \sigma_{u,v} + \alpha \cdot min\{Q_{v,D-\sigma_{u,v},w} \mid \forall\ (v, w) \in \mathcal{A}_{v,D-\sigma_{u,v}}\}$$

where $\alpha \in (0, 1]$ is the *learning rate*. For each instance through the network, the procedure ADAPT(D, ϵ, α) is used to update the Q-values. The ADAPT() procedure starts with an instance at the source $u = s$ with a deadline D. The corresponding initial state is (u, D), and the set of valid actions are calculated in Line 3. The exploration factor $0 \leq \epsilon \leq 1$ denotes the probability of exploring a random action. As shown in Lines 4 to 10, according to the exploration factor ϵ, the probability of each action is calculated, based on which action is taken with an incurred cost $\sigma_{u,v}$. The Q-value $Q_{u,D,v}$ is then updated based on $\sigma_{u,v}$ (Line 11). With the updated

current vertex u and deadline D (Line 12), the process is repeated until the instance reaches the destination t.

Limitations of existing work. Seetanadi et al. recently proposed a Q-Learning approach named `safe RL` [19], [20] that can be applied to our problem. `safe RL` constructs a Q-table specifying the accumulated expected reward for each routing decision when the end-to-end deadline at the source vertex s is D. To our knowledge, this is the first work that leverages machine learning for real-time routing problems. However, because `safe RL` does not use any domain knowledge (e.g., the theoretical analysis of the problem) to guide the Q-table formulation, it has several major drawbacks. First, the deadline is only used for identifying safe paths during training (and run-time), so its constructed Q-table only works for a fixed deadline D. In other words, `safe RL` cannot be trained by instances with different deadlines. Moreover, during run-time, the path chosen by `safe RL` trained with a deadline D may not be optimal (i.e., with the minimum expected delay) for an instance with a deadline $D' < D$. We illustrate this limitation in Example V.1 below. Hence, to obtain the optimal solution for instances with a wide range of deadlines, the training process of `safe RL` must be performed for every possible deadline to obtain their individual Q-tables, which is impractical. Moreover, `safe RL` initializes the Q-table to 0, which may lead to a very long convergence time in training. Finally, unlike the optimal table obtained by the algorithm proposed in Section IV, the Q-table of `safe RL` cannot be used for routing an instance from a vertex v to the destination t, where the vertex v is not the source s.

***Example* V.1.** Consider the network in Figure 2. The converged Q-table trained using Algorithm 2 in [20] with deadline $D = 80$ is shown in the left of Figure 5. Now, consider an instance with deadline $D' = 65$ at vertex s. From the table in Figure 5, the edge (s, v_2) with the maximum Q-value at s will be taken by `safe RL` with an expected delay of 40 and a worst-case delay of 60; while the best choice is (s, v_1) with an expected delay of 36.67 and a worst-case delay of 65, from Table III. This is because `safe RL` has not yet been trained for $D' = 65$ and the converged Q-table in Figure 5 only works for $D = 80$.

In the right of Figure 5, we "froze" the training process to highlight the range of deadlines where `safe RL` is sub-optimal while ours is optimal. Specifically, it shows the average-case and worst-case delay experienced by $10,000$ instances at each deadline D when routing decisions are based on safe RL [20] and our approach. Both approaches are correct in the sense that none exceeds the deadlines during graph traversal, while our proposed approach follows the expected delay more closely. The average delay experienced when routed using safe RL deviates, and most importantly, is higher, compared to the expected delay (which is determined using the optimal offline algorithm). Although both our proposed approach and `safe RL` have only been trained for $D = 80$, our Q-learning formulation allows it to learn how to minimize the average delay for all different deadlines. To obtain the minimum average delays for different deadlines, one would need to perform the training of `safe RL` to all possible deadlines, which could take tremendous time and is impractical. □

S	A	Q
s	v_1	38.163
s	v_2	39.889
v_1	v_3	42.421
v_1	t	17.465
v_2	v_4	44.166
v_2	t	16.665
v_3	t	47.262
v_4	t	49.111

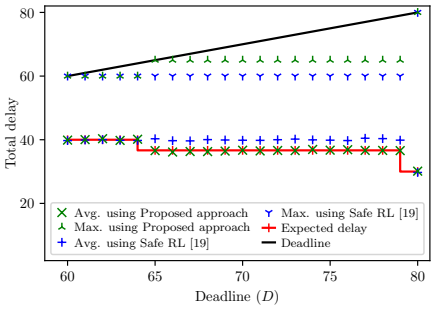


Fig. 5: Left: the converged Q-table using `safe RL` [20] for the network in Figure 2 with deadline $D = 80$; Right: the expected and maximum delays under `safe RL` trained with deadline $D = 80$ and our proposed approach for the network in Figure 2 with increasing deadlines up to 80.

