Batch Messages in Mixnets

0.1 Mixim Batch Matching

Consider n incoming batches of size k: $B_1, B_2, ..B_n$ and n outgoing batches of size k: $C_1, C_2, ..C_n$.

Let M_{ij} denote the jth incoming message from batch B_i.

Let O_{ij} denote the jth outgoing message in batch C_i.

Variables and Functions	Description	Data Type
M_{ij}	j-th message in the incoming batch i	Message object
O_{pq}	q-th message in the outgoing batch p	Message object
$t_{M_{ij}}$	Arrival time of message M_{ij}	Timestamp
$t_{O_{pq}}$	Sending time of message O_{pq}	Timestamp
$IncomingBatches$	Mapping of incoming batch ids to its messages	Dictionary of dictionaries
$OutgoingBatches$	Mapping of outgoing batch ids to its messages	Dictionary of dictionaries
$OutBatchMappingCount$	For each outgoing batch, it maps the candidate incoming batches to the number of valid permutations to that batch	Dictionary of dictionaries
$OutMsgMappingSet$	Set of valid input messages for each outgoing batch, C_p	Dict of set of messages
$Valids$	List of valid message permutations for all outgoing batches	List of dict of lists
$BatchProb$	Probability mapping for all output batches	Dict of dict
$batchid(msg)$	Returns the batch id of the message	Function (returns integer)
$msgid(msg)$	Returns the message id	Function (returns integer)
$appendMsg(subpermutation, msg)$	Operation to append a message to an existing sup-permutation of a batch	Function (returns list)

0.2 Algorithm: Mixim Batch Matching

$Valids \leftarrow []$
$BatchProb \leftarrow \{\}$
$AnonymitySet \leftarrow \{\}$
$AnonymitySetSize \leftarrow \{\}$
Outgoing message O_{pq} enters the outgoing batch, p at time $t_{O_{pq}}$
$OutMsgMappingSet[p] \leftarrow \{\}$
for each i in $IncomingBatches$ **do**
    $lenIn \leftarrow len(IncomingBatches[i])$
    $lenOut \leftarrow len(OutgoingBatches[p])$
    if $lenIn \geq lenOut$ **then**
        $OutBatchMappingCount[p][i] \leftarrow 0$
    end if
end for
for each i in $OutBatchMappingCount[p]$ **do**
    for each $M_{ij}, t_{M_{ij}}$ in $IncomingBatches[i]$ **do**

```
        if t_{M_ij} < t_{O_pq} then
            OutMsgMappingSet[O_pq].add(M_ij)
        end if
    end for
end for
if Valids ← [] then
    for each M_ij ∈ OutMsgMappingSet[O_pq] do
        Valids ← Valids.append({p : [M_ij]})
    end for
else
    tempValids ← []
    for each M_ij ∈ OutMsgMappingSet[O_pq] do
        i ← batchid(M_ij)
        j ← msgid(M_ij)
        for each x inValids do
            newX ← x
            count ← 0
            msgList ← newX.get(p)
            if msgList then
                for v ← 0 to len(msgList) do
                    b ← batchid(msgList[v])
                    m ← msgid(msgList[v])
                    if b = i and m ≠ j then
                        count ← count + 1
                    else
                        break
                    end if
                end for
                if count ← len(msgList) then
                    newMsgList ← appendMsg(msgList, M_ij)
                    newX[p] ← newMsgList
                    tempValids ← tempValids.append(newX)
                    OutBatchMappingCount[p][i] ← OutBatchMappingCount[p][i] + 1
                    count ← 0
                else
                    count ← 0
                end if
            else
                for each batchMsgs inx do
                    if batchid(batchMsgs[0]) ≠ i then
                        count ← count + 1
                    end if
                end for
                if count ← len(x) then
                    newX[p] ← [M_ij]
                    tempValids ← tempValids.append(newX)
                    OutBatchMappingCount[p][i] ← OutBatchMappingCount[p][i] + 1
                    count ← 0
                else
                    count ← 0
                end if
            end if
        end for
    end for
    Valids ← tempValids
    tempValids ← []
```

```
72: end if
73: for each x in Valids do
74:     for each outid in x do
75:         if outid ≠ p then
76:             inId ← batchid(x[outid][0])
77:             OutBatchMappingCount[outid][inId] ← OutBatchMappingCount[outid][inId] + 1
78:         end if
79:     end for
80: end for
81: for each outBatch in OutBatchMappingCount do
82:     if outBatch not in BatchProb then
83:         BatchProb[outBatch] ← {}
84:     end if
85:     nonZero ← {}
86:     for each (inBatch, count) in OutBatchMappingCount[outBatch] do
87:         prob ← count / len(Valids)
88:         if prob > 0 then
89:             nonZero[inBatch] ← prob
90:         end if
91:     end for
92:     if nonZero then
93:         BatchProb[outBatch] ← nonZero
94:         AnonymitySet[outBatch] ← {nonZero.keys()}
95:         AnonymitySetSize[outBatch] ← len(AnonymitySet[outBatch])
96:     else
97:         if outBatch in BatchProb then
98:             delBatchProb[outBatch]
99:         end if
100:     end if
101: end for
102: OutBatchMappingCount ← {}
103: BatchProb ← {}
104: AnonymitySet ← {}
105: AnonymitySetSize ← {}
```

