Alcom-FT Technical Report Series
ALCOMFT-TR-03-193

A Framework for Network Reliability Problems on Graphs of Bounded Treewidth*

Thomas Wolle

Institute of Information and Computing Sciences,
Utrecht University, The Netherlands

Abstract. In this paper, we consider problems related to the network reliability problem restricted to graphs of bounded treewidth. We consider undirected simple graphs with a rational number in $[0, 1]$ associated to each vertex and edge. These graphs model networks in which sites and links can fail with a given probability, independently of whether other computers or links fail. The number in $[0, 1]$ associated to each element is the probability that this element does not fail. In addition, there are distinguished sets of vertices: a set S of servers, and a set L of clients. This paper presents a dynamic programming framework for graphs of bounded treewidth. For a large number of different properties Y, we can compute with this framework the probability that Y holds in the graph formed by the nodes and edges that did not fail. For instance, it is shown that one can compute in linear time the probability that all clients are connected to at least one server, assuming the treewidth of the input graph is bounded. The classical L-terminal reliability problem can be solved in linear time for graphs of bounded treewidth as well using this framework. The method is applicable to a large number of related questions. Depending on the particular problem, the algorithm obtained by the method uses linear, polynomial, or exponential time.

1 Introduction

In a world with a growing need for communication and data exchange, computer as well as telecommunication networks play a decisive role. In most applications, it is of great importance that connections between the communicating parties remain intact. Unfortunately, hardware and software are not infallible. For this reason modern networks are designed such that two communication sites can have more than one possibility of information exchange. Furthermore, one can often observe that some sites in a network are of greater importance than others. For instance, connections between servers may be more important than between clients. It is important to compute the reliability of a network not only during construction process but also for improving and determining the quality of existing networks.

In this paper, we investigate the problem of computing the probabilities that a given network has a certain property, for several types of properties. We use the following model: To each element of the network (node or edge), a rational value in $[0, 1]$ is assigned, which expresses the probability that the element is working. We assume that the occurrence of failures of elements are statistically independent. Given a network $G = (V, E)$, with associated probabilities of non-failure, and a set of vertices $S \subseteq V$, the S-terminal reliability problem asks for the probability that there is a connection between every pair of vertices of S. For general graphs or networks this problem is $\#P$-complete [10]. When restricting the set of input networks to graphs of bounded treewidth, linear time algorithms are possible.

Much work has been done on algorithms for graphs of bounded treewidth, as well as for (special cases of) the S-terminal reliability problem. A general description of a common

* This research was partially supported by EC contract IST-1999-14186: Project ALCOM-FT (Algorithms and Complexity - Future Technologies).

'bottom up' method for graphs of bounded treewidth can be found in [3] or [4]. Arnborg and Proskurowski considered in [1] the case that only edges can fail and that nodes are perfectly reliable. They gave a linear time algorithm for this problem on partial k-trees, i.e., graphs of treewidth at most k. Mata-Montero generalized this to partial k-trees in which edges as well as nodes can fail [9]. Rosenthal introduced in [11] a decomposition method, which was applied by Carlier and Lucet to solve the network reliability problems with edge and node failures [5]. Furthermore, with this approach Carlier, Manouvrier and Lucet solved the 2-edge connected reliability problem for graphs of bounded pathwidth [8].

We will generalize both the type of the problems which can be solved, and the method used in [5] and [8]. Instead of looking at graphs of bounded pathwidth, we will deal with graphs of bounded treewidth.

Since elements of the network can break down (i.e. the corresponding vertices are deleted from the graph), we call the graph induced by the still existing vertices (i.e. the correspondents to the 'up' network elements) the 'surviving subgraph'. For graphs with associated non-failure probabilities, we present a framework for answering a large number of questions, which can be more complicated and more general than the classic S-terminal reliability problem. These questions ask for the probability that the surviving subgraph has a certain property. We allow that there are two types of special vertices. We have a set S of servers and a set L of clients as well as other vertices that serve simply to build connections. With this technique, we can answer questions such as: 'What is the probability that all clients are connected to at least one server?' and 'What is the expected number of components of the graph of the non-failed elements that contain a vertex of S or L?' Many of these questions are shown to be solvable in linear or polynomial time when G has bounded treewidth.

The properties one can ask for, define the connectivity between vertices of the set S, and also the connectivity between vertices of S and vertices of L. Furthermore, the number of connected components which contain a vertex of S or L can be dealt with. The more information needed to answer a question, the higher the running time of the algorithm. This starts with linear time and can increase to exponential time.

Algorithms that require exponential time are necessary for solving problems in which most or all of the possible information must be preserved. Hence no diminution of information or run time is possible.

The structure of the paper is described in this paragraph. In Section 2, we give definitions, which are important for the technique described in Section 3. In Section 3, we describe the dynamic programming approach that exploits a nice treedecomposition. In Section 4, we look at how to use equivalence classes in our framework. We consider the properties of problems that are solvable with our method. We describe how to reduce the number of objects needed during the computation, and hence how to reduce the running time by using equivalence relations. We discuss the structure of these equivalence relations and the correctness of using them. In Section 5, we list some solvable problems, give a mechanism to generate certain equivalence relations automatically, and show how these reduce the running time. The equivalence relations of problems that ask for the probability that the surviving subgraph fulfils a number of properties, are also discussed in Section 5. In Section 6, we look at an example that demonstrates how the method can be extended through additional ideas to gain a better running times. The paper is concluded by the discussions given in Section 7. There, we consider the consequences of simulating edge failures, using a pathdecomposition instead of a treedecomposition, and we examine the global running times.

