Algorithms for Fast Resilience Analysis in IP Networks

Michael Menth, Jens Milbrandt, and Frank Lehrieder

Department of Distributed Systems, Institute of Computer Science
University of Würzburg, Am Hubland, D-97074 Würzburg, Germany
Phone: (+49) 931-888 6644, Fax: (+49) 931-888 6632
{menth,martin,lehrieder}@informatik.uni-wuerzburg.de

Abstract. When failures occur in IP networks, the traffic is rerouted over the next shortest paths and potentially causes overload on the respective links. This leads to congestion on links and to end-to-end service degradation. They can be anticipated by evaluating the bandwidth requirements of the traffic on the links after rerouting for a set of relevant failure scenarios $\mathcal{S}$. As this set can be large in practice, a fast evaluation of the bandwidth requirements is needed. In this work, we propose several optimized algorithms for that objective together with an experimental assessment of their computation time. In particular, we take advantage of the incremental shortest path first (iSPF) algorithm to reduce the computation time.

1 Introduction

In IP networks, traffic is forwarded according to the shortest path principle. The OSPF or the IS-IS protocol [1, 2] signal the topology information by the use of link state advertisements (LSA) through the network such that every node in the network knows all working links with their associated cost metrics. Based on this information, each router can run the shortest path first (SPF) algorithm to calculate the least cost paths and to insert the information about the next hops towards any destination in the network into its routing table. When a node or a link fails, this information is disseminated by the routing protocol to all nodes in the network and the distributed routing tables are recomputed. This is called rerouting and leads to the restoration of traffic forwarding when a failure occurs.

In case of single shortest path (SSP) routing, the router calculates the well-defined next hop for the shortest paths towards any destination (cf. e.g. 7.2.7 in [3]). In case of equal-cost multipath (ECMP) routing, the router identifies all next hops on the paths with equal costs towards any destination and records them in the routing table. As a consequence, the traffic is forwarded almost evenly over all of these next hops. The calculation of the shortest paths is quite time consuming since it scales with $O(n \cdot \log(n))$ when n is the number of nodes in the network. This has a significant impact on the restoration time [4]. If a single link or a single router changes its cost, joins, or disappears, only a small fraction of the shortest paths changes. The incremental SPF (iSPF) algorithm adjusts only those shortest paths that are affected by the change. The iSPF algorithm is known from the early days of the ARPANET in the seventies and

This work in cooperation with Infosim GmbH & Co.KG was funded by the Bavarian Ministry of Economic Affairs. The authors alone are responsible for the content of the paper.

has been published in [5]. It speeds up the time for the distributed computation of the shortest paths substantially [6] and it has been implemented recently in routers, e.g., Cisco Systems supports iSPF both for IS-IS and for OSPF [7]. In [8] the complexity regarding comparisons of several iSPF algorithms has been compared experimentally and analytically and another comparison regarding the runtime of Dijkstra's SPF and the iSPF algorithm is provided in [9].

In this paper, we present and assess algorithms to calculate the link utilization for a set of relevant failure scenarios $\mathcal{S}$ in order to detect potential bottlenecks a priori. This set is rather large even if it contains only all single and double element (link or node) failures. Therefore, the applied algorithms must be fast. They calculate (a) the path layout for all end-to-end traffic aggregates in different failure scenarios and based on them (b) the required bandwidth of all links in the network. In contrast to the distributed routing algorithms above, our objective is the computation of the well-defined path layout generated by the distributed calculation of the next hop information. This is achieved by the use of destination graphs. To speed up the computation time, we take a special order of the considered failure scenarios in $\mathcal{S}$ to allow an incremental update of the destination graphs and the required bandwidths. We also adapt the iSPF algorithm to that context to minimize the computation effort. In particular, the iSPF is simplified [10] since it needs to react only to link or node failures but not to new links or nodes, or to the change of their costs. We do not implement any optimization for sorting heaps in the algorithms that can further speed up the calculation [11]. Our results show that our new algorithm is significantly faster than a straightforward naive implementation. Therefore, we recommend their implementation in tools for the a priori detection of overload due to network failures.

The paper is structured as follows. Section 2 presents several optimized algorithms to calculated the bandwidth requirements due to traffic rerouting. Section 3 describes our experimental comparisons of the computation time of these algorithms. In Section 4 we summarize our work and draw our conclusions.