0.3 Explanation of Mixim Batch Matching Algorithm

Objective

This algorithm constructs all valid mappings between incoming and outgoing batches in a mixnet and computes a full probability distribution ($BatchProb$) for each output batch.

Inputs and Outputs

- **Input:** Outgoing message O_{pq} entering batch C_p at time $t_{O_{pq}}$.
- **Output:** Updated probability estimates $BatchProb[p][k]$ for each candidate batch B_k.

Step-by-Step Explanation of Algorithm

The following describes each major stage of the Mixim Batch Matching algorithm, corresponding to the pseudocode steps.

1. Initialize Candidate Sets. The algorithm begins by initializing empty structures for valid permutations ($Valids = []$) and batch probabilities ($BatchProb = \{\}$) (Lines 1–2). When an outgoing message O_{pq} enters the outgoing batch C_p, $BatchMappingC_p$ is initialised to an empty dictionary so that the count of valid permutations can be recalculated (Lines 3-4). It then determines which incoming batches are eligible to have contributed to the current outgoing batch C_p (Lines 5–9). Specifically, for each incoming batch, if the number of messages is greater than or equal to the number of messages in the outgoing batch C_p, an entry is added to $BatchMappingC_p$ with an initial count of 0. Subsequently, all messages M_{ij} that arrived before O_{pq} and belong to a batch already present in $BatchMappingC_p$ are collected in $MsgMappingO_{pq}$ (Line 10).

2. Construct Initial Valid Permutations. If no valid permutations exist yet (i.e., $Valids = []$) (Line 11), each eligible message M_{ij} from $MsgMappingO_{pq}$ is wrapped in its own single-message path and added to $Valids$ (Lines 12–14). This forms the initial set of valid permutations.

3. Expand Valid Permutations. If $Valids$ already contains existing permutations (Lines 15–55), the algorithm attempts to extend each path with the current message M_{ij}. This is handled in two cases:

- **Existing sub-permutation for batch p:** If the current permutation already has a sub-permutation at index p (Lines 20–37), the algorithm checks whether all messages in that sub-permutation originate from batch B_i and are distinct from M_{ij}. If so, M_{ij} is appended to that sub-permutation, and the updated permutation is added to $tempValids$. The corresponding entry in $BatchMappingC_p$ is incremented.
- **No existing sub-permutation for batch p:** Otherwise (Lines 38–52), the algorithm verifies that batch B_i is not already present in earlier sub-permutations. If valid, a new sub-permutation containing M_{ij} is appended to the path, and the updated path is added to $tempValids$. The batch count in $BatchMappingC_p$ is incremented.

At the end of this step, $tempValids$ replaces $Valids$ (Line 56).

4. Update Batch Mappings for All Other Outgoing Batches. For each valid path in $Valids$, the algorithm iterates through all output batches r except the one corresponding to the current batch $(p-1)$ (Lines 57–64). For each such position, the incoming batch is identified using `batchid()`, and the corresponding count in $BatchMappingC_{r+1}$ is incremented. This ensures that, in addition to updating counts for the current output batch C_p, the algorithm also maintains counts for all other outgoing batches in the permutation.

5. Compute Batch Probabilities for All Outgoing Batches. Finally, the algorithm computes normalized probabilities for every outgoing batch (Lines 65–69). For each index r (covering all batches in each valid paermutation) and each incoming $(batch, count)$ in $BatchMappingC_{r+1}$, the probability is calculated.