2 Definitions

2.1 Basic Definitions

We give some basic definitions needed to define graphs with bounded treewidth and the network reliability problem. For more information on treewidth and the network reliability problem, see references [1], [3], [4], [9], [11].

Definition 1. *A* treedecomposition *of a graph* $G = (V, E)$ *is a pair* (T, X) *with* $T = (I, F)$ *a tree, and* $X = \{X_i \mid i \in I\}$ *a family of subsets of* V*, one for each node of* T*, such that*

- $\bigcup_{i \in I} X_i = V$.
- *for all edges* $\{v, w\} \in E$ *there exists an* $i \in I$ *with* $\{v, w\} \subseteq X_i$.
- *for all* $i, j, k \in I$: *if* j *is on the path in* T *from* i *to* k*, then* $X_i \cap X_k \subseteq X_j$.

The width *of a treedecomposition* $((I, F), \{X_i \mid i \in I\})$ *is* $\max_{i \in I} |X_i| - 1$. *The* treewidth *of a graph* G *is the minimum width over all treedecompositions of* G. *A treedecomposition* (T, X) *is* nice*, if* T *is rooted and binary, and the nodes are of four types:*

- Leaf nodes i *are leaves of* T *and have* $|X_i| = 1$.
- Introduce nodes i *have one child* j *with* $X_i = X_j \cup \{v\}$ *for some vertex* $v \in V$.
- Forget nodes i *have one child* j *with* $X_i = X_j \setminus \{v\}$ *for some vertex* $v \in V$.
- Join nodes i *have two children* j_1, j_2 *with* $X_i = X_{j_1} = X_{j_2}$

A treedecomposition can be converted into a nice treedecomposition with $|I| = O(V|G|)$ of the same width in linear time (see [7], [4]). The properties of a nice treedecomposition will in general not provide more algorithmic power. However, designing algorithms is often easier when using a nice treedecomposition.

A subgraph of $G = (V, E)$ induced by $V' \subseteq V$ is a graph $G[V'] = (V', E')$ with $E' = E \cap (V' \times V')$. A graph G can consist of connected components $O_1, ..., O_q$. In that case, we write $G = O_1 \cup ... \cup O_q$, where $\cup$ denotes the disjoint union of graphs, and each O_i $(1 \leq i \leq q)$ is connected. Throughout this paper, $G = (V, E)$ denotes a fixed, undirected, simple graph, with V the set of vertices and E the set of edges. $T = (I, F)$ is a nice treedecomposition of the graph G. The term vertex refers to a vertex of the graph and the term node refers to a vertex of the tree (decomposition).

A vertex v of the network or graph is either *up* or *down*. That means the network site represented by v is either functioning or it is in a failure state. The probability that v is up is denoted by $p(v)$. A *scenario* f assigns to each vertex its state of operation: v is up ($f(v) = 1$) or down ($f(v) = 0$). Hence, a scenario describes which vertices are up and which are down in the network. For the scenario f, $W^{f=1}$ is the set of all vertices of W which are up and $W^{f=0} = W \setminus W^{f=1}$, for $W \subseteq V$. The probability of a scenario f (for the whole graph G) is:

$$\Pr_V(f) = \prod_{v \in V^{f=1}} p(v) \cdot \prod_{v \in V^{f=0}} 1 - p(v)$$

Clearly, there are $2^{|V|}$ scenarios. The elements in $L \subseteq V$ represent special objects, the *clients*, and the elements of $S \subseteq V$ are the *servers*, where $n_S = |S|$ and $n_L = |L|$. We say $W \subseteq V$ is connected in G, if for any two vertices of W there is a path joining them. One can easily simulate edge failures by introducing a new vertex on each edge, and then using the methods for networks with only node failures. Thus, to ease presentation, we will assume from now on that edges do not fail. For more details on how to simulate edge failures, see Section 7.1.

Summary of Notations: Let $G = (V, E)$ be a graph.

- $p : V \to [0, 1]$ assigns to each vertex its probability to be up
- $f : V \to \{0, 1\}$ assigns to each vertex its state for this scenario f
- $W^{f=q} = \{v \in W \mid f(v) = q\}$, for $q \in \{0, 1\}$, $W \subseteq V$, $f : V \to \{0, 1\}$
- $L \subseteq V$ is the set of clients of G, $n_L = |L|$
- $S \subseteq V$ is the set of servers of G, $n_S = |S|$

2.2 Blocks and Representatives

In this section, the definitions of blocks and representatives are given. These notions play an important role in the main methods given in this paper. We assume to have a nice treedecomposition $(T, X) = ((I, F), \{X_i \mid i \in I\})$ of $G = (V, E)$. Since T is a binary rooted tree, the ancestor relation of nodes of the tree is well defined. As described in many articles, e.g. [3] and [4], we will use a bottom up approach with this treedecomposition. Each node i of the tree can be viewed as the root of a subtree. We will compute partial solutions for the subgraphs corresponding to these subtrees step by step. These partial solutions are contained in the roots of the subtrees. To compute this information, we only need the information stored in the children of a node, which is typical for dynamic programming algorithm using treedecompositions. The following definition provides us the terminology for the subgraphs.