2 Fast Calculation of Resource Requirements in Failure Cases

We represent the network as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V}$ being the set of nodes and $\mathcal{E}$ being the set of edges. A failure scenario s is characterized by the sets of its failed nodes $\hat{\mathcal{V}}(s)$ and links $\hat{\mathcal{E}}(s)$ such that the remaining topology is given by $\mathcal{G}(s) = (\mathcal{V}(s), \mathcal{E}(s))$ with $\mathcal{V}(s) = \mathcal{V} \backslash \hat{\mathcal{V}}(s)$ and $\mathcal{E}(s) = \mathcal{E} \backslash \hat{\mathcal{E}}(s)$. The destination graph $\mathcal{DG}_s^w$ is a directed acyclic graph (DAG) that contains all least cost paths in $\mathcal{G}(s)$ from any source $v \in \mathcal{V}(s)$ to the destination w. To calculate the resource requirements in a special failure case s, we calculate first the destination graphs $\mathcal{DG}_s^w$ for all possible destinations $w \in \mathcal{V}(s)$ and derive then the vector $\mathbf{\hat{r}_s} \in \mathbb{R}_0^{+\,|\mathcal{E}|}$ of the required bandwidth on all links depending on the choice of single shortest path (SSP) or equal-cost multipath (ECMP) routing. Vectors are printed bold, i.e., $\mathbf{0}$ is the zero-vector.

To construct the destination graphs $\mathcal{DG}_s^w$, we present four different methods with increasing optimization degree.

(R0) The simplest one uses Dijkstra's algorithm [12] and recalculates $\mathbf{\hat{r}_s}$ entirely.

(R1) If the destination graphs $\mathcal{DG}_s^w$ for a failure scenario $s \subset s'$ exist, most of the destination graphs $\mathcal{DG}_{s'}^w$ are equal to $\mathcal{DG}_s^w$ and do not need to be computed anew. In this case, the bandwidth vector $\hat{\mathbf{r}}_\mathbf{s}$ can be updated incrementally.

(R2Copy) If a destination graph $\mathcal{DG}_{s'}^w$ changes relatively to $\mathcal{DG}_s^w$ due to an additionally failed element, $\mathcal{DG}_{s'}^w$ may be computed based on a copy of $\mathcal{DG}_s^w$ using the iSPF algorithm. In this case, the incremental update of the bandwidth vector $\hat{\mathbf{r}}_\mathbf{s}$ is even more efficient.

(R2) Copying the full destination graph $\mathcal{DG}_s^w$ may take a long time, therefore, we work with a single copy of $\mathcal{DG}_s^w$ that is reset for another use if it is not needed anymore after its modification to $\mathcal{DG}_{s'}^w$.

2.1 Naive Calculation (R0)

The naive method (R0) calculates the path layout using Dijkstra's algorithm whenever it is needed and computes the bandwidth vector for each scenario $s \in \mathcal{S}$ from scratch.

Basic Path Calculation Based on Dijkstra's Algorithm Algorithm 1 calculates the destination graph $\mathcal{DG}_s^w$ for a specific destination node w by assigning the appropriate set of predecessor nodes $Pred_s^w(v)$ and successor nodes $Succ_s^w(v)$ to each node $v \in \mathcal{V}(s)$.

The remaining list $\mathcal{R}$ contains all nodes without any successors and the tentative list $\mathcal{T}$ contains all nodes that can still become successor nodes for other nodes and for which shorter paths towards the destination can be possibly found. The function $d(v)$ indicates the distance from any node v to the destination w.

The distance from the node v with the shortest distance to the destination w among all nodes on the tentative list $\mathcal{T}$ cannot be further diminished. Therefore, its set of successors is fixed and v can be removed from the tentative list $\mathcal{T}$. Prior to that, the algorithm checks whether v can be used to find shorter paths to one of its predecessor nodes. This is done by looking at all links l whose destination router $\omega(l)$ is v. The source router of such a link l is denoted by $\alpha(l)$ which is a predecessor node of v within the graph. If a shorter path can be found via v, the set of successors $Succ(\alpha(l))$ is substituted. If routing via v provides an equal-cost path, v is just added to the set $Succ(\alpha(l))$. At last the node v is registered as a predecessor node for all its successors. This information is required to implement the incremental algorithm efficiently. The shortest paths towards the destination w are constructed by calling $Dijkstra(\{w\}, \mathcal{V}(s) \setminus \{w\}$ with the initialization $d(w) = 0$ and $d(v) = \infty$ for $v \neq w$.

Basic Calculation of the Required Bandwidth We calculate now the required bandwidth of all links in a special failure scenario s based on the destination graphs $\mathcal{DG}_{v,w}^s$, $v, w \in \mathcal{V}$. The required link bandwidths can be represented by a vector $\hat{\mathbf{r}}_\mathbf{s} \in \mathbb{R}_0^{+|\mathcal{E}|}$ whereby each of its entries $\hat{\mathbf{r}}_\mathbf{s}(l)$ relates to link l.

The rate of the aggregates from a source v to another destination w is given by the entry $\mathbf{R}(v, w)$ of the traffic matrix $\mathbf{R}$. This rate together with the path of the aggregate that can be derived from the destination graph $\mathcal{DG}_s^w$ induce the aggregate-specific rate vector $\mathbf{r}_\mathbf{s}^{\mathbf{v},\mathbf{w}} \in \left(\mathbb{R}_0^+\right)^{|\mathcal{E}|}$. Equal-cost multipath (ECMP) routing uses all suitable paths in

Input: tentative list $\mathcal{T}$, remaining list $\mathcal{R}$
  while $\mathcal{T} \neq \emptyset$ **do**
    $v = argmin_{u \in \mathcal{T}}(d(u))$
    for all $\{l : \omega(l) = v\}$ **do**
      if $d(\alpha(l)) > d(v) + cost(l)$ **then**
        {a shorter path is found for $\alpha(l)$}
        $d(\alpha(l)) \leftarrow d(v) + cost(l)$
        $Succ(\alpha(l)) \leftarrow \{v\}$
      end if
      if $d(\alpha(l)) = d(v) + cost(l)$ **then**
        {an equal-cost path is found for $\alpha(l)$}
        $Succ(\alpha(l)) \leftarrow Succ(\alpha(l)) \cup \{v\}$
      end if
      if $\alpha(l) \in \mathcal{R}$ **then** {move visited node to $\mathcal{T}$}
        $\mathcal{R} \leftarrow \mathcal{R} \setminus \{\alpha(l)\}$, $\mathcal{T} \leftarrow \mathcal{T} \cup \{\alpha(l)\}$
      end if
    end for
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{v\}$ {shortest path fixed for v}
    for all $u \in Succ(v)$ **do**
      $Pred(u) \leftarrow Pred(u) \cup \{v\}$
    end for
  end while

Algorithm 1: DIJKSTRA: calculates a unidirectional destination graph $\mathcal{DG}_s^w$ with links towards the destination.

the destination graph $\mathcal{DG}_s^w$ to forward the traffic, and the traffic is distributed equally over all outgoing interfaces to the destination w. With single shortest path (SSP) routing, the traffic is only forwarded towards the next hop with the lowest ID within all equal-cost paths towards the destination w. This is one choice according to 7.2.7 in [3]. Algorithm 2 and Algorithm 3 calculate the aggregate-specific rate vector $\mathbf{r_s^{v,w}}$ for ECMP and SSP. This vector is first initialized by $\mathbf{r_s^{v,w}} = \mathbf{0}$ before the algorithms are called by RATEVECTORX($\mathbf{r_s^{v,w}}, v, w, \mathbf{R}(v,w)$) with $X \in \{ECMP, SSP\}$. In case of ECMP, Algorithm 2 distributes the rate c at the node v over the links towards the successors $Succ_s^w(v)$ within the destination graph $\mathcal{DG}_s^w$. In case of SSP, Algorithm 3 distributes the traffic from any node v over the single link towards its successor node with the lowest node ID in the destination graph $\mathcal{DG}_s^w$. If a node v has failed, the destination graph $\mathcal{DG}_s^v$ does not exist and no other destination graph $\mathcal{DG}_s^x$ contains v. Thus, the aggregate-specific rate vectors $\mathbf{r_s^{x,v}} = \mathbf{0}$ and $\mathbf{r^{v,x}} = \mathbf{0}$ are zero.

The vector of the required link bandwidth $\mathbf{\hat{r}_s}$ is computed as the sum of all aggregate-specific rate vectors

$$\mathbf{\hat{r}_s} = \sum_{v,w \in \mathcal{V}: v \neq w} \mathbf{r_s^{v,w}}. \tag{1}$$

Input: destination graph $\mathcal{DG}_s^w$, rate vector $\mathbf{r}$, node v, destination w, rate c
  $c' \leftarrow \frac{c}{|Succ_s^w(v)|}$
  for all $u \in Succ_s^w(v)$ **do**
    $\mathbf{r}(l(v,u)) \leftarrow \mathbf{r}(l(v,u)) + c'$
    if $u \neq w$ **then**
      $RateVectorECMP(\mathcal{DG}_s^w, \mathbf{r}, u, w, c')$
    end if
  end for

Algorithm 2: RATEVECTORECMP: calculates the aggregate-specific rate vector $\mathbf{r}$ induced by a flow from v to w with rate c for ECMP routing.

Input: destination graph $\mathcal{DG}_s^w$, rate vector $\mathbf{r}$, node v, destination w, rate c
  $u \leftarrow argmin_{u' \in Succ_s^w(v)}(ID(u'))$
  $\mathbf{r}(l(v,u)) \leftarrow \mathbf{r}(l(v,u)) + c$
  if $u \neq w$ **then**
    $RateVectorSSP(\mathcal{DG}_s^w, \mathbf{r}, u, w, c)$
  end if

Algorithm 3: RATEVECTORSSP: calculates the aggregate-specific rate vector $\mathbf{r}$ induced by a flow from v to w with rate c for ECMP routing.