Discussion. The key idea in our proposed approach is that we leverage the structure and property of the optimal routing table in Section IV to construct our Q-table. After the Q-learning algorithm converges, the converged Q-table would be approximately identical to the optimal routing table that can be calculated by our table-driven algorithm if given the accurate delay distributions. In fact, any learning-based algorithm that can be constructed to have the same property, such as SARSA [24], can benefit from our table-driven solution.

If the initial distributions used for generating the optimal offline table is inaccurate or deviate gradually, our Q-learning approach takes a certain number of iterations, where each iteration uses the incurred delays (i.e., the samples from distributions) of one instance through the network to update its Q-table. In addition, our empirical experiments in Section VII show that the time cost for one iteration of updating the Q-table is relatively low. Hence, the convergence of our Q-learning approach depends mostly on the number of samples needed from the distributions to (re-)establish accurate distribution estimations. Hence, when systems have higher dynamics, it is harder for our approach to adapt, which is similar to any statistical/learning-based method. Therefore, we believe that it is the most important to guarantee hard deadlines while the expected delay can be minimized sub-optimally when distribution estimations are not 100% accurate.

VI. Experimental Evaluation

We have implemented the proposed algorithms described in Section IV and evaluated their performance on randomly-generated synthetic workloads by comparing the expected delays achieved by our proposed algorithm against those achieved by (preëxisting) "baseline" algorithms.

§1. Workload Generation. We adapted the modified version of the Erdos-Renyi method [25] proposed by Yang et al. [26] to generate our graphs. When generating acyclic graphs, we avoid cycles by only adding edges that direct from a lower-indexed vertex to a higher-indexed one — the edge (v_i, v_j)

may exist only if $i < j$. To ensure that all paths from the source to the destination contain a reasonably large number of edges (thereby yielding a richer set of feasible paths), we place an additional constraint on an edge (v_i, v_j) where:

$$j - i \leq \beta\sqrt{N} \tag{5}$$

for some specified constant β, say 2, and N is the number of vertices — see below. Condition (5) eliminates the possibility of "shortcut" edges that are obvious shortest paths (e.g., a direct edge from the source to the destination with small average delays and worst-case delays).

To cover a rich subset of scenarios, we control the following parameters (default options are in **bold**):

- Number of vertices: $N \in \{5, 10, 15, \mathbf{20}, 30, 40\}$;
- Probability of a directed edge connecting any pair of vertices: $p \in \{0.1, 0.3, 0.5, \mathbf{0.7}\}$ [6];
- For each edge (v_i, v_j), we set its expected delay to be proportional to $(3\times)$ $|j - i|$, with an additional multiplicative factor that lies uniformly within the range of $[1 - \sigma, 1 + \sigma]$ to ensure a certain degree of randomness ($\sigma \in \{0.1, 0.3, \mathbf{0.5}, 0.7\}$).
- For each edge (v_i, v_j), with the generated expected and maximum delays, we randomly generate $K \in \{\mathbf{2}\ (\mathbf{Bernoulli}), 4, 8\}$ possible delays with probabilities that sum to 1.

§2. Baseline Approaches. We compare our proposed algorithm (denoted as **Algorithm** *opt*) with two baseline algorithms, Algorithm $\mathcal{W}$, Algorithm $\mathcal{T}$, and `safe RL`:

Algorithm $\mathcal{W}$ performs routing considering only the maximum delay (the d_W) parameters. The optimal deterministic Dijkstra's shortest-path algorithm [22] is used. Note that solutions of similar classical (deterministic) routing problems, such as Dijkstras shortest-path algorithm, do not work since the goal of our problem is minimizing the expected delay while guaranteeing that the worst-case delay does not exceed the hard deadline. Using only the maximum delay (the d_W) parameters by the deterministic shortest-path algorithm can ensure that the hard deadlines can be met, but the obtained "shortest-path" in terms of the maximum delay may not be the path with the shortest expected delay.