Definition 2. *Let $G = (V, E)$ be a graph and $(T = (I, F), X = \{X_i \mid i \in I\})$ a nice treedecomposition of G.*

- $V_i = \{v \in X_j \mid j = i$ *or* j *is a descendant of* $i\}$
- $G_i = G[V_i]$
- $L_i = L \cap V_i$
- $S_i = S \cap V_i$

Blocks. For a certain subgraph G_i, we consider the scenarios of G restricted to V_i. A scenario f^1 causes G_i to decompose into one or more connected components $O_1, ..., O_q$, i.e. $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$. For these components, we consider the intersection with X_i. These intersection sets $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$ are called *blocks* and we use the following notation:

$$blocks_i(f^1) = \{B_1, ..., B_q\}$$

Thus, each scenario f^1 specifies a multiset of blocks. It is a multiset since more than one component may have an empty intersection with X_i. When using the symbol '$\cup$' between multisets (of blocks), it means the 'multiunion' of the sets, i.e. multiple occurrences are not deleted and the result is again a multiset.

We augment the blocks with L-flags and/or S-flags. Since we are looking at reliability problems with a set L of clients and a set S of servers, it is important to store the information whether $O_j \cap L^{f=1} \neq \emptyset$ and/or $O_j \cap S^{f=1} \neq \emptyset$. If this is the case we add $|O_j \cap L^{f=1}|$ L-flags and/or $|O_j \cap S^{f=1}|$ S-flags to the block B_j. Hence, the blocks for a scenario f reflect the connections between the vertices of X_i in $G[V_i^{f=1}]$ as well as the number of servers and clients of the components of $G[V_i^{f=1}]$. We use the notation '$\#X$flags(B)' to refer to the number of X-flags of block B, $(x \in \{S, L\})$.

Definition 3. *A* block B *of the graph* G_i *for scenario* f *is a (perhaps empty) subset of* X_i *extended by* x L*-flags and* y S*-flags. For* $\{v_1, ..., v_t\} \subseteq X_i$*, we write* $B = \{v_1, ..., v_t\}_{y \cdot S}^{x \cdot L}$*.*

For small x and y instead of $B = \{v_1, ..., v_t\}_{y \cdot S}^{x \cdot L}$, we can also write $B = \{v_1, ..., v_t\}_{S..S}^{L..L}$ where the number of L-flags equals x and the number of S-flags equals y.

Example Blocks. Let $X_i = \{u, v, w\}$. Depending on G_i, example blocks could be:

- $B_1 = \{uv\}$; u and v are up and connected within one component. This component contains no vertex of S nor of L.
- $B_2 = \{\ \}_S^{LL}$; this represents a component with empty intersection with X_i. Furthermore this component contains two clients and one server.
- $B_3 = \{w\}_{SS}$; this represents a component with nonempty intersection with X_i, namely w. This component contains no clients but two servers.

Representatives. For $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$, we have the following blocks $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$, which can be used to represent f^1. A block B of such a representation has x L-flags, if and only if the component O corresponding to B contains exactly x vertices of L. The same is also true for the S-flags.

To each scenario of node i, we associate the multiset of its blocks. We call this representation of a scenario f^1 its *representative* C^1:

$$C^1 = blocks_i(f^1) = \{B_1, ..., B_q\}$$

for $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$ and $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$. To G_i we associate a multiset of representatives, one for each possible scenario. Note that two different scenarios f^1 and f^2 can have identical representatives $C^1 = C^2$, but these will be considered as different objects in the multiset of the representatives of G_i. In Section 4 of this paper equivalence relations on scenarios and hence on representatives are defined. Thus equivalence classes will be formed that can correspond to more than one scenario. It is then possible and perhaps also sensible to define C^1 and C^2 as equivalent. Furthermore, it will be seen that the algorithmic techniques for computing the collection of representatives of the graphs G_i and their corresponding probabilities will carry over with little modification to algorithms for computing the collections of equivalence classes for many equivalence relations.

Definition 4. *A* representative $C^1 = \{B_1, ..., B_q\}$ *of* G_i *is a multiset of blocks of* G_i*, such that the scenario* f^1 *specifies* $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$*. It must hold that #Lflags(*B_j*)=* $|O_j \cap L|$ *and #Sflags(*B_j*)=* $|O_j \cap S|$*, for* $j = 1, ..., q$*, and* $G[V_i^{f^1=1}] = O_1 \cup ... \cup O_q$*.*

The number of representatives per node is a crucial point to the algorithm running time as will be demonstrated. In the Section 4.2 we examine equivalence relations between scenarios to reduce the number of representatives.

Example Representatives. Let $X_i = \{u, v, w\}$. Some of the possible representatives are:

- $C^1 = (\{uv\}_S^{LL}\{w\}\{\ \}^L\{\ \}_S)$; all vertices are up, u, v are in one component containing two clients and one server, w is in another component without clients and servers. There are two components with empty intersection with X_i, one of them contains a client the other a server.

- $C^2 = (\{vw\}\{\ \}\{\ \}\{\ \}_S)$; v and w are up and connected. There are three components with empty intersection with X_i, one of them contains a server.
- $C^3 = (\{vw\}\{\ \}_S)$; v and w are up and connected. There is one component with empty intersection with X_i and this component contains a vertex of S.

Here, the representatives C^2 and C^3 are somehow 'similar'. They only differ in the number of empty blocks without flags. Such blocks neither give information about clients or servers nor can be used to create additional connections. Hence, it can be sensible to define C^2 and C^3 to be equivalent (see Section 4.2).

3 A Technique for Solving Network Reliability Problems with Treedecompositions

In this section, we describe a method where treedecompositions of a network can be used to solve variations of the network reliability problem. As already mentioned, we compute the set of representatives for every node of the nice treedecomposition using a bottom up approach. We will not only compute the representatives C but their associated probabilities $\Pr(C)$ as well. For this computation, we give algorithms for each of the four types of nodes that are used in a nice treedecomposition. Before looking at these algorithms, we consider a subroutine to repair the structure of a representative. This subroutine is used in the algorithms for introduce- and join-nodes.

For a representative C^1 of node i, $\Pr(C^1)$ is the probability that G_i is in scenario f^1. To compute $\Pr(C^1)$, we could simply compute $\Pr(f^1)$. We could do this for all representatives C of node i. Nevertheless, Lemma 1 and 2 and similar easy statements for leaf and forget nodes, will make evident the fact that we only need the already computed representatives and their probabilities of the children of the node, insofar as they exist. This is a relevant point for applying the dynamic programming scheme, described in [3] or [4].

Starting at the leaves of the nice treedecomposition, we compute for each node all representatives and their probabilities that are possible for this node. In this computation, the representatives of the child(ren) of a node play a decisive role. When 'walking up' the treedecomposition to a node i the representatives will be refined and split into other representatives representing the states of the vertices of X_i and their connections. Later in Section 4.2 we will see that representatives are not only refined, while walking up the tree, but at the same time, some representatives will 'collapse' into one equivalence class, since they become equivalent. Once we have the probabilities of all representatives of all nodes, especially of the root, we can easily solve problems by computing the requested probabilities.

3.1 Repairing Representatives.

Before considering the particular algorithms, we describe a procedure that is a subroutine of the introduce- and join-node-algorithm. It is used to repair the structure of a representative. The procedure REPAIR receives as input a multiset of blocks. This multiset is not necessarily a representative, as some blocks may contain common vertices in X_i. The REPAIR procedure 'repairs' or 'corrects' the multiset by unifying such blocks.

By introducing a new vertex (see Section 3.2 for the algorithm) or joining two representatives (Section 3.2), a representative can contain blocks with nonempty intersection. Between two components O_a and O_b corresponding to such blocks B_a and B_b exists a connection via a vertex of $B_a \cap B_b$. This means that after introducing a new vertex or joining representatives, we have a larger component $O = O_a \cup O_b$, which has to be represented by only one block B. Having two blocks with a nonempty intersection hurts the proper structure of a representative. Hence, we *merge* or *unify* such non-disjoint blocks B_a and B_b into one block B. Since blocks are sets of vertices, multiple appearances of a vertex are not possible. The new component O contains all the clients (i.e. vertices $v \in L$) of O_a and O_b. We have to add to B the number of L-flags of blocks B_a and the number of L-flags of B_b. On the other hand, we must subtract the number of clients which belong to both blocks B_a and B_b, because they are counted twice. The same applies to servers. Here is the code of the procedure to repair representative C:

```
REPAIR(C)

  while ∃B_a, B_b ∈ C with a ≠ b and B_a ∩ B_b ≠ ∅ do
      B := B_a ∪ B_b
      add (#Lflags(B_a) + #Lflags(B_b) - |L ∩ B_a ∩ B_b|)
          L-flags to B;
      add (#Sflags(B_a) + #Sflags(B_b) - |S ∩ B_a ∩ B_b|)
          S-flags to B;
      C := {B} ∪ C \ {B_a, B_b}
  endwhile
```

The while loop can be implemented to take at most $O(n_b^2)$ iterations, where n_b is the maximum number of blocks in one representative. One block can have at most $k = |X_i|$ elements. Thus, the condition and the body of the while loop is executable in $O(k^2)$ steps. The REPAIR algorithm needs altogether $O(k^2 \cdot n_b^2)$ time.

3.2 Computation of the Representatives and their Probabilities

In the following we use i to denote a node of the tree T. Node i may have $0, 1$ or 2 children, depending on the type of i. These children are referred to as j or j_1, j_2. Accordingly, C_i denotes a representative of i, which is computed using the representative(s) C_j (C_{j_1}, C_{j_2}) of node(s) j (j_1, j_2), respectively. We will have a look at each type for node i. When handling representatives, a lower index refers to the node the representative belongs to, and an upper index is used to differentiate different representatives belonging to the node considered.

Leaf Nodes. For leaves i of T, we have a relatively simple algorithm, since $|X_i| = 1$, and leaves have no children. Let $v \in X_i$. So we have the following possibilities, either v is up or down, and we can have $v \in L$ or $v \notin L$ and $v \in S$ or $v \notin S$. We distinguish these cases:

- v is up: this results in a representative $C_i = \{ \{v\} \}$, with $\Pr(C_i) = p(v)$. We add C_i to the set of representatives of node i.
- v is down: here we have another representative $C_i' = \{ \}$, with $\Pr(C_i') = 1 - p(v)$. Again, we add it to i.

If v is up and $v \in L$ and/or $v \in S$ we must add one L- and/or S-flag to the block. An algorithm performing the appropriate steps is very simple and would clearly need constant time per leaf. Since $|V_i| = 1$, we have only two scenarios.

Introduce Nodes. Suppose i is an introduce node with child j, such that $X_i = X_j \cup \{v\}$. We use the representatives and their probabilities of node j to compute the representatives and their probabilities of node i. For any state of v (up or down, in L or not in L, in S or not in S) we compute the representatives and add them to the set of representatives of node i. This is done by considering the following two cases for each representative C_j of j:

- v is up: we add v to each block $B \in C_j$ which contains a neighbour of v, since there is a connection between v and the component corresponding to B. If v is a client or a server, then we add an L- or S-flag to block B. Now, we apply the algorithm in Section 3.1 to repair the structure of a representative. The probability of this new representative C_i is: $\Pr(C_i) = p(v) \cdot \Pr(C_j)$.
- v is down: we do not make any changes to the representative C_j to get C_i, since no new connections can be made and no flags were added. However, we compute the probability: $\Pr(C_i) = (1 - p(v)) \cdot \Pr(C_j)$.

The following code formalises this procedure:

Introduce-Node-Algorithm

```
for every representative C_j of node j do

/* v is up */
  C_i := C_j
  for each block B of C_i do
      if B ∩ N(v) ≠ ∅
      then B := B ∪ {v}
           if v ∈ L then add one L-flag to B endif
           if v ∈ S then add one S-flag to B endif
      endif
  endfor
  REPAIR(C_i)
  Pr(C_i) := Pr(C_j) · p(v)
  add C_i to the set of representatives of node i

/* v is down */
  C_i := C_j
  Pr(C_i) := Pr(C_j) · (1 − p(v))
  add C_i to the set of representatives of node i
endfor
```

Lemma 1. *Given all representatives and their probabilities of the child j of an introduce node i, the introduce-node-algorithm computes the representatives and their probabilities for node i correctly.*

Proof. Note that we can compute the representatives and their probabilities for node i by considering every possible scenario for node i. However, recall that v is the only vertex in $V_i \setminus V_j$. So, each scenario for node i can be obtained by taking a scenario for node j and extending it by specifying whether v is up or down. All possible scenarios for node j are represented by the representatives of j, whose probabilities are known. For that reason, it is sufficient to combine the representatives of j with v to create the representatives of i. To do this we consider both v being up and down for each representative C_j of j and make the appropriate modifications. In this way, we implicitly look at every scenario which is possible for G_i.

It is not hard to see that the algorithm realises the description above. To demonstrate that this is true, we consider the representative C_j of a scenario f of j, hence $C_j = \mathit{blocks}_j(f)$. Two cases: v is up, or v is down, have to be considered:

Case I: v is up. We extend f by $f(v) = 1$, i.e. for all $w \in V_j : f(w)$ is unchanged. Since v may have neighbours in blocks of C_j, we must add v to these blocks. There is a connection between v and these neighbours and hence, between all vertices of the component represented by this block. Furthermore, if v is a client or a server then the component containing v has one more client or server, respectively. Thus, we have to add an L- or S-flag to such a block. That is exactly what happens in the inner for loop. After this we can have nonempty blocks with nonempty intersection, namely v. We use the REPAIR algorithm to re-establish the structure of the representative. As a final result, we obtain a new representative C_i, which we add to the set of representatives of i. The probability of C_i is:

$$\Pr(C_i) = \Pr_{V_i}(f) = \Pr(C_j) \cdot p(v)$$

Case II: v is down. Similar to Case I, we extend scenario f by setting v as down. Hence, no new connections between vertices of blocks or components can be made. Thus, we do not modify C_j to get the new representative C_i. The probability of C_i is:

$$\Pr(C_i) = \Pr(C_j) \cdot (1 - p(v))$$

□

We can now analyse the running time of the algorithm. The outer for-loop is executed n_c times, the inner for-loop n_b times, whereby n_c is the maximum number of representatives for a node, and n_b is the maximum number of blocks of a representative and k is the treewidth. Since we have to check only $O(k)$ neighbours, the condition $B \cap N(v) \neq \emptyset$ can be tested in $O(k^2)$ steps. Hence, the algorithm needs time $O(n_c \cdot n_b^2 \cdot k^2)$.

Forget Nodes. Node i is a forget node with child j, such that $X_i = X_j \setminus \{v\}$. Note that $V_i = V_j$, thus i and j have the same scenarios. A representative C_i of i differs from the corresponding representative C_j of j only because v does not appear in the blocks of C_i. Thus, we modify every representative C_j of j in the following way to create a representative C_i of node i. We simply delete v from every block of C_j to obtain C_i. As C_i and C_j represent the same scenarios, we have $\Pr(C_i) = \Pr(C_j)$. A simple algorithm would need time linear in $n_c \cdot k$ to create all new representatives. n_c is the maximum number of representatives for a node and k is the treewidth.

Join Nodes. For join nodes i with children j_1 and j_2 we have: $X_i = X_{j_1} = X_{j_2}$. Note that $V_i = V_{j_1} \cup V_{j_2}$, and that $V_{j_1} \cap V_{j_2} = X_i$. Let $\mathit{up}(C)$ denote the set of up vertices of X_i for a representative C of node i.

Definition 5. *Let C be a representative.*

$$\mathit{up}(C) := \{v : v \in B \text{ for } B \in C\}$$

Two representatives C^1 and C^2 are compatible if $\mathit{up}(C^1) = \mathit{up}(C^2)$.

Each scenario f of node i can be seen as the 'merge' of its restriction f_1 to V_{j_1} and its restriction f_2 to V_{j_2}. Conversely, scenarios f_1 of j_1, and f_2 of j_2 can be merged, if and only if the same vertices in X_i are up, i.e. corresponding representatives must be compatible.

A representative C_{j_1} of j_1 represents states and connections of vertices of X_{j_1}. The same is true for a representative C_{j_2} of j_2. Note that there is no connection between vertices of a representative, if they belong to two different blocks.

Combining two compatible representatives means checking for new connections, since this translates to combining the corresponding subgraphs. If we have a representative C_{j_1} with two nonempty blocks B_1 and B_2, this means there is no connection in G_{j_1} between vertices of the corresponding components. Note that there is no connection between two blocks of a representative if they are disjoint. However, when combining this representative with a compatible representative C_{j_2}, it is possible that there will be a connection between B_1 and B_2 via a connection of C_{j_2}. Therefore, to combine two representatives, we unite all blocks with nonempty intersection. Specifically this is done by the REPAIR algorithm given in Section 3.1. Since the probabilities of the states of vertices of X_i are 'contained' in $\Pr(C_{j_1})$ and in $\Pr(C_{j_2})$ as well and since the probabilities of the vertices of $V_i \setminus X_i$, being in their appropriate state, are independent, we compute

$$\Pr(C_i) = \frac{\Pr(C_{j_1}) \cdot \Pr(C_{j_2})}{\Pr_{X_i}(C_i)}$$

with

$$\Pr_{X_i}(C_i) = \prod_{v \in up(C_i)} p(v) \quad \cdot \quad \prod_{v \in X_i \setminus up(C_i)} 1 - p(v)$$

Again, we give pseudocode to formalize the complete method for join nodes:

Join-Node-Algorithm

```
for every representative C_j1 of node j1 do
  for every representative C_j2 of node j2 do
    if up(C_j1) = up(C_j2)
    then C_i := C_j1 ∪ C_j2
         REPAIR(C_i)
         Pr(C_i) = Pr(C_j1) · Pr(C_j2) ÷ Pr_X_i(C_i)
         add C_i to the set of representatives of node i
    endif
  endfor
endfor
```

Lemma 2. *Given all representatives and their probabilities of the children j_1 and j_2 of a join node i, the join-node-algorithm computes the representatives and their probabilities for node i correctly.*

Proof. We consider a representative C_i of node i. C_i represents a scenario f with $C_i = blocks_i(f)$. Such a scenario f determines two scenarios f_1 of G_{j_1} and f_2 of G_{j_2}. These scenarios f_1 and f_2 are represented by representatives C_{j_1} and C_{j_2} of node j_1 or j_2, respectively. C_{j_1} and C_{j_2} are compatible and already computed. Hence, we combine them to get representative C_i. Thus, it is sufficient to combine a representative of j_1 with a compatible one of j_2. By this means, we compute all possible representatives of i.

The search for and processing of compatible representatives is done in the body of the two nested for-loops. Two compatible representatives C_{j_1} and C_{j_2} represent a scenario f of

G_i or its restrictions f_1 and f_2 to G_{j_1} and G_{j_2}, respectively. They correspond to components of G_{j_1} and G_{j_2}. If we find one block B_1 in C_{j_1} and another block B_2 in C_{j_2} with $B_1 \cap B_2 \neq \emptyset$, this means that there is a path from a vertex of the component represented by B_1 to a vertex of the component represented by B_2, via a vertex of $B_1 \cap B_2$. In the new representative for i these two components are connected and hence make up one component, with the number of servers and clients added together. Thus, we must join both blocks B_1 and B_2 and modify the number of flags. This is done by using the REPAIR algorithm, described in Section 3.1.

Before demonstrating that the formula for computing $\Pr(C_i)$ is correct, we introduce and repeat the following notations:

$$\Pr_W(f) = \prod_{v \in W^{f=1}} p(v) \cdot \prod_{v \in W^{f=0}} 1 - p(v); \quad \text{for a scenario } f \text{ and } W \subseteq V$$
$$\Pr_{X_i}(C) = \prod_{v \in up(C)} p(v) \cdot \prod_{v \in X_i \setminus up(C)} 1 - p(v); \quad \text{for a set of blocks } C$$

We look at representative C_i of node i which is the combination of representative C_{j_1} of j_1 and C_{j_2} of j_2. Since $\Pr_{X_i}(C_i) = \Pr_{X_i}(f)$ with $C_i = blocks_i(f)$, we have:

$$\begin{aligned}
\Pr(C_i) = \Pr_{V_i}(f) &= \Pr_{V_{j_1}}(f_1) \cdot \Pr_{V_{j_2}}(f_2) \cdot \Pr_{X_i}(C_i) \\
&= \frac{1}{\Pr_{X_i}(C_i)} \cdot \Pr_{V_{j_1}}(f_1) \cdot \Pr_{X_i}(C_i) \cdot \Pr_{V_{j_2}}(f_2) \cdot \Pr_{X_i}(C_i) \\
&= \frac{1}{\Pr_{X_i}(C_i)} \cdot \Pr_{V_i}(f_1) \cdot \Pr_{V_i}(f_2) \\
&= \frac{1}{\Pr_{X_i}(C_i)} \cdot \Pr(C_{j_1}) \cdot \Pr(C_{j_2})
\end{aligned}$$

□

The two nested for-loops have $O(n_c^2)$ iterations. At most k^2 steps are required for the if-condition and $O(k^2 \cdot n_b^2)$ for the REPAIR algorithm. Altogether, we have $O(n_c^2 \cdot n_b^2 \cdot k^2)$ steps. n_c is the maximum number of representatives for a node, n_b is the maximum number of blocks of a representative and k is the treewidth.

4 The Framework and Equivalence Classes

The classic S-terminal reliability problem can be expressed as the question for the probability that the surviving subgraph has a certain property. In this example, the property would be the connection between all pairs of vertices of S. The framework given in the previous section can handle more complex properties, and hence can answer the appropriate question for the probability of such more complex properties. In this section, we take a close look at properties that can be handled by the given framework. We also consider equivalence relations between scenarios (i.e. their representatives). With the help of equivalence relations, it is possible to speed up the entire method. Of course, such relations must meet certain requirements for a given problem, which will also be discussed.

4.1 Properties of Solvable Network Reliability Problems

Our results which can be obtained with the method described above, can be divided into two groups. The first group does not use an equivalence relation between scenarios. Its consideration is a kind of preparation or warm up for the second group which uses equivalence relations between scenarios (see Section 4.2).

So far, a representative C of a node i represents only one scenario f of G_i, and $\Pr(C)$ is the probability that this scenario occurs for G_i. $C = \mathit{blocks}_i(f)$ contains information about the connectivity of vertices of S and L. More precise, the blocks of C reflect the components and the flags the number of vertices of S and L in each component of the graph $G[V_i^{f=1}]$. Since we are dealing with network reliability problems, the term 'property of the graph' should refer to a property concerning the components and connectivity of vertices of S and L. Hence, any question which asks for the probability of such a property can be answered. Clearly, only graphs can have such properties, but we will generalise this to scenarios and representatives.

Definition 6. *A scenario f for node i* has *property Y, if $G[V_i^{f=1}]$ has property Y. A representative C of node i* has *property Y, if the scenario f with $C = \mathit{blocks}_i(f)$ has property Y.*

Now, we know that representatives can have properties. If we want to use them for solving problems, it is also very important to be able to make the decision whether a (representative of a) scenario has a certain property just by considering the information given by the blocks of the representative. If this information is sufficient to determine that a representative has a property, we say that it shows this property.

Definition 7. *A representative C of node i* shows *property Y, if the information given by the blocks of C is sufficient to determine that C has Y.*

The following definition describes all the properties that can be handled or checked with our approach. Therefore, an equivalence between showing and having is necessary.

Definition 8. *A possible property Y of G_i can be* checked *by* using representatives *of i, if for all representatives C of i hold:*

$$C \text{ has property } Y \iff C \text{ shows property } Y$$

Note that from Definition 7, it already follows that the implication from right to left always holds. After having computed all representatives for node i and their associated probabilities, we can use them to easily solve several problems for G_i.

Lemma 3. *Let Y be a possible property of G_i that can be checked by the representatives of i. The probability that G_i has property Y equals the sum over the probabilities of the representatives of node i which show this property Y.*

Proof. This can easily be seen, since obviously we have to collect the scenarios with properties Y. Such scenarios are represented by representatives which also have this property Y. Hence, we have to add together the probabilities of representatives showing Y. □

When considering representatives of the root r of the treedecomposition, we can solve problems for the entire graph G. Since there are $O(2^{|V_i|})$ scenarios, this results in a method with exponential running time, because the number of representatives is a crucial point for the running time of the algorithm. Thus, in the next Section 4.2, we consider how to reduce the number of representatives. However, the more representatives per node we 'allow' the more information we conserve during the bottom up process and hence the more problems we can solved.

4.2 Reducing the Number of Representatives

In this section we will have a global look at the strategy. The correctness and restrictions follow in subsequent sections.

By using an equivalence relation between scenarios it is possible to reduce the number of representatives or in other words: it is possible to reduce the number of objects that have to be considered during the algorithms. For this, we join several scenarios to one equivalence class. We extend this formalism to representatives and say that two representatives (of equivalence classes) are equivalent if the corresponding scenarios (belonging to these classes) are equivalent. Equivalent scenarios, however, must be 'similar' with regard to the way they are processed by the algorithms. Additionally, it is reasonable to consider equivalence of scenarios of one node at a time. We have to take care that the equivalence relation R we have chosen, is suitable to solve our problem. Further, we have to show that R is 'preserved' by the algorithms, i.e. if we have two equivalent scenarios of a node as input to an algorithm, then the resulting scenarios are also equivalent. (This is described in detail in Section 4.4.) Then, we use the algorithms given in Section 3.2 with the equivalence classes (or better: with their representatives) instead of using the representatives of only one scenario. For that reason, it is only reasonable if we define classes to be equivalent, which are treated in 'the same way' by the algorithms. We have to modify the algorithms slightly, since we have classes now and we have to check after processing each node, if (the scenarios of) two classes became equivalent. If this is the case, we join them into one equivalence class with new probability the sum of the probabilities of the joined representatives (of equivalence classes).

The following list of steps summarises what we have to do.

- Step 1: We have to define an equivalence relation R which is fine enough to solve our problem. On the other hand it should be as coarse as possible to reduce the number of classes (i.e. the number of representatives) and hence the running time as much as possible. In Section 4.3, we give more details.
- Step 2: We must modify our algorithms to handle (representatives of) classes now and to maintain them, i.e. it is necessary to check for equivalent representatives/classes after processing each node. If we find such two classes among all classes of a node, we keep only one of them with accumulated probability. For doing this, we add as a last step the following code to the algorithms:

  ```
  while ∃ representatives C_a, C_b of i with a ≠ b and C_a ≡ C_b do
      Pr(C_a) := Pr(C_a) + Pr(C_b)
      delete C_b for node i
  endwhile
  ```

  Let n_c be the maximum number of equivalence classes per node, and n_b be the maximum number of blocks per representative. This code-fragment can be implemented to take $O(n_c^2)$ iterations. We assume that the check for equivalence of two representatives can be performed in $O(n_b^2)$ time. Thus, we need $O(n_c^2 \cdot n_b^2)$ steps, altogether.

– Step 3: We have to show that it is correct to use the class representatives of R instead of scenario representatives, i.e. we have to show that the algorithms preserve R. This is considered in detail in Section 4.4.
– Step 4: We analyse the running time by estimating the maximum number of equivalence classes. The maximum number of blocks is therefore very important.

A representative C is a multiset of blocks. When considering equivalence relations with an equivalence class representative C, we use $[C]$ to denote the equivalence class represented by C, i.e. $[C]$ is a set containing all scenarios represented by C.

4.3 Information Content of Equivalence Classes

In Section 4.1, we introduced some notations for scenarios. Here, we give analogous notations for equivalence classes. Therewith, we want to express the information 'contained' in equivalence classes. Furthermore, we clarify what it means for a relation to be fine enough, and we give a lemma that tells us what we can do with relations that are fine enough.

When using an equivalence relation, we define scenarios to be equivalent which may not be equal. Thus, by doing this, we lose the information that these scenarios were different and also why they were different. However, we reduce the number of class representatives, as intended. Clearly, using no equivalence relation provides as much information as possible with this method. However, for solving the S-terminal reliability problem a rather coarse equivalence relation is sufficient. Hence, we are faced with a trade-off between information content and efficiency. In analogy to the Definitions 6, 7 and 8, we give the following definition for equivalence classes.

Definition 9. *An equivalence class $[C]$ of scenarios for node i* has *property Y, if each scenario belonging to $[C]$ has property Y. An equivalence class $[C]$ of node i* shows *property Y, if its representative C shows Y.*

Definition 10. *An equivalence relation R is fine enough for property Y, if for all nodes i of the treedecomposition the following holds: For all equivalence classes $[C]$ of node i it holds:*

$$(\forall f \in [C] : f \text{ has property } Y) \quad \vee \quad (\forall f \in [C] : f \text{ does not have property } Y)$$

That means that R is fine enough for Y, iff there is no equivalence class which contains a scenario that has property Y and another scenario which does not have property Y.

In the same flavour as Lemma 3, the next one tells us that under certain conditions, we can use equivalence classes to solve problems for a graph G, after computing all equivalence classes of all nodes until we reached the root r.

Lemma 4. *Let R be an equivalence relation which is fine enough for property Y that can be checked using representatives. The probability that G_i has property Y equals the sum over the probabilities of the equivalence classes of node i which show this property Y.*

Proof. This can be proven very similar to the proof of Lemma 3. Again, we have to add together the probabilities of the scenarios with property Y. These scenarios are collected in equivalence classes which also have and show this property Y, because Y can be checked using representatives. Since R is fine enough, it is sufficient to sum up the probabilities of these equivalence classes. □

4.4 Conditions for Using Equivalence Relations

In earlier sections, we discussed when an equivalence relation is fine enough. In this section we will see under which conditions we can use (the representatives of) the equivalence classes during the execution of the bottom-up computation in the treedecomposition by means of the leaf-, introduce-, forget-, and join-node algorithms.

The choice of the equivalence relation R cannot be made arbitrarily. As mentioned in Step 1, R has to be fine enough to provide enough information to solve the problem. Furthermore, R must have some structural properties, i.e. R must be 'respected' or 'preserved' by the algorithms. This targets to the property that if two scenarios are equivalent before being processed by one of the algorithms for the four types of nodes, they are also equivalent after this processing. To capture exactly the notion of 'preserving an equivalence relation', some definitions are required.

Leaf nodes need no consideration, since we have no class representatives as input to the algorithm for leaf nodes. For forget nodes, the situation is rather simple.

Definition 11. *Let i be a forget node with child j. C_j^x is a representative of j which results, after processed by the forget-node-algorithm, in C_i^x of node i. The forget-node-algorithm* preserves *the equivalence relation R if the following holds:*

$$\forall C_j^1, C_j^2 : (C_j^1, C_j^2) \in R \Longrightarrow (C_i^1, C_i^2) \in R$$

Now, we look at introduce nodes. The introduce-node-algorithm produces two output-representatives for each input-representative, namely one for the introduced vertex v being up and one for v being down.

Definition 12. *Let i be an introduce node with child j with $X_i = X_j \cup \{v\}$. Representative C_j^x of node j is the input for the introduce-node-algorithm, which gives as output $C_i^{x,up}$ and $C_i^{x,down}$ of node i, for v is up and down, respectively. The introduce-node-algorithm* preserves *the equivalence relation R if the following holds:*

$$\forall C_j^1, C_j^2 : (C_j^1, C_j^2) \in R \Longrightarrow (C_i^{1,up}, C_i^{2,up}) \in R \wedge (C_i^{1,down}, C_i^{2,down}) \in R$$

For join nodes, we have to consider representatives of two children.

Definition 13. *Let i be a join node with children j_1 and j_2. Furthermore, let $C_{j_1}^x$ and $C_{j_2}^y$ be representatives of node j_1 and j_2, respectively. $C_i^{x,y}$ is the result of the join-node-algorithm for processing $C_{j_1}^x$ and $C_{j_2}^y$. The join-node-algorithm* preserves *the equivalence relation R if the following holds:*

$$\forall C_{j_1}^1, C_{j_1}^2, C_{j_2}^1, C_{j_2}^2 :$$
$$(C_{j_1}^1, C_{j_1}^2) \in R \wedge (C_{j_2}^1, C_{j_2}^2) \in R \wedge up(C_{j_1}^1) = up(C_{j_1}^2) = up(C_{j_2}^1) = up(C_{j_2}^2)$$
$$\Longrightarrow$$
$$(C_i^{1,1}, C_i^{1,2}) \in R \wedge (C_i^{1,2}, C_i^{2,1}) \in R \wedge (C_i^{2,1}, C_i^{2,2}) \in R$$

If we use only (the representatives of) the equivalence classes in the algorithms, we can process all scenarios represented by such equivalence classes by only one 'central execution' of an algorithm. The results will be, again, equivalence classes. Using an equivalence relation which is preserved by the algorithms is therefore important. That means that we do not have to worry whether an algorithm 'hurts' R by creating a result that actually is not an equivalence class of R.

Lemma 5. *When using equivalence classes of R as input for the forget-, introduce- and join-node-algorithm, then the result will be equivalence classes of R as well, if R is preserved by the forget-, introduce- and join-node-algorithm.*

Proof. It follows directly from the corresponding definitions above that all scenarios represented by a resulting 'equivalence class' are indeed equivalent. