Network Algorithms and Performance

Static Analysis of Inter-AS Routing

Nuno Venturinha 67682
nuno.venturinha.f@tecnico.ulisboa.pt

João Borrego 78990
joao.borrego@tecnico.ulisboa.pt

I. Introduction

This project explores methods for static analysis of Inter-AS (Autonomous System) routing. We start by representing the network as a graph, in which the ASes constitute the nodes and their commercial relationships are the edges. These commercial relationships can essentially be of two different types: *customer-provider* or *peer-peer*. Usually, a network has a hierarchical structure, where the providers are represented above the customers and side-by-side their own peers. At the top of this hierarchy are the so called Tier-1 nodes which have no providers.

The Border Gateway Protocol is the routing protocol that specifies how each node should redirect their traffic. There are three possible route types depending on what type of node they are learned from: *customer*, *peer* and *provider* routes. BGP specifies not only the preference of customer routes over peer, and peer over provider routes, but also which routes can be exported. A node exports every route to its customers and customer routes to its neighbours.

We were tasked with the development and testing of algorithms to perform static analysis of a network for:

1) Customer cycle detection;
2) Determining if the network is commercially connected;
3) Determining types of elected routes from every node in order to reach a given destination;
4) Performing basic statistics regarding elected routes.

II. Problem Formulation

Firstly, it is crucial to decide on a representation for the network graph. We can think of the network as a directed graph. In spite of the fact that a relationship between two nodes is necessarily two-way, it is asymmetrical: if A is a provider of B, B is a customer of A. The peer relationship is a clear exception, since it is symmetric. We opted to use a simple adjacency list representation since the graphs are expected to be sparse. It is also important to establish the naming conventions for relationship and route types. The type of relationships between nodes is encoded as the cost of a relationship as shown in Table I.

TABLE I
Edge types

Edge type	Origin	Destination
C	provider	customer
R	peer	peer
P	customer	provider

The type of route elected by a given node to reach a destination node is encoded as seen in Table II.

The restriction of the routes that can be exported depending on the commercial relationship between to nodes can be expressed as a binary operation denoted by $\otimes$. It consists of a mapping from the type of edge between current and destination node and the type of the route to reach the current node to the resulting type of route between current and destination node, as shown in Table III.

TABLE II
Route types

Route type	Description
c	Origin uses customer route to reach destination
r	Origin uses peer route to reach destination
p	Origin uses provider route to reach destination
$\bullet$	Origin cannot reach destination

TABLE III
Extension operator

$\otimes$	c	r	p	$\bullet$
C	c	$\bullet$	$\bullet$	$\bullet$
R	r	$\bullet$	$\bullet$	$\bullet$
P	p	p	p	$\bullet$

For instance, $(C \otimes c) = c$ represents that a customer route learned from a customer is exported to a provider as a customer route as well. The operator also expresses the impossibility of a node exporting a provider route learned from another provider with $(C \otimes p) = \bullet$.

The preference order for routes types can be modelled by a total order $\prec$, such that

$$c \prec r \prec p \prec \bullet. \tag{1}$$

This total order is associated with a selection binary operation $\oplus$ with

$$\alpha \preceq \beta \text{ if } \alpha \oplus \beta = \alpha. \tag{2}$$

A. Selective semiring formulation and properties

The aforementioned elements are important building blocks for exploiting properties of the problem structure and designing efficient solutions.

For this analysis we refer to the concept of semirings. Semirings are algebraic structures that consist of a set S equipped with two operations $\oplus$ and $\otimes$, usually denoted by $(S,\ \oplus,\ \otimes,\ \bar{0},\ \bar{1})$. The formulation in [1] makes use of this notation, which is similar to the one we chose, yet adds $\bar{0}$, which corresponds to $\bullet$, and $\bar{1}$, which represents an empty path. The author states that some routing problems cannot be modelled with semirings, since some properties such as distributivity, may not be present in the associated structure. However, we may not require that our formulation verifies all of the usual properties for the concrete problem we need to solve.

III. Customer cycle detection

Definition 3.1: A customer cycle consists of a path involving only customer-type edges in which it is possible for a given vertex to reach itself.

In order to perform customer cycle detection one needs only concern with customer-provider type edges. Furthermore, it is only necessary to take either *C* or *P*-type edges.

