Technical Report
MATRIX-BASED ROUTE PRIORITY UPDATE

With all route priority bytes organized in matrices, we can extend the byte-wise operations to equivalent matrix-wise operations. Algorithm 1 presents the matrix-based counterpart of the UPDATE function, MATRIXUPDATE, which takes the link matrix L and the state matrix P^T as its arguments and returns P^{T+1}, the state matrix updated for one more iteration. Within MATRIXUPDATE, we first define the byte-wise function BYTEUPDATE to compute the priority byte for a route received by an AS from its neighbor. Following Equations (3)-(7), BYTEUPDATE takes two arguments: l_{ik}, representing the priority byte of the link from AS a_i to a_k, and p^T_{kj}, representing the priority byte of the best route from a_k to a_j after T iterations. The function returns p^{T+1}_{ikj}, the updated priority byte. To efficiently fulfill this function, BYTEUPDATE starts by refactoring the link information l_{ik}, where the two most significant bits are denoted by $(a\ b)_2$, into three components following Equations (4), (5) and (7). First, *l1* isolates the two most significant bits using a bitwise AND operation with the mask 0b11000000 (Line 3). Next, *l2* is derived by shifting and inverting bits to place the factor of $\bar{a}b$ in the second significant bit position, followed by setting the six lowest bits to ones using the mask 0b00111111 (Line 4). The third component, *l3*, captures the path length, setting it to zero if the link does not exist or is a self-pointing link, or one if the link is an external AS-to-AS link (Line 5). This function then processes the state information p^T_{kj}, where the two most significant bits are denoted by $(c\ d)_2$, in a similar manner. It decomposes p^T_{kj} into *p1*, by isolating the two most significant bits (Line 6), and *p2*, by swapping these bits (Line 7). The bitwise AND operation then combines *p1* with *p2* in place to obtain the factor of *cd* in *p1*'s two most significant bit positions (Line 8), and the bitwise OR operation combines *p2* with p^T_{kj} in place to obtain the factor of $c+d$ in the two most significant bit positions while setting the six least significant bits to those of p^T_{kj} (Line 9). Finally, the components are used to compute p^{T+1}_{ikj} according to Equations (4), (5) and (7). This involves bitwise operations to combine *l1* and *p1*, *l2* and *p2*, followed by a subtraction of *l3* to account for the path length change (Line 11). For clarity, we depict the per-step bit state change for the components described above in Figure 1.

The other nested function BYTESELECTION compares two priority bytes, returning the one with the higher priority. This function is straightforward, using the MAX function to select the more preferable priority byte since the one-byte encoding inherently ensures that a greater byte corresponds to a higher route priority. To enable matrix-wise operations, BYTEUPDATE and BYTESELECTION are vectorized/generalized (Lines 14-15). This allows the entire matrices to be processed efficiently, leveraging highly optimized broadcasting[1] to handle the intermediate dimensionality expansion introduced by Equation (8). The matrix P^{T+1} is then computed using these vectorized/generalized functions. Specifically, the link matrix

Inputs:	l_{ik} =	a	b	~PL[l_{ik}]
	p^T_{kj} =	c	d	~PL[p^T_{kj}]
Line 3:	$l1$ =	a	b	all zeros
Line 4:	$l2$ =	0	$\bar{a}b$	all ones
Line 5:	$l3$ =	0	0	PL[l_{ik}]
Line 6:	$p1$ =	c	d	all zeros
Line 7:	$p2$ =	d	c	all zeros
Line 8:	$p1$ =	cd	cd	all zeros
Line 9:	$p2$ =	c+d	c+d	~PL[p^T_{kj}]
Output:	$\hat{p}^{T+1}_{ikj}$ =	e	f	~PL[p^T_{kj}]−PL[l_{ik}]

Fig. 1: The per-step bit state change inside BYTEUPDATE.

Algorithm 1 Matrix-Based Route Priority Update

```
1: function MATRIXUPDATE(L, P^T)
2:     function BYTEUPDATE(l_ik, p^T_kj)
3:         l1 = l_ik & 0b11000000
4:         l2 = ((~l1 ≫ 1) & l1) | 0b00111111
5:         l3 = ~l_ik & 0b00111111
6:         p1 = p^T_kj & 0b11000000
7:         p2 = ((p1 ≪ 1) | (p1 ≫ 1)) & 0b11000000
8:         p1 = p1 & p2
9:         p2 = p2 | p^T_kj
10:        p̂^{T+1}_ikj = ((l1 & p1) | (l2 & p2)) − l3
11:        return p̂^{T+1}_ikj
12:    function BYTESELECTION(b1, b2)
13:        return MAX(b1, b2)
14:    f1 = VECTORIZE(BYTEUPDATE)
15:    f2 = GENERALIZETOREDUCE(BYTESELECTION)
16:    P̂^{T+1} = f1(L[..., new_axis], P^T[new_axis, ...])
17:    P^{T+1} = f2(P̂^{T+1}, reduce_along_axis = 1)
18:    return P^{T+1}
```

[1]Broadcasting is a mechanism commonly used in matrix libraries such as NumPy that enables efficient arithmetic operations on matrices of differing dimensions by implicitly expanding the smaller matrix to match the larger one's shape, adhering to predefined rules for element-wise calculations.

L and the state matrix P^T are expanded from the shape $n \times n$ to $n \times n \times 1$ and $1 \times n \times n$, respectively, by inserting the new axis accordingly (Line 16). This expansion allows for broadcasting and computing the updated priority bytes across the entire matrix, *i.e.,* $(n \times n \times 1) \cdot (1 \times n \times n) \rightarrow (n \times n \times n)$. The best priority bytes are then selected and reduced along axis one to form the updated $n \times n$ state matrix P^{T+1} (Line 17).

Note that in practice, pre-computing the decomposed components from L (Lines 3-5) can effectively save computational overhead in each iteration. Besides, the size inflation caused by matrix-wise operations can be mitigated by matrix blocking or partial/fully-loop solutions, potentially accelerated using tools like Numba. Also, the state matrix update can be implemented in place, using the same matrix for P^T and P^{T+1}.