Algorithm $\mathcal{T}$ performs routing using both the maximum delay d_W and the expected delay d_E parameters. The algorithms proposed in [16], [17] are used, with the expected delay parameter playing the role of the *typical delay* parameter that is needed by them. We point out that for Algorithm $\mathcal{T}$, the TAB$[s]$ calculated in [16], [17] may not provide a precise expected delay from the source to the destination. Take the graph in Figures 1 and 2 as an example, where typical delays are the expectations of all possible delays. The $e'_{\mathcal{T}}$ values in Table IV shows the "typical" delays calculated by Algorithm $\mathcal{T}$ that only uses the expected and maximum delays of edges and not the full distribution — some (shown with underline) clearly do not reflect the actual expected delays ($e_{\mathcal{T}}$) even when routing decisions are identical. Here the optimized expected delays (e_{opt}) by our approach (Algorithm *opt*) are adopted from Table III and are accurate.

[6]To ensure the generated graph is strongly connected, after adding edges with the probability p, if a vertex has in-degree of 0 (or out-degree of 0) and is not the source (or destination), then a random edge that satisfies Condition (5) is added into (or from) that vertex.

d	π	$e'_{\mathcal{T}}$	$e_{\mathcal{T}}$	e_{opt}
80	v_1	30	30	30
65	v_1	$\underline{30}$	$36\frac{2}{3}$	$36\frac{2}{3}$
60	v_1	50	50	50

d	π	$e'_{\mathcal{T}}$	$e_{\mathcal{T}}$	e_{opt}
80	v_1	30	30	30
65	v_1	30	$36\frac{2}{3}$	$36\frac{2}{3}$
60	v_2	$\underline{50}$	40	40

TABLE IV: **Left**: TAB$[s]$ after relaxing (s, v_1); **Right**: TAB$[s]$ after relaxing (s, v_2)

To compare the impact of initial training in the dynamic case, we consider a Q-table generated using **`safe RL`** [20]. As we vary the graph parameters, the complexity of the graph (and thereby the number of state-action pairs in Q-learning) varies, which affects the number of iterations required for the Q-table to be stable. To allow for stabilization, we follow Algorithm 2 in [20] to train the Q-table for $10,000$ iterations by setting a fixed deadline sufficient to cover the longest path in the graph. After this training period, the total expected delay $e_{\mathcal{Q}}$ for each graph is calculated by averaging the total delay over the next 1000 iterations (for testing).

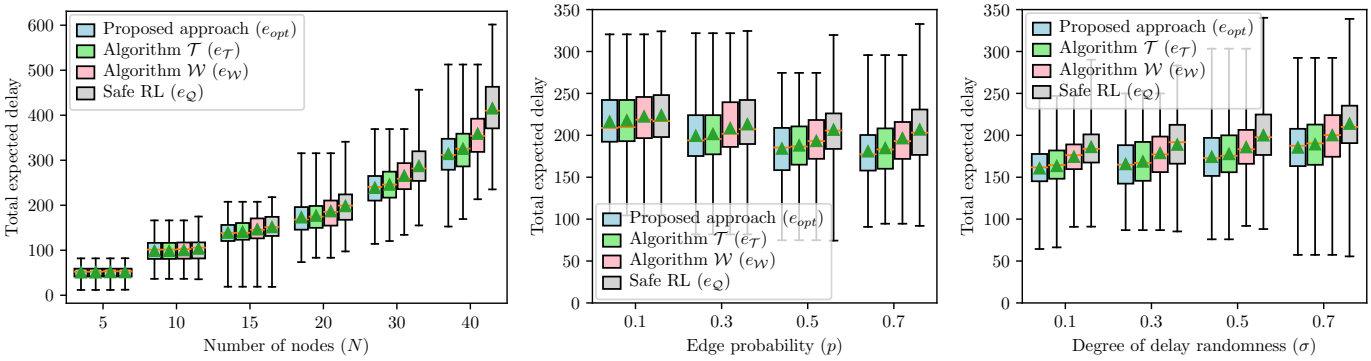
Note that all algorithms select safe paths and will never exceed the end-to-end deadline.

§3. Metrics. Note that when deadline d is too small, no path is safe for guaranteeing to meet the deadline in the worst case. In contrast, when d is too large, all paths become safe, and any shortest path algorithm would suffice. Thus, for deadline d within a meaningful range, we evaluate the variation in the total expected delay e from the source to the destination.