We used a simple algorithm that merely performs several Depth-First Search attempts and tries to detect a back edge, i.e. an edge leading to an ancestor of the current node in the exploration tree. If one is found, then the graph has a customer cycle. Our solution is detailed in Algorithm 1.

Algorithm 1: Detects if network has a customer cycle

```
input : graph The network graph
output: has_cycle Whether graph has a customer cycle
Function hasCycle(graph)
    for u in graph do
        color[u] ← white;
    end
    for u in graph do
        if color[u] is white then
            if DFS(graph, u, color) then
                return true;
            end
        end
    end
    return false;
Function DFS(graph, source, color)
    color[source] ← grey;
    for u in neighbours of source do
        if color[u] is grey then
            return true;
        else if color[u] is white then
            return DFS(graph, u, color);
        end
    end
    color[source] ← black;
    return false;
```

Our implementation resorts to an auxiliary flag per node, interpreted as a colour, which can be:

- White, if the node has not been visited;
- Grey, if the node has been visited in the current path;
- Black, if the DFS started at the node was concluded.

Every node is initialised with a white colour. Then, we pick an unvisited node and start a DFS. If the DFS finds a back edge, the algorithm terminates and we can conclude that there is indeed a customer cycle. Otherwise, it will continue the DFS until every node in the sub-network is visited. The algorithm terminates when all nodes have been visited, in case no back-edge was found. For this reason, we say that the algorithm is $\mathcal{O}(V)$, V number of nodes in the graph, since each node is visited at most once.

One should point out once more that the neighbourhood of a given node only concerns its customer links, disregarding both peer and provider links.

IV. Determination of whether network is commercially connected

Generally, a graph is considered to be strongly connected if from every vertex there exists a possible route to all other vertices in the graph. In this particular domain, we say the network is commercially connected when each node can elect a viable route in order to reach every other node.

Our approach is quite simple, but relies on domain-specific properties. Firstly, let us formally define a Tier-1 node in a given network:

Definition 4.1: A node that has no providers is known as Tier-1.

Theorem 4.1: Let G be a graph composed of several sub-networks, each with a single Tier-1 node. It suffices that each Tier-1 has a peer connection with the remaining Tier-1 nodes in order to determine that G is commercially connected.

Proof: Let us suppose we have a single network that has a single Tier-1 node as its root, and an arbitrary structure and number of customers. For now we can ignore the presence of peer connections. This graph is commercially connected, since every node can reach all others through the Tier-1.

Now, let us add a second network with similar properties, and equally arbitrary structure and customer nodes. Recall that ascending to a provider, traversing a peer edge, and descending once more by customer edges is a legal action. For this reason, if we add a peer connection between the Tier-1 nodes of each of the two networks, every node will now be able to reach all others, even if it requires the route to reach the Tier-1 node of the sub-network it is in and traverse the peer link to the other sub-network.

Therefore, for the network to be commercially connected, it suffices that the Tier-1 nodes of the network are all connected by peer links. If they are not, one would have the initial situation, with disconnected sub-networks.

The algorithm we designed to verify whether a network is commercially connected is described in Algorithm 2.

Our algorithm merely detects which nodes have no providers, which consist of the list of possible Tier-1 candidates. Then, it is only required to check that each of these nodes is connected to all other via peer links. Determining which nodes are candidates to Tier-1 requires evaluating every edge in the graph once. Checking that indeed each of these nodes is connected to the remaining grows with the square of the number of Tier-1 nodes. Thus, the complexity of this algorithm is $\mathcal{O}(E + (N_1)^2)$, with E the number of edges in the graph and N_1 the amount of Tier-1 nodes.

Notice that the second half of the algorithm essentially obtains the adjacency matrix for the set of nodes in the Tier-1 candidates array, one row at a time. Due to the symmetric behaviour of peer links, it suffices to calculate and evaluate the lower-triangular part of the adjacency matrix in order to evaluate whether there is an all-to-all connection between nodes.

V. Calculation of best route to destination

Our task is to determine the type of route that a given node chooses to reach a certain destination. Dijkstra's algorithm traditional formulation calculates the shortest path tree, i.e., the shortest path from a given source node to every other node in the network. By modifying this algorithm, it is possible to

Algorithm 2: Detects if network is commercially connected

```
input :  $graph$ The network graph
output:  $is\_connected$ Whether graph is commercially
         connected

Function isStronglyConnected($graph$)
    tier_1 ← ∅
    for u in graph do
        if u has no provider then
            add $u$ to tier_1;
        end
    end
    for u in tier_1 do
        for v in tier_1 and $u \neq v$ do
            peer_connected[$v$] ← false;
        end
        for n in neighbours of u do
            if n is peer of u then
                index ← position of n in tier_1;
                if n in tier_1 then
                    peer_connected[$index$] ← true;
                end
            end
        end
        for v in tier_1 and $u \neq v$ do
            if peer_connected[v] is false then
                return false;
            end
        end
    end
    return true;
```

acquire the type of routes that each node elects in order to reach a given destination node.

A. Modified Dijkstra's algorithm

We refer to our problem formulation in Section II. Our objective is to apply a modified version of Dijkstra's algorithm to compute shortest paths. In this case, the cost we are trying to minimise is the type of the routes elected by each node in order to reach the destination.

Before progressing any further, we should begin with ensuring that the algorithm can be applied to this problem and indeed produces a correct solution.

Our $\otimes$ operator is not distributive on $\oplus$[1]. However, we have verified left-isotonicity (equivalent to left-distributivity):

$$\forall_{a,b,c \in S} \quad b \preceq c \Rightarrow a \otimes b \preceq a \otimes c \tag{3}$$

This property essentially means the order relation between the cost of any two paths is preserved if both of them are prefixed by a common, third path.

Our Dijkstra's algorithm will essentially start the path at the destination and gradually prefix edges to the origin. Each of the added edges is extracted from a priority queue, and necessarily has the minimum possible cost in order for the origin node to reach the destination. For this reason, left-isotonicity suffices to ensure that the order relation of the cost of each edge is maintained when extending the path. Thus, a greedy approach is possible.

Our solution is presented in Algorithm 3.

Algorithm 3: Modified Dijkstra's algorithm

```
input :  $graph$ The network graph
         $node$ The destination node
         $queue$ The priority queue
output:  $routes$ Type of route elected by each node in
         order to reach the destination

Function dijkstra($graph$)
    for u in routes do
        routes[u] ← •;
    end
    while queue is not empty do
        extract min from $queue$ such that cost[min] is
            $\bigoplus_{x \in queue}$ cost[x];
        if min.cost is • then
            return;
        end
        routes[min] = min.cost;
        for n in neighbours of min do
            if cost[n] $\prec d(min, n) \otimes cost[n]$ then
                cost[n] ← $d(min, n) \otimes cost[n]$;
            end
        end
    end
```

It is important to point out that since we wish to calculate the optimal paths to a given node instead of the shortest path tree rooted in a given origin node, we need to reverse the values of the edges. This step ensures that we evaluate the correct cost. For instance, assume node A is a provider of B, and we wish to calculate the elected route to reach B. The cost that we wish to evaluate is that of A reaching B which is C. As an implementation detail, instead of altering the values of the edges themselves, we simply swap the first and third lines of the mapping of $\otimes$.

Regarding complexity, the implementation is $\mathcal{O}(E + V) \cdot \mathcal{O}(P)$, with $\mathcal{O}(P)$ the complexity of decrease key of the priority queue, which as we will see is $\mathcal{O}(1)$ (not dependent on the topology of the graph).

B. Priority queue structure

The most common Dijkstra's algorithm formulation uses a priority queue implemented as a minheap, which ensures $\mathcal{O}(1)$ removal of minimum value and $\mathcal{O}(\log n)$ value update. However, in this particular domain, the cost to reach a node is a value from a small set of possibilities. In fact, there are only four possible values for the cost. Using a minheap might underperform if compared with a data structure designed to fit the domain.

We designed a priority queue that also provides $\mathcal{O}(1)$ removal of minimum value and has optimal value update. Furthermore, it requires only a single array, much like a normal minheap implementation.

Our queue is essentially an ordered array, which is divided into blocks, each corresponding to a possible cost value. Since

[1]We have empirically tested some properties by writing code to check every possibility and evaluate the left and right-hand sides of the respective equations.

the array is sorted, the extraction of the minimum value can just be done at the start index. This index should be incremented each time we extract a node.

We keep track of the start of each block so we can reorder the array whenever there is an update with the least amount of value swaps possible. First, recall that we only need to decrease a cost of an element. When doing so, we simply find the element and swap it with the first element of the block it should now belong to. This operation leads to the updated node being in its correct final position, whereas the node that used to be the first in its block may now be out of place. If that is the case we merely repeat the procedure, swap it with the start of the next block. This results in an optimal amount of swaps which is $C-1$, with C the number of possible cost values.

One should also mention that the node we wish to update is found with $\mathcal{O}(1)$ with a separate array that stores the current index for each node that is updated whenever we perform a swap operation. The pseudo-code describing the update node cost method is shown in Algorithm 4.

Algorithm 4: Decreases the cost of a node in the queue

```
input : queue The priority queue
        node The node to be updated
        cost The new cost value
1  Function decreaseKey(queue, node, cost)
2      idx_node ← position of node in index array;
3      old_cost ← queue.array[idx_node].cost;
4      if cost < old_cost then
5          queue.array[idx_node].cost ← cost;
6          for each block with index between cost + 1 and
             old_cost do
7              if block not empty then
8                  swap(queue.array[idx_node],
                     queue.array[block.start]);
9                  update index array to reflect changes;
10             end
11         end
12     end
```

VI. Statistics

Finally, we analysed the provided `LargeNetwork.txt` file, which is an approximate snapshot of the topology of the Internet and obtained the types of routes elected from each node to reach all others. This was achieved by running our modified Dijkstra's algorithm once per node in the network. We summed the total amount of customer, peer and provider routes, and obtained the values shown in Table IV. Our program determined that the graph had no customer cycles, that it had 13 Tier-1 nodes and that it was commercially connected.

TABLE IV
Elected route types statistics

Route type	Absolute count	Ratio (%)
c	11828240	0.61
r	217369490	11.24
p	1704470972	88.14
Total	1933668702	100.00

VII. Optimisation

A. Commercially connected graphs

Since we developed an algorithm to evaluate whether a network is commercially connected, we can leverage that insight in order to optimise our Dijkstra's algorithm performance. Let us suppose that indeed a graph is commercially connected. Then, in the worst case, a node elects a provider route to reach a destination, since it is known it can reach all other nodes. This means that the array of elected routes can be initialised with provider routes. Furthermore, this allows us to stop the algorithm as soon as we extract a node reached by a provider route from the priority queue. Since Dijkstra's algorithm ensures that the each extracted node has the lowest possible cost value, we know the queue will only hold nodes with a final elected route of type provider, or that cannot reach the destination node. However, our initial assumption that the network is commercially connected establishes that every node can reach the destination, which allows us to halt the search while still producing the correct result. Nevertheless, this optimisation can only be used when indeed the network is known to be commercially connected.

The typical architecture of a network of this domain results in nodes electing mostly provider routes. Thus, this optimisation should greatly improve performance.

B. OpenMP parallelization

We experimented further with multi-threaded parallel processing, since each run of the Dijkstra's algorithm is essentially independent. The main concern is ensuring that each thread has its own priority queue structure. The main graph can be shared across threads with no synchronisation issues since we only need read-only access. We modified our solution to support multithreaded processing, assigning a Dijkstra's algorithm run to a set of nodes to each thread. This was implemented using OpenMP. The execution times of each of the versions of our code for the `LargeNetwork.txt` graph can be seen in Figure 1.

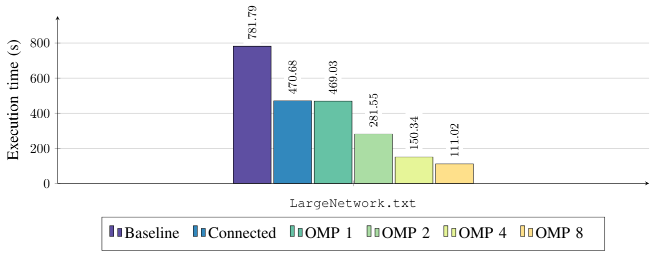

Fig. 1. Execution times for the differently optimised versions of our code

By comparing our OpenMP version running with 8 threads which exploits the fact that the network is commercially connected with the baseline version we obtain that the speedup is approximately 7.04.

Finally, we should mention as an implementation note that we assume that the numbering of each node in the network is sequential, even though we ignore nodes that are completely detached from the network, i.e. are not connected to any other node. This may lead to several empty positions if that is not the case, and produce an inefficient memory usage.

References

[1] Sobrinho, J. L., and Griffin, T. G. Routing in equilibrium.