2.2 Incremental Naive Calculation (R1)

The incremental naive method takes advantage of the fact that the set of protected failure scenarios $\mathcal{S}$ contains many similar failure scenarios s' being a superset of others ($s \subseteq s'$). It saves computation time for the failure-specific destination graphs $\mathcal{DG}_s^w$ and allows an incremental calculation of the bandwidth vector $\hat{\mathbf{r}}_\mathbf{s}$.

Selective Path Calculation Using Dijkstra's Algorithm If two failure scenario s and s' are similar, most of their destination graphs $\mathcal{DG}_s^w$ and $\mathcal{DG}_{s'}^w$ do not differ. In particular, if s is a subset of s' ($s \subset s'$), $\mathcal{DG}_{s'}^w$ only differs from $\mathcal{DG}_s^w$ if $\mathcal{DG}_s^w$ contains an element of the set difference $\Delta_s = s' \setminus s$. Thus, we construct a function $Contains_s^w(x)$ whenever we build a new destination graph $\mathcal{DG}_s^w$. In addition, we arrange all failure scenarios $s \in \mathcal{S}$ hierarchically in such a way that we can take advantage of the contains-relationships. An example for such an order is given in Figure 1. As a result, if $(s \subset s')$ holds, the destination graph $\mathcal{DG}_{s'}^w$ can be overtaken from $\mathcal{DG}_s^w$ unless $Contains_s^w(x)$ evaluates to $true$ for any $x \in \Delta_s$.

Incremental Calculation of the Required Link Bandwidth $\hat{\mathbf{r}}_{\mathbf{s}'}$ Based on Recalculated Destination Graphs $\mathcal{DG}_s^w$ If the path of the aggregate is the same in the failure scenario s and s', $\mathbf{r}_\mathbf{s}^{\mathbf{v},\mathbf{w}}$ can be used instead of $\mathbf{r}_{\mathbf{s}'}^{\mathbf{v},\mathbf{w}}$ for the calculation of the required bandwidth vector in Equation (1). The path is the same if s contains only a subset of the failures in s' and if all links and nodes in $\mathcal{DG}_s^w$ are working in $\mathcal{DG}_{s'}^w$, too. Algorithm 4 calculates the vector of the required link bandwidths for failure scenario s' based on the one for s. It uses the function $Contains_s^w(x)$ to find all aggregates whose destination

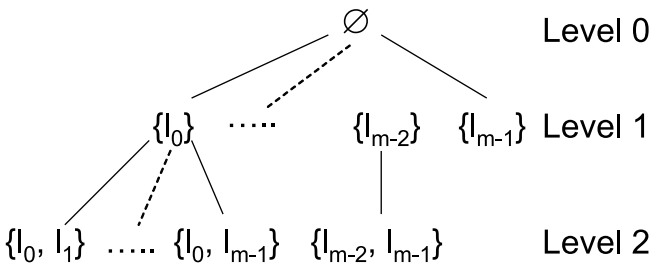


Fig. 1. The failure scenarios $s \in \mathcal{S}$ are organized in a tree structure such that each failure scenario is a child of one of its subsets.

graph $\mathcal{DG}_{s'}^{w}$ has changed with respect to $\mathcal{DG}_{s}^{w}$, calculates for these aggregates the new aggregate-specific rate vector $\mathbf{r}_{\mathbf{s}'}^{\mathbf{v},\mathbf{w}}$, and updates the vector for required link bandwidth $\hat{\mathbf{r}}_{\mathbf{s}'}$ incrementally.

Input: required bandwidth vector $\hat{\mathbf{r}}_{\mathbf{s}}$, failure scenarios s and s'
  $\hat{\mathbf{r}}_{\mathbf{s}'} \leftarrow \hat{\mathbf{r}}_{\mathbf{s}}$, $\Delta_s \leftarrow s' \setminus s$
  for all $w \in \mathcal{V}$ **do**
    $equal \leftarrow true$
    for all $x \in \Delta_s$ **do**
      if $Contains_s^w(x)$ **then**
        $equal \leftarrow false$
      end if
    end for
    if $equal \neq true$ **then**
      for all $v \in \mathcal{V}$ **do**
        $\mathbf{r}_{\mathbf{s}'}^{\mathbf{v},\mathbf{w}} = \mathbf{0}$
        RATEVECTORX($\mathcal{DG}_s^w, \mathbf{r}_{\mathbf{s}'}^{\mathbf{v},\mathbf{w}}, v, w, \mathbf{R}(v,w)$)
        $\hat{\mathbf{r}}_{\mathbf{s}'} \leftarrow \hat{\mathbf{r}}_{\mathbf{s}'} - \mathbf{r}_{\mathbf{s}}^{\mathbf{v},\mathbf{w}} + \mathbf{r}_{\mathbf{s}'}^{\mathbf{v},\mathbf{w}}$
      end for
    end if
  end for
Output: required bandwidth vector $\hat{\mathbf{r}}_{\mathbf{s}'}$

Algorithm 4: INCREMENTALREQUIREDBANDWIDTH: calculates the required bandwidth vector $\hat{\mathbf{r}}_{\mathbf{s}'}$ based on $\hat{\mathbf{r}}_{\mathbf{s}}$ incrementally.

2.3 Incremental Calculation Based on iSPF (R2Copy)

The incremental method based on iSPF has two advantages compared to the incremental naive calculation: it requires less effort to construct a new required destination graph $\mathcal{DG}_s^w$ and updates the bandwidth vector $\hat{\mathbf{r}}_{\mathbf{s}}$ by only those aggregates whose path has changed effectively.

Selective Path Calculation Using iSPF When a link l or node v within a destination graph $\mathcal{DG}_s^w$ fails, the new destination graph $\mathcal{DG}_{s'}^w$ must be constructed anew. The iSPF achieves that in an efficient way by copying the existing, similar destination graph $\mathcal{DG}_s^w$ and modifying it instead of computing it entirely from scratch. The paths from all nodes v that reach the destination node w only via a failed link or node in $\mathcal{DG}_s^w$ have then lost connection and must be rerouted. In addition, all nodes that contain the failed network element in their path need to recompute their rate vector afterwards using the algorithms in Section 2.1.

The recursive Algorithm 5 removes from the destination graph $\mathcal{DG}_s^w$ all network elements that have lost connection due to the failure of link l and adds the disconnected nodes to the set $\mathcal{R}$. The $|.|$-operator denotes the cardinality of a set and $l(u,v)$ is the link from node u to node v.

Input: failed link l, set of disconnected network elements $\mathcal{R}$
  $Pred(\omega(l)) \leftarrow Pred(\omega(l)) \setminus \{\alpha(l)\}$
  $Succ(\alpha(l)) \leftarrow Succ(\alpha(l)) \setminus \{\omega(l)\}$
  if $|Succ(v)| = 0$ **then** {v disconnected}
    $d(v) \leftarrow \infty, \mathcal{R} \leftarrow \mathcal{R} \cup \{v\}$
    for all $u \in Pred(v)$ **do**
      $RemoveLink(l(u,v), \mathcal{R})$
    end for
  end if

Algorithm 5: REMOVELINK: removes from the destination graph $\mathcal{DG}_s^w$ all network elements that have lost connection due to the failure of link l.

Algorithm 6 removes a node v from the destination graph by disconnecting it explicitly from all its successor nodes and by disconnecting it implicitly from all its predecessor nodes by calling REMOVELINK$(l, \mathcal{R})$ for all links leading to v. The failed node v is not added to the set of disconnected nodes since it should not be reconnected to the graph. Thus, its path contains then no elements.

Input: failed node v, set of disconnected network elements $\mathcal{R}$
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{v\}, \mathcal{C} \leftarrow \mathcal{C} \cup \{v\}$
  $d(v) \leftarrow \infty$
  for all $u \in Succ(v)$ **do**
    $Pred(u) \leftarrow Pred(u) \setminus \{v\}, Succ(v) \leftarrow Succ(v) \setminus \{u\}$
  end for
  for all $u \in Pred(v)$ **do**
    $RemoveLink(l(u,v), \mathcal{R})$
  end for

Algorithm 6: REMOVENODE: removes from the destination graph $\mathcal{DG}_s^w$ all network elements that have lost connection due to the failure of node v.

Algorithm 7 reconnects the disconnected working nodes in $\mathcal{R}$ by first connecting them to the connected structure of the remaining destination graph $\mathcal{DG}_s^w$ and moving then the freshly connected nodes to the tentative list $\mathcal{T}$. Finally, DIJKSTRA$(\mathcal{T}, \mathcal{R})$ is called and completes the destination graph $\mathcal{DG}_s^w$.

Input: set of disconnected working nodes $\mathcal{R}$
  for all $v \in \mathcal{R}$ **do**
    for all $\{l : \alpha(l) = v\}$ **do**
      if $d(\omega(l)) < \infty$ **then** {$\omega(l)$ has a path to w}
        if $d(\omega(l)) + cost(l) < d(v)$ **then**
          {a shorter path from v to w is found}
          if $d(v) = \infty$ **then**
            {move v from remaining to tentative list}
            $\mathcal{R} \leftarrow \mathcal{R} \setminus \{v\}$, $\mathcal{T} \leftarrow \mathcal{T} \cup \{v\}$
          end if
          $d(v) \leftarrow d(\omega(l)) + cost(l)$, $Succ(v) \leftarrow \{w\}$
        else if $d(\omega(l)) + cost(l) = d(v)$ **then**
          {an equal-cost path to v is found}
          $Succ(v) \leftarrow Succ(v) \cup \{w\}$
        end if
      end if
    end for
  end for
  $Dijkstra(\mathcal{T}, \mathcal{R})$

Algorithm 7: RECONNECTNODES: reconnects the disconnected working nodes in $\mathcal{R}$ to the destination graph $\mathcal{DG}_s^w$.

Incremental Calculation of the Required Link Bandwidth $\hat{\mathbf{r}}_{\mathbf{s}'}$ Based on Recalculated Destination Graphs $\mathcal{DG}_s^w$ The iSPF limits the overhead to reroute paths that are affected by a link or a node failure. In addition, the incremental update of the required link bandwidth can be limited to those nodes within a destination graph whose ECMP paths have changed. We find them by identifying the indirect predecessor nodes of a failed link or node within the base destination graph $\mathcal{DG}_s^w$. Algorithm 8 collects all predecessor nodes of the node v recursively and stores them in the set $\mathcal{C}$. At the beginning of the algorithm, the set of collected nodes is empty, i.e. $\mathcal{C} = \emptyset$. If a node v fails, we collect COLLECTINDIRECTPREDECESSOR$(v, \mathcal{C})$ and if a link l fails, we collect COLLECTINDIRECTPREDECESSOR$(\alpha(l), \mathcal{C})$. Finally, the set $\mathcal{C}$ contains all nodes that have a changed path layout in $\mathcal{DG}_{s'}^w$ compared to $\mathcal{DG}_s^w$. As a consequence, the incremental update of the bandwidth vector $\hat{\mathbf{r}}_{\mathbf{s}}$ in Algorithm 4 can be limited to the nodes in $\mathcal{C}$.

```
Input: node v, set of indirect predecessors C
  for all u ∈ Pred(v) do
    if u ∉ C then
      C ← C ∪ {u}
      CollectIndirectPredecessors(u, C)
    end if
  end for
```

Algorithm 8: CollectIndirectPredecessors: collects in the set $\mathcal{C}$ all indirect predecessor nodes of node v.

2.4 Incremental Calculation Based on iSPF with Reduced Copy Overhead (R2)

We discuss some implementation issues regarding an efficient memory management which finally leads to the improved version R2 with respect to R2Copy.

When the bandwidth requirements of many failure scenarios are computed, many destination graphs $\mathcal{DG}_s^w$ are sequentially constructed and evaluated. Deleting such a graph after its analysis and constructing a new, similar one requires quite an effort for memory allocation, which should be avoided if possible. The naive calculation in Section 2.1 recomputes all graphs from scratch. Therefore, the sets $Pred(v)$ and $Succ(v)$ of the old destination graph $\mathcal{DG}_s^w$ may be emptied and the new destination graph $\mathcal{DG}_{s'}^w$ may be constructed reusing the nodes from the old destination graph $\mathcal{DG}_s^w$. The incremental naive calculation in Section 2.2 recomputes all graphs $\mathcal{DG}_{s'}^w$ from scratch that have changed with regard to a predecessor destination graph $\mathcal{DG}_s^w$. The overall analysis traverses all failure scenarios of interest $\mathcal{S}$ recursively along a tree structure (cf. Figure 1). Thus, the destination graph for a specific destination w may change for each of the failure scenarios $s_0 \subset s_1 \subset ... \subset s_n$. Therefore, a complete set of nodes must be available on each level of the tree to construct the destination graph.

The incremental calculation based on the iSPF algorithm in Section 2.4 requires not only a new set of nodes but a copy of the destination graph $\mathcal{DG}_s^w$ that serves as a base for the construction of the destination graph $\mathcal{DG}_{s'}^w$ using iSPF. When $\mathcal{DG}_{s'}^w$ is not needed anymore, only the sets $Pred(v)$ and $Succ(v)$ of those nodes need to be reset that have been changed relative to the one in $\mathcal{DG}_s^w$. This saves the entire deletion of the current connectivity of $\mathcal{DG}_{s'}^w$ and generating a new copy of $\mathcal{DG}_s^w$.

3 Comparison of Experimental Computation Times

We implemented the above presented algorithms in Java $1.5.0\_06$ and test the computation time experimentally on a standard PC Pentium M, 1.86 GHz with 1 GB RAM and WinXP Pro SP2. We use random topologies in our study for which the most important network characteristics are the network size in terms of nodes $n = |\mathcal{V}|$ and links $m = |\mathcal{E}|$. They define the average node degree $\delta_{avg} = \frac{2 \cdot m}{n}$ that indicates the average number of adjacent links of a node and is thereby an indirect measure for the network connectivity. We use the topology generator from Section 4.4.2 in [13] to control the minimum and the maximum node degree δ_{min} and δ_{max} which are limited by the maximum deviation δ_{dev}^{max} of the node degrees from their average value. It generates connected networks and avoids loops and parallels.

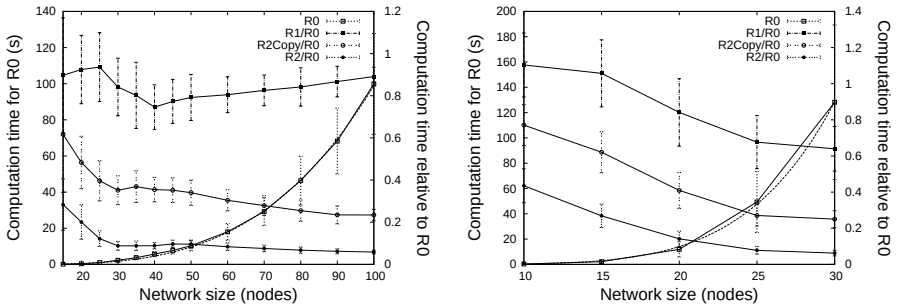
3.1 Comparison of Computation Times

We consider networks of different sizes with an average node degree $deg_{avg} \in \{3,4,5,6\}$ and a maximum deviation from the average node degree of $deg_{dev}^{max} \in \{1,2,3\}$. We randomly generate 5 networks of each combination. Figures 2(a) and 2(b) show the time for the computation of the ECMP routing and the link load for failures of single network elements and for failures of up to two network elements, respectively. The computation time is given in seconds for the naive method (R0) depending on the network size in nodes. The x-axes of both figures have a different scale since the calculation of the double failure scenarios is very time-consuming. We fit the experimental computation time of R0 by a function of the form $O(n^k)$ and derive k from an approximation that minimizes the sum of the squared deviations from the experimental results. In the single failure case, the experimental computation time grows approximately like $O(n^{3.36})$ (dashed line) with the number of nodes n in the network which results from a $O(n^2)$ worst case runtime of the Dijkstra algorithm and an $O(n)$ number of considered failure scenarios ($\binom{n+m}{0}+\binom{n+m}{1}$). In the double failure case, we observe a growths of about $O(n^{5.44})$ which is due to a larger number of failure scenarios ($\binom{n+m}{0}+\binom{n+m}{1}+\binom{n+m}{2}$). This is the practical runtime of the program for small network instances and all software overhead while the theoretical runtime of the mere algorithm is bounded by $O(n^4)$.

The computation time for the incremental naive method is presented relative to the one for R0. Surprisingly, the incremental naive method (R1) takes about the same time as the naive method (R0). Data structures for the implementation of the function $Contains_s^w(x)$ must be updated whenever the destination graph $\mathcal{DG}_s^w$ is reconstructed by R1. This makes the algorithm more complex. In addition, the destination graphs contain often more than the minimum number of $n-1$ links since equal-cost paths frequently occur due to our hop metric assumption, and must be updated in more than $\frac{n-1}{m}$ of all cases. As a consequence, the savings of destination graph calculations of R1 are too small to achieve a considerable speedup for its computation time compared to the one of R0. This is different for the incremental calculation based on iSPF (R2) which requires only 10% of the computation time of R0. This holds only, if the data structures are reused. If the data structures are copied (R2Copy), we still se significant savings of up to 75%, but compared to (R2), the computation time takes four times longer in large networks. The confidence intervals in both figures are based on a confidence level of 95% to guarantee that the results from our experiments are sufficiently accurate.

3.2 Sensitivity Analysis Regarding Network Connectivity

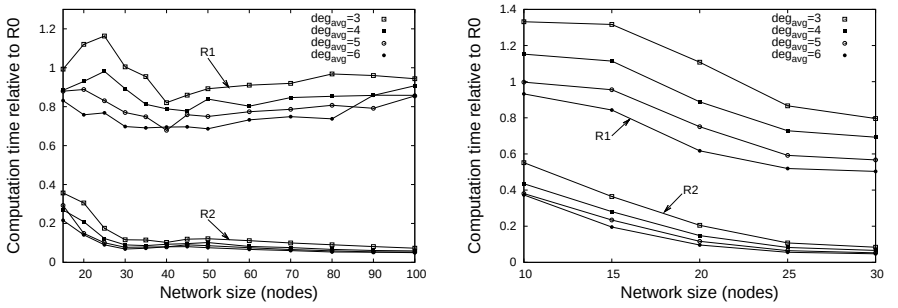
To underline the above observations, we conduct a sensitivity analysis of the computation time regarding the average node degree δ_{avg} of the networks. Figures 3(a) and 3(b) show the relative computation time of R1 and R2 compared to R0 separately for networks with different node degrees. The curves for both R1 and R2 show that networks with a large node degree like $\delta_{avg}=6$ lead to larger time savings than networks with small node degrees like $\delta_{avg}=3$. Networks with a large average node degree have more links than those with a small one, but their destination graphs contain approximately the same number of links since $(n-1)$ links already form a spanning tree. As a consequence, in networks with the same number of nodes but a larger number of links



(a) Single link or node failures. (b) Single and double link or node failures.

Fig. 2. Comparison of the computation time of the naive calculation (R0), the incremental naive calculation (R1), the incremental calculation based on iSPF with (R2) and without copy reduction (R2Copy).

it is less likely that a destination graph is affected by a link failure. Thus, they offer an increased savings potential for destination graph calculations. However, if the average node degree and the network size are small, the optimization method R1 can lead to clearly increased computation time and becomes counterproductive. These findings are very well visible if up to two network elements fail. For R2 we observe basically the same phenomenon, but its computation time is mostly limited to 20% or less of the one for R0. Hence, the proposed method R2 effectively reduces the computation time for programs that analyze the network resilience.



(a) Single link or node failures. (b) Single and double link or node failures.

Fig. 3. Comparison of the computation time of the incremental naive calculation (R1) and the incremental calculation based on iSPF with copy reduction (R2) depending on the node degree of the networks.

4 Conclusion

In this work we have presented a simple and a complex optimized method to speed up the calculation of the (re-)routing and the link load in a network for a large set of different failure scenarios. The reference model for our performance comparison is Dijkstra's shortest path first algorithm (R0). The simple method (R1) just skips the recalculation of a destination graph if it does not contain the failed network element. However, this achieves hardly any speedup. The complex method (R2) is based on an incremental shortest path first calculation and on a careful reuse strategy for data structures. It reduces the computation time to 10% while without the reuse strategy, the computation time is decreased to 25%. Hence, computer programs for the analysis of the network resilience should implement the complex method with a careful reuse strategy for data structures as it considerably accelerates the calculation of the routing and the traffic distribution for a large set of failure cases.

References

1. Moy, J.: RFC2328: OSPF Version 2 (1998)
2. ISO: ISO 10589: Intermediate System to Intermediate System Routing Exchange Protocol for Use in Conjunction with the Protocol for Providing the Connectionless-Mode Network Service (1992)
3. Oran, D.: RFC1142: OSI IS-IS Intra-Domain Routing Protocol (1990)
4. Iannaccone, G., Chuah, C.N., Bhattacharyya, S., Diot, C.: Feasibility of IP Restoration in a Tier-1 Backbone. IEEE Network Magazine (Special Issue on Protection, Restoration and Disaster Recovery) (2004)
5. McQuilan, J.M., Richer, I., Rosen, E.C.: The New Routing Algorithm for the ARPANET. IEEE Transactions on Communications **28** (1980)
6. Francois, P., Filsfils, C., Evans, J., Bonaventure, O.: Achieving Sub-Second IGP Convergence in Large IP Networks. ACM SIGCOMM Computer Communications Review **35** (2005) 35 – 44
7. Vasseur, J.P., Pickavet, M., Demeester, P.: Network Recovery. 1. edn. Morgan Kaufmann / Elsevier (2004)
8. Narvaez, P.: Routing Reconfiguration in IP Networks. PhD thesis, Massachusetts Institut of Technology (MIT) (2000)
9. El-Sayed, H., Ahmed, M., Jaseemuddin, M., Petriu, D.: A Framework for Performance Characterization and Enhancement of the OSPF Routing Protocol. In: IASTED International Conference on Internet and Multimedia Systems and Applications (EuroIMSA), Grindelwald, Switzerland (2005)
10. Nelakuditi, S., Lee, S., Yu, Y., Zhang, Z.L.: Failure Insensitive Routing for Ensuring Service Availability. In: IEEE International Workshop on Quality of Service (IWQoS). (2003)
11. Buriol, L., Resende, M., Thorup, M.: Speeding up Dynamic Shortest Path Algorithms. Technical Report TD-5RJ8B, AT&T Labs Research (2003)
12. Disjkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik **1** (1959) 269 – 271
13. Menth, M.: Efficient Admission Control and Routing in Resilient Communication Networks. PhD thesis, University of Würzburg, Faculty of Computer Science, Am Hubland (2004)