§4. Experimental Results. We use the procedure described in §1 to randomly generate 200 graphs for each setting. Figure 6 reports the distribution (by aggregating with boxplots) of the expected total delays (from the source to the destination) for the four approaches, while varying several key parameters one at a time. For each graph, the expected delays ($e_{opt}, e_{\mathcal{T}}, e_{\mathcal{W}}, e_{\mathcal{Q}}$) from the source to the destination is calculated for each "meaningful" deadline d.

Figure 6(a) reveals that when the size of the graph (N) increases, the absolute values of expected delays increase with more edges and vertices on the paths from the source to destination. Moreover, the relative improvements of our proposed algorithm from the baseline approaches increase as well. This is expected because there are more safe paths from the source to the destination when the graphs are larger. While our proposed algorithm can find the path with the minimum expected delay, the baseline approaches have a higher chance of choosing the other paths with larger expected delays.

Figure 6(b) shows the delay distributions when the probability of generating an edge (p) varies. We can see that delays of all approaches tend to be larger when the graph is sparse (small p), since the number of safe paths is relatively small. Our proposed algorithm constantly outperforms the baseline ones and has higher relative improvements when the number of edges is larger. Intuitively, a graph with more edges has



(a) Varying graph size (N). (b) Varying edge probability (p). (c) Varying delay randomness (σ).

Fig. 6: Comparison of distributions of delays d, where all parameters for graph generation take default values (see §1) unless being varied. The solid triangles represent the mean of each boxplot and the horizontal bar represents its median. The box covers the middle 50% of the data points, and the lower (and upper) limit of the box represents the 25^{th} (75^{th}) percentile.

more safe paths, where the baseline approaches have higher chances to make the wrong choice.

Finally, figure 6(c) shows that when the range of randomness (σ) on edges' actual delays increases, the expected delay slightly increases for all the approaches. This is because with higher randomness the worst-case delay of an edge (and hence a path) becomes larger. This would make certain paths switch from safe to unsafe, which makes the expected delay increases for the approaches.

Under all three settings, the expected delay of `Safe RL` is sub-optimal and the deviation from optimal grows with the graph size N and edge probability p parameters. This is likely due to the increase in state-action pairs as N and p increases, as well as the fact that it is only trained for the largest deadline and cannot generalized well to other deadlines. Additionally, Algorithm $\mathcal{T}$ that uses both the maximum and the expected delay parameters performs better than Algorithm $\mathcal{W}$ that uses only the maximum delay. This is as expected, since the objective here is to minimize the expected delay and more information on the actual delays can help make better decisions. Furthermore, our proposed approach e_{opt} that is able to make use of the full delay distributions achieves the smaller expected delays on average and is optimal for all settings when the delay distributions are known and precise. The variations of different approaches are similar under different settings.

§5. Running time of our approach. Although our offline optimal table-driven solution has a pseudo-polynomial time complexity in the network size, the computation time of calculating the optimal table is relatively low in practice. For example, for a graph with 20 nodes, it takes only $65ms$ on average to obtain the optimal table on a single CPU core.

For our online Q-learning approach, each iteration of updating the Q-table (i.e., running the ADAPT in Figure 4 for one time) is actually not time-consuming as well. The average time cost of each such iteration for a network with 20 nodes is only $3.7ms$ on average running on a single CPU core, which can be further reduced by running it in parallel or on a GPU.

VII. Context and Perspectives

The central issue explored in this paper is *the safe and effective use of probabilistic information in the design and implementation of safety-critical systems*. In the safety-validation of safety-critical systems prior to their deployment, it has been a long-standing principle that only information that can be trusted to the highest level of assurance should be used. As safety-critical systems have become increasingly more complex, their run-time behavior has come to encompass greater uncertainty. Hence, the degree of conservatism and pessimism inherent in such high-assurance information has increased tremendously: the actual run-time behavior tends to be far superior to what can be guaranteed solely based on the high-assurance information available prior to runtime. This begs the question: can we use additional information that cannot be trusted to the very high levels of assurance required for safety-verification in order to improve run-time performance *without compromising safety*?

In this paper, we have considered the use of probabilistic information for this purpose. There has been a large body of research in recent years on the application of probabilistic techniques to the analysis of hard-real-time systems (see [27] for a recent survey on the topic, citing 136 references). However, the safety-critical systems community has remained somewhat skeptical regarding the applicability of such techniques, the main sticking point being the challenge of assuring the trustworthiness of the probabilistic models that are used. The approach we are proposing here circumvents this issue by not using the probabilistic information for safety-verification: our algorithms assure safety using only the worst-case models (the worst-case delay bounds), so the safety holds even if the probabilistic models are completely inaccurate. We have shown that the probabilistic information, if (somewhat) correct, can considerably improve performance: this is shown both via theoretical results (that the expected delay is minimized) and simulations on synthetically generated workloads, where it is shown to outperform prior algorithms that do not use the probabilistic information.

REFERENCES

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[2] C. J. C. H. Watkins, "Learning from delayed rewards," *PhD thesis, King's College, University of Cambridge*, 1989.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf

[4] A. Violante, "Simple Reinforcement Learning: Q-learning," Jul. 2019. [Online]. Available: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

[5] P. J. Bickel and E. L. Lehmann, "Frequentist interpretation of probability," in *Selected Works of E. L. Lehmann*, J. Rojo, Ed. Springer US, 2012, pp. 1083–1085.

[6] H. Ortega-Arranz, D. R. Llanos, and A. Gonzalez-Escribano, *The Shortest-Path Problem: Analysis and Comparison of Methods*. Morgan and Claypool, 2014.

[7] J. F. Bard and J. L. Miller, "Probabilistic shortest path problems with budgetary constraints," *Computers & Operations Research*, vol. 16, no. 2, pp. 145 – 159, 1989.

[8] H. Frank, "Shortest paths in probabilistic graphs," *Operations Research*, vol. 17, no. 4, pp. 583–599, 1969.

[9] M. Hua and J. Pei, "Probabilistic path queries in road networks: traffic uncertainty aware path selection," in *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 347–358.

[10] E. Nikolova, J. A. Kelner, M. Brand, and M. Mitzenmacher, "Stochastic shortest paths via quasi-convex maximization," in *European Symposium on Algorithms*. Springer, 2006, pp. 552–563.

[11] J. Boyan, M. Mitzenmacher, and M. Mitzenmacher, "Improved results for route planning in stochastic transportation," in *Proceedings of the 12th annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2001, pp. 895–902.

[12] Y. Fan, R. Kalaba, and J. Moore, "Arriving on time," *Journal of Optimization Theory and Applications*, vol. 127, no. 3, pp. 497–513, 2005.

[13] R. P. Loui, "Optimal paths in graphs with stochastic or multidimensional weights," *Communications of the ACM*, vol. 26, no. 9, pp. 670–676, 1983.

[14] E. Nikolova, M. Brand, and D. R. Karger, "Optimal route planning under uncertainty." in *ICAPS*, vol. 6, 2006, pp. 131–141.

[15] G. H. Polychronopoulos and J. N. Tsitsiklis, "Stochastic shortest path problems with recourse," *Networks: An International Journal*, vol. 27, no. 2, pp. 133–143, 1996.

[16] S. Baruah, "Rapid routing with guaranteed delay bounds," in *Real-Time Systems Symposium (RTSS), 2018 IEEE*, Dec 2018.

[17] K. Agrawal and S. Baruah, "Adaptive real-time routing in polynomial time," in *Real-Time Systems Symposium (RTSS), 2019 IEEE*, Dec 2019.

[18] S. Quinton, M. Hanke, and R. Ernst, "Formal analysis of sporadic overload in real-time systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. EDA Consortium, 2012, pp. 515–520.

[19] G. Nayak Seetanadi and K.-E. Årzén, "Routing using safe reinforcement learning," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[20] G. Nayak Seetanadi, M. Maggio, and K.-E. Årzén, "Adaptive routing with guaranteed delay bounds using safe reinforcement learning," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2020.

[21] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[22] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[23] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[24] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[25] D. Cordeiro, G. Mounié, S. Pérarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST, Mar. 2010.

[26] K. Yang, M. Yang, and J. H. Anderson, "Reducing Response-Time Bounds for DAG-Based Task Systems on Heterogeneous Multicore Platforms," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2016, pp. 349–358.

[27] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for real-time systems," *LITES*, vol. 6, no. 1, pp. 04:1–04:53, 2019.