Internet Routing: Sequential and Distributed Computation of Routing Paths Across the Internet

João Gonçalves and Teresa Nogueira
Instituto Superior Técnico, University of Lisbon, Portugal

***Abstract*—The Internet consists of a collection of Autonomous Systems (ASes) interconnected by *provider-customer* and *peer-peer* commercial relations. Routing decisions among ASes are driven by policy rather than optimality, producing stable equilibria that sustain global connectivity. This report explores the implementation of sequential and distributed algorithms that compute and simulate these equilibria, their convergence properties and the structure of resulting paths.**

***Index Terms*—Internet Routing, Routing Algebra, Sequential, Distributed, Simulations, Stable Routing, Optimal Routing**

I. Introduction and Motivation

THE Internet consists of a collection of Autonomous Systems (ASes) interconnected by *provider-customer* and *peer-peer* commercial relations:

- in a *provider-customer* relation, the customer *pays* its provider to transit data packets to and from the rest of the Internet;
- in a *peer-peer* relation two ASes exchange traffic between themselves and respective customers *without payment*.

These relations impose hierarchical policies that constrain which paths are valid and how traffic may flow across the network. Routing between ASes follows a *destination-based* forwarding model: each node maintains a table mapping destinations to forwarding neighbors, and packets are routed solely by their destination address. Forwarding decisions are thus local, yet together they determine the global routing structure and its equilibrium.

To model this behavior, each path is assigned a *type–length* attribute (α, n), where $\alpha \in \{1, 2, 3\}$ encodes customer, peer, and provider relations, and n is the number of links (path's length). Type–lengths are compared *lexicographically*,

$$(\alpha, n) \preceq (\alpha', n'), \text{ if } \alpha < \alpha' \ \vee \ (\alpha = \alpha' \wedge n \leq n'), \qquad (1)$$

which establishes preference among competing paths. The concatenation of two path attributes follows the *extension* operation $\oplus$, defined as

$$(\alpha, n) \oplus (\alpha', n') = \begin{cases} (\alpha, n + n'), & \text{if } \alpha = 3 \ \vee \ \alpha' = 1; \\ \bullet, & \text{otherwise;} \end{cases} \qquad (2)$$

where the special symbol $\bullet$ represents an invalid path. This operation captures the commercial constraints: a provider may prepend any path, and a customer may append one, but peer–peer transit is not permitted across tiers.

Given these definitions, the routing state for a destination t is described by a mapping E_t assigning each AS u its preferred type–length,

$$E_t(u) = \sqcap \{ tl[uv] \oplus E_t(v) \mid v \in N_u^+ \}, \qquad (3)$$

where $\sqcap$ selects the most preferred among all candidate paths according to the order defined in (1), $tl[uv]$ denotes the type-length of link uv and N_u^+ the set of out-neighbors of AS u.

✉ {jrazevedogoncalves, maria.teresa.ramos.nogueira}@tecnico.ulisboa.pt

A fixed point of this equation defines a *stable routing*, in which every node forwards packets to a neighbor $v \in N_u^+$ satisfying $E_t(u) = tl[uv] \oplus E_t(v)$. The resulting forwarding links form an in-tree rooted at t, and no node can unilaterally improve its path.

The remainder of this document is organized as follows: Sections II and III introduce the sequential and distributed algorithms,[1] together with implementation details. Section IV discusses the experimental results, and Section V provides final remarks.

II. Sequential Algorithms

A. Stable Routing

The following algorithm computes stable type–lengths $E_t(u)$ to t for all ASes, per (3).

Algorithm 1: StableTypeLength(adj, t)

Input: Array of adjacency lists *adj*; destination AS t;
Output: Stable routing map, E_t, and next-hop from all ASes to t, $next\_hop_t(u)$;
Params: Empty queues: Q_{cust}, Q_{peer}, Q_{prov}.

Function StableTypeLength(adj, t) **is**

```
foreach u ∈ {0, ..., 65535} do
    E_t(u) := •; next_hop_t(u) := u;
E_t(t) := (1, 0); Q_cust := {t} ;   // gets small ϵ
while Q_cust ≠ ∅ or Q_peer ≠ ∅ or Q_prov ≠ ∅ do
    if Q_cust ≠ ∅ then
        v := pop(Q_cust);
    else if Q_peer ≠ ∅ then
        v := pop(Q_peer);
    else
        v := pop(Q_prov);
    foreach u ∈ adj[v] do
        if E_t(u) ≻ tl[uv] ⊕ E_t(v) then
            E_t(u) := tl[uv] ⊕ E_t(v);
            next_hop_t(u) := v;
            α := type(E_t(u)) ;      // α ∈ {1, 2, 3}
            if α = 1 then
                push(Q_cust, u);
            else if α = 2 then
                push(Q_peer, u);
            else if α = 3 then
                push(Q_prov, u);
```

[1] The source code implementing both algorithms is available at https://github.com/Kons-5/IST-ARA-Project, written in standard `ISO C99`.

The algorithm's behavior relies on the properties of the comparison and extension operators introduced earlier. Some implementation and theoretical aspects are discussed next:

1) **Initialization:** All entries of $E_t(u)$ are initialized to the invalid path $\bullet$, which is the worst (maximum) attribute. Hence, starting with $E_t(u) = \bullet$ guarantees that any valid extension is immediately chosen by the selection operator:

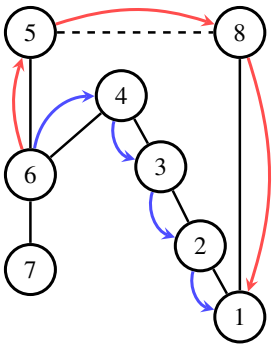
$$\forall \alpha \in \{1,2,3\}: \quad \bullet \sqcap \alpha = \alpha \implies \alpha \preceq \bullet.$$

So any feasible path will replace $\bullet$ at the first comparison.

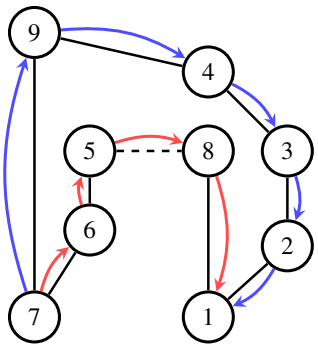
2) **Update Step:** (Highlighted in gray) For each out-neighbor u of v, we compare the current estimate $E_t(u)$ with the candidate $tl[uv] \oplus E_t(v)$. The guard $E_t(u) \succ tl[uv] \oplus E_t(v)$ acts as selection: it triggers if the extension is better than the current estimate under the established preference (1).

Note: The algebra is left-inflationary: for any path P and link uv, $P \preceq tl[uv] \oplus P$. Thus a left extension can never be better than its suffix and $E_t(u)$ is an upper bound on the final value.

3) **Queue Management:** The algorithm keeps three queues (Q_{cust}, Q_{peer}, Q_{prov}), each fed with AS whose extension with their chosen successor yields a path of the Queue type. Queues store only the AS identifier, and E_t is accessed solely at pop time during the update step. Queues are popped in type order: $Q_{\text{cust}} \prec Q_{\text{peer}} \prec Q_{\text{prov}}$. Therefore, paths are explored by type priority, and at all times each AS holds the best type currently attainable. Some subtleties are worth mentioning:



(a) Example 1



(b) Example 2

Iter.	Q_{cust}	Q_{peer}	Q_{prov}	$E_t(6)$
1	{2,8}	{}	{}	$(4,0)$
2	{8,3}	{}	{}	$(4,0)$
3	{3}	{5}	{}	$(4,0)$
4	{4}	{5}	{}	$(4,0)$
5	{}	{5}	{6}	$(3,4)$
6	{}	{}	{6}	$(3,3)$

(c) Evolution of example 1

Iter.	Q_{cust}	Q_{peer}	Q_{prov}	$E_t(7)$
1	{8,2}	{}	{}	$(4,0)$
2	{2}	{5}	{}	$(4,0)$
...				
5	{9}	{5}	{}	$(4,0)$
6	{}	{5}	{7}	$(3,5)$
7	{}	{5}	{7,6}	$(3,5)$

(d) Evolution of example 2

Fig. 1: Queue evolution by iteration for $t = 1$.

- **Example 1**: For $t = 1$, AS 6's preferred path is highlighted in red. Yet the advertisement from AS 4 (in blue) arrives first: 6 computes $tl[6,4] \oplus E_t(4)$, which is of provider type, and is therefore enqueued in the provider queue. Meanwhile, AS 5 has already been placed in the peer queue (iteration 3). Since queues are popped in type order, 5 is processed before 6; its update improves E_t along the red path, so by the time 6 is popped, it already holds the best value for t, which then propagates to AS 7.
- **Example 2**: However, in this case, AS 7's preferred path (in red) is the one that goes via AS 5, but the path that reaches 7 first contains an intermediate provider step through 6, while node 9 reaches 7 directly. In the queues, this puts 7 ahead of 6, so 7 is popped first and advertises a sub-locally-optimal path to its out-neighbors.
  When 6 is popped later, its update improves $E_t(7)$ and overwrites the earlier updates, forcing the correction of the whole subtree beyond 7. This ordering causes extra (avoidable) computations due to transient propagation of inferior paths.

Consequentially, since popping is strictly managed by type, the blue path in Example 2 is explored before the red one. However, once both are extended to reach 7, the preference reverses and the red path becomes better, so a higher-priority path can later be corrected by a lower-priority one.

B. Complexity

Let n denote the number of ASes and m the number of links in a given network.

1) **Setup**: StableTypeLength maintains E_t and $next\_hop_t$ and processes vertices through three FIFO queues. On a pop of v, we scan all neighbors $u \in adj[v]$ and relax upon improval $E_t(u) := tl[uv] \oplus E_t(v)$;
2) **Updates**: Each successful relaxation strictly improves a node's type–length in the order (type priority, then length). Lengths are at most $n-1$; thus, across the three types (multiplicative constant), a node can be improved $\mathcal{O}(n)$ times overall (i.e., enqueued/dequeued $\mathcal{O}(n)$ times);
3) **Work per dequeue**: Popping v touches all incoming edges once, $\mathcal{O}(\deg(v))$. Over all $\mathcal{O}(n)$ possible dequeues of v, this contributes $\mathcal{O}(n \cdot \deg(v))$.

Summing over all vertices, we get the time complexity,

$$T(n,m) = \sum_v \mathcal{O}(n \cdot \deg(v)) = \mathcal{O}(n \cdot m).$$

To circumvent the inefficient "echo" problem stated in the preceding subsection, an alternative to the First-In First-Out (FIFO) queues are priority queues, implemented as min-heaps (keyed by length). Each AS is finalized once, yielding n operations of $\mathcal{O}(\log(n))$, and each relaxed edge can trigger at most one key update, also $\mathcal{O}(\log(n))$. Hence the upgraded variant runs in

$$T(n,m) = \mathcal{O}((n+m)\log(n)).$$

For space, both variants have the same asymptotic progression, which we can bound once:

- **Graph representation**: adjacency lists take $\mathcal{O}(n+m)$;
- **Per-node state**: E_t and $next\_hop_t$ take $\mathcal{O}(n)$;
- **Queues** (three FIFOs or three min-heaps): stores only node indices. Since duplicates are allowed, the number of pending entries can grow up to $\mathcal{O}(n+m)$, which is dominated by adjacency.

Therefore, in both cases the total space is

$$S(n,m) = \mathcal{O}(n+m).$$

C. Optimal Routing

The stable paths from StableTypeLength are not necessarily optimal, as optimality requires a left-isotone algebra, meaning that the relative preference between the attributes of two paths is preserved when both are prefixed by the same preceding path. This property does not hold in this case:

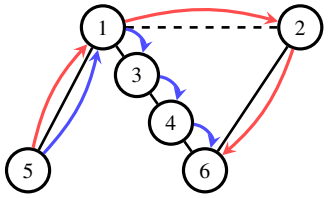


Fig. 2: Violation of left-isotonicity for type–length algebra.

From Fig. 2, assuming the point of view of AS 1, there are two paths to 6: one through 3 (customer path) and another through 2 (peer path), such that

$$(1,3) \prec (2,2).$$

However, when extended to AS 5,

$$(3,1) \oplus (1,3) \prec (3,1) \oplus (2,2) \implies (3,4) \prec (3,3)$$

no longer holds. Intuitively, AS 5 has no alternative but to use a provider path. If it must do so, it prefers the one with the shortest length.

To compute optimal paths, we apply a left-isotonic reduction to the original algebra. This reduction removes order relations that would otherwise violate left-isotonicity. Specifically, paths whose comparison fits:

$$(a', n') \prec_l (a, n), \text{ but } (a, n) \prec_t (a' n)$$

That is, when one path has a better type while the other is shorter, they are considered incomparable, and neither dominates the other.

► **Example**: From Fig. 2, AS 1 would first receive path $(2,2)$ and then $(1,3)$. Upon comparison, it would classify them as incomparable and retain both. AS 5 would then select, among the paths saved by 1, the one with the shortest length. In addition, from (2) we obtain:

- Each AS retains at most two paths: one with the best type among those it receives (its locally optimal path) and one with the shortest length.
- If u is a provider or peer of v and its route to t goes through v, then u may only use v's customer path.
- If u is a customer of v and forwards via v, then u selects, among the paths retained at v, the one with minimum length.

Algorithm 2 computes optimal type–lengths $O_t(u)$ to t for all ASes. This algorithms is similar to StableTypeLength except for the update step, where each AS keeps two per–destination states: $O_t(u)$ (best-by-type) and $I_t(u)$ (best-by-length, if different from $O_t(u)$). Only the update rule changes (highlighted in gray):

Base choice:

- u is a provider/peer of v: base $\coloneqq O_t(v)$;
- u is a customer of v: base $\coloneqq I_t(v)$ (if set), else $O_t(v)$.

Update:

- *Better type*: if $ext \prec_{\text{type}} O_t(u)$, optionally keep the displaced $O_t(u)$ as $I_t(u)$ (if shorter), then set $O_t(u) \coloneqq ext$ and enqueue u by type(ext);
- *Better length*: if not better by type and ext is shorter than $I_t(u)$, set $I_t(u) \coloneqq ext$ and enqueue u by type(ext).

A comparison of results from both StableTypeLength and OptimalTypeLength is made in Section IV.

Algorithm 2: OptimalTypeLength(adj, t)

Input: Array of adjacency lists adj; destination AS t;
Output: Optimal routing map, O_t, Shortest length map, I_t, and next-hop from all ASes to t, $next\_hop_t(u)$;
Params: Empty queues: Q_{cust}, Q_{peer}, Q_{prov}.

Function OptimalTypeLength(adj, t) **is**
  (...) // Similar to StableTypeLength
  while $Q_{\text{cust}} \neq \emptyset$ ***or*** $Q_{\text{peer}} \neq \emptyset$ ***or*** $Q_{\text{prov}} \neq \emptyset$ **do**
    // Pop v from most preferable queue
    foreach $u \in adj[v]$ **do**
      if u *is a customer of* v ***and*** $I_t(v) \neq \bullet$ **then**
        $base \coloneqq I_t(u)$;
      else
        $base \coloneqq O_t(v)$;
      $ext \coloneqq tl[uv] \oplus base$;
      if $ext \neq \bullet$ **then**
        if $ext \prec_{type} O_t(u)$ **then**
          if $O_t(u) \neq \bullet \wedge O_t(u) \prec_n ext$ **then**
            $I_t(u) \coloneqq O_t(u)$;
          $O_t(u) \coloneqq ext$; $next\_hop_t(u) \coloneqq v$;
        else if $ext \prec_n O_t[u]$ **then**
          if $I_t(u) = \bullet \vee ext \prec_n I_t(u)$ **then**
            $I_t(u) \coloneqq ext$; $next\_hop_t(u) \coloneqq v$;
          continue
        // Push u to respective queue

III. Distributed Algorithms

The sequential algorithms in Section II computes stable type–lengths by scanning neighbors in a fixed order. We now present a Discrete Event Simulator (DES) that models asynchrony:

► Advertisements travel on links with delays and trigger local updates.

Both simulators share the same DES skeleton and scheduling rule; they differ only in the update rule, as shown below.

Algorithm 3: UpdateSimple$(u, v, adv, time)$

Input: u receiver, v sender, payload $adv = (\alpha, n)$ at $time$.

if $E_t(u) \succ tl[uv] \oplus adv$ **then**
  $E_t(u) \coloneqq tl[uv] \oplus adv$;
  $next\_hop_t(u) \coloneqq v$;
  foreach $w \in adj[u] \setminus \{v\}$ **do**
    Schedule$(u \to w, E_t(u), time, d)$;

The behavior of SimuSimple is an adaptation of the Chandy-Misra algorithm, where each node advertises an update as long as it detects a local improvement. An important remark is that SimuSimple computes stable/optimal paths only under a total order, which requires the underlying algebra to be isotone. The update rule of the algorithm is monotone non-increasing: an update is generated and propagated only if the extension at the advertiser yields a strictly better estimate.

Algorithm 4: UPDATECOMPLETE$(u, v, adv, time)$

Input: u receiver, v sender, payload $adv = (\alpha, n)$ at *time*.

```
// refresh per-neighbor entry and
   recompute best
```
$tab_u[v] \coloneqq tl[uv] \oplus adv$;
$(E^\star, h^\star) \coloneqq \arg\min_{w \in adj[u]} tab_u[w]$; // under $\prec$
$E^{\text{old}} \coloneqq E_t(u)$;
$h^{\text{old}} \coloneqq next\_hop_t(u)$;
if $E^{old} \neq E^\star$ ***or*** $(h^{old} \neq h^\star \wedge E^\star \neq \bullet)$ **then**
  $E_t(u) \coloneqq E^\star$;
  $next\_hop_t(u) \coloneqq h^\star$;
  foreach $w \in adj[u]$ **do**
    if $h^{old} \neq h^\star$ ***and*** $w = h^\star$ **then**
      continue // avoid when hop changes
    SCHEDULE$(u \to w,\ E_t(u),\ time,\ d)$;

Given that the type–length algebra is non-isotone (as we discussed in Subsection II-C), the algorithm cannot guarantee stable routing paths: preference inversions allow the system to converge to an incorrect terminal configuration.

Consider Fig. 2 again from a distributed perspective, and suppose $(2, 2)$ is the first message to arrive at AS 1. Since $E_t(1) = \bullet$ up until this instant, AS 1 accepts this value and diffuses it to AS 3. The preferred path $(1, 3)$ arrives later and replaces $(2, 2)$ at AS 1, but AS 3 will not update since

$$(3,1) \oplus (2,2) \prec (3,1) \oplus (1,3) \implies (3,3) \prec (3,4)$$

its current estimate is not improved by the new extension. Consequently, the paths computed are incorrect.

In contrast, SIMUCOMPLETE keeps the latest advertisement from each neighbor. Concretely, upon receiving an update, the node stores it in a per-neighbor table, recomputes the best path over all neighbors, and broadcasts the new choice only if it differs from the previously advertised one. Therefore, this algorithm permits estimates to become worse. As a result, SIMUCOMPLETE converges to the same stable solution as the sequential algorithm. An analysis of this behavior is presented in Section IV.

IV. RESULTS AND DISCUSSION

This section presents the results obtained from the algorithms discussed in the previous sections when applied to some test networks. The datasets, listed below, includes both small and large instances corresponding to the topologies provided with the project statement:

- File A: `20090101.as-rel-small.txt`
- File B: `20090101.as-rel.txt`
- File C: `20240701.as-rel-small.txt`
- File D: `20240701.as-rel.txt`

A. Path-Type Distribution Across Algorithms

Table I reports the distribution of types (customer, peer, provider) across the four datasets for STABLEALL, OPTIMALALL, and SIMUCOMPLETEALL. Results for STABLEALL and OPTIMALALL were obtained on all datasets; SIMUCOMPLETEALL could not be run on the last dataset due to prohibitive runtime.

TABLE I: Distribution of types for each dataset.

Type	File A	File B	File C	File D
Customer	03.146%	00.086%	08.166%	00.319%
Peer	22.129%	02.465%	40.966%	11.666%
Provider	74.725%	97.449%	50.867%	88.015%
Invalid	00.000%	00.000%	00.000%	00.000%

Given Table I, we note:

1) The type distribution is the same across STABLEALL, OPTIMALALL, and SIMUCOMPLETEALL.
2) Preference inversions arise only at customers whose path goes through their provider. Providers/peers can forward only along customer paths (by the extension rule), so their first-hop type, and thus the optimal path's type, is fixed. Therefore, OPTIMALALL matches the others in type.
3) Transitions from stable to optimal concern length, not type: a customer will switch to a path of the same type but shorter length given the chance.
4) Consequently, we may infer that even when SIMUSIMPLE fails to build a stable routing map, it still recovers the correct path types.

Additionally, as discussed above, SIMUCOMPLETEALL result's coincides with STABLEALL, and so, for the last dataset we can infer SIMUCOMPLETEALL's output from STABLEALL. We discuss the duration/termination behavior of SIMUCOMPLETEALL relative to STABLEALL in Subsection IV-D.

B. SIMUSIMPLE and Failure to Reach Stable Routing

Figure 4 shows the statement network and Table II lists the per-AS entries computed by the two distributed algorithms for destination $t = 6$.

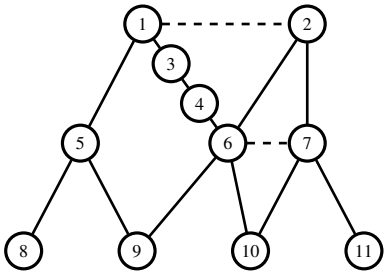


Fig. 4: `statement.txt` network

TABLE II: Comparison of results between SIMUSIMPLE and SIMUCOMPLETE for `statement.txt` (drawn in Fig. 4) for destination $t = 6$ and $d = 1.0$.

	SIMUSIMPLE			SIMUCOMPLETE		
AS	Type	Len	Next Hop	Type	Len	Next Hop
1	1	3	3	1	3	3
2	1	1	6	1	1	6
3	1	2	4	1	2	4
4	1	1	6	1	1	6
5	3	**3**	1	3	**4**	1
6	1	0	6	1	0	6
7	2	1	6	2	1	6
8	3	**4**	5	3	**5**	5
9	3	1	6	3	1	6
10	3	1	6	3	1	6
11	3	2	7	3	2	7

Above we see that SIMUSIMPLE stabilizes on *incorrect* entries at AS 5 and AS 8 (highlighted in red), a consequence of its sensitivity to randomly delayed advertisements on non isotone algebras. In this run, the advertisement via AS 2 reaches AS 1 before the (preferred) one via AS 3, so AS 1

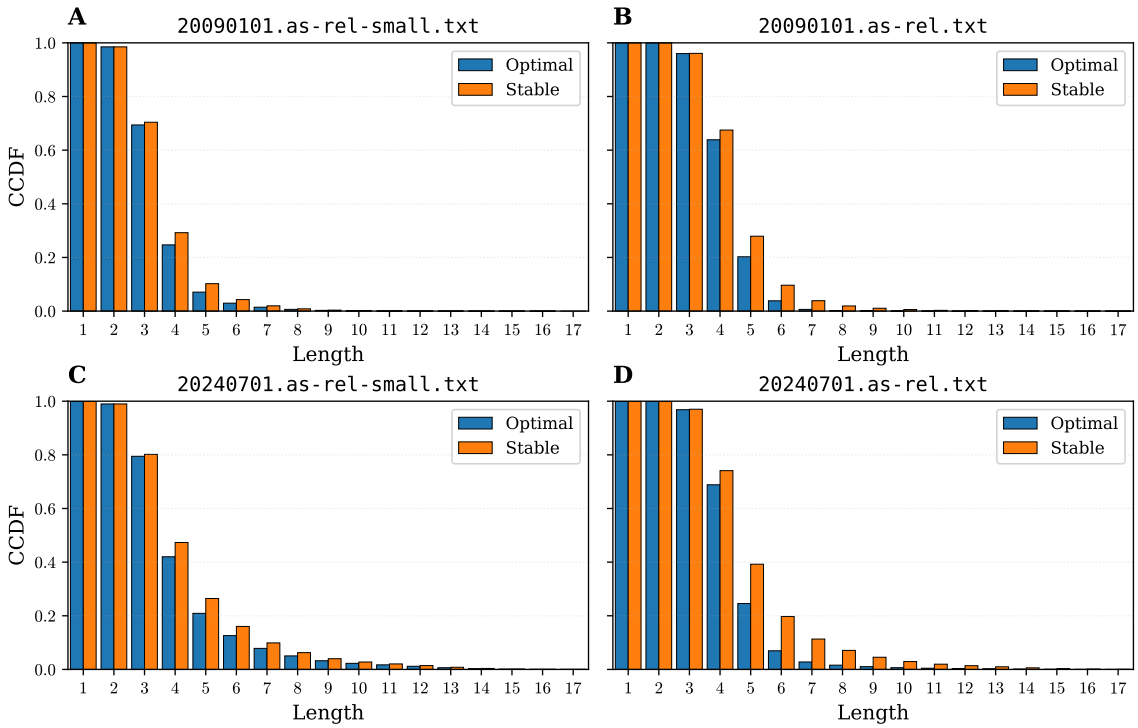


Fig. 3: Complementary Cumulative Distribution Function (CCDF) of the final path lengths under the stable (orange) and optimal (blue) configurations for all test networks.

diffuses the path through 2 to AS 5, and AS 5 further propagates it to AS 8. When the better path via 3 later arrives, AS 1 updates locally, but AS 3 does not, its current estimate is not improved by the new extension, so the correction never reaches 5 or 8.

C. Optimal Paths vs. Stable Paths

Fig. 3 shows the CCDF of lengths for both stable and optimal cases. As expected, the optimal configuration yields consistently shorter routes, with its CCDF shifted left across all datasets. In effect, many nodes traverse paths longer than their individual optimum.

D. Termination Time and Exchanged Messages

The sequential StableAll finishes all files quickly (up to $\sim$ 22 minutes on D). In contrast, the distributed SimuCompleteAll is orders of magnitude slower: it matches A, already lags by $\sim$ 170 times on B ($\sim$ 4 hours), and does not finish C and D within the time budget. This gap is consistent with the previous discussion. We do not claim an exponential bound, but empirically the growth is clearly superlinear and prohibitive on larger inputs.

TABLE III: Time comparison between StableAll and SimuCompleteAll ($d = 1.0$) for an AMD Ryzen 5 3600X 6-Core Processor over 15 trials.

	StableAll	SimuCompleteAll
File A	00h 00min 01s	00h 00min 01s
File B	00h 01min 24s	04h 00min 01s
File C	00h 00min 05s	—
File D	00h 22min 20s	—

For the smallest topology (File A with 737 nodes), the simulator's CCDF of messages is steep and nearly linear, as seen in Fig. 5, most destinations terminate between $\sim$ 9 thousand and $\sim$ 20 thousand messages, with no single "dominant" count. A near-linear CCDF means the underlying counts are spread roughly uniformly over that range. By contrast, on *smaller* networks we often see visible steps/plateaus (many destinations tie at exactly the same message count). As the network grows, path diversity and asynchronous interactions possibly smooth those ties out, so the CCDF becomes a slope.

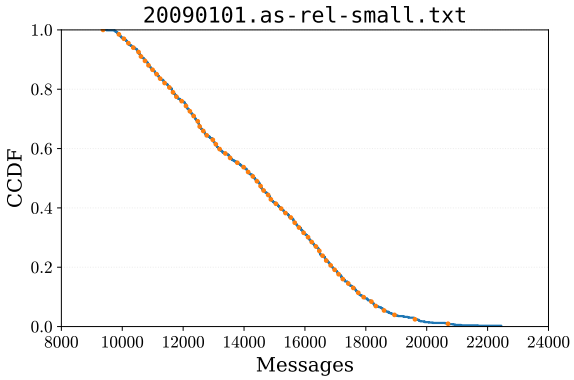


Fig. 5: CCDF of routing messages until termination for the smallest topology (737 nodes).

V. Conclusions

We studied both a distributed algorithm and a sequential algorithm; although the Internet computes routes with distributed protocols; our study shows that a sequential reaches the same stable map orders of magnitude faster. Speed aside, deployment realities (autonomy, asynchrony, failures) force the distributed. Finally, and more broadly, a stable routing map is not a path assignment that jointly best serves all nodes, it is first and foremost policy driven. Hierarchical relationships and economic incentives govern which routes are feasible and preferred. Consequently, efficiency is often sacrificed in the name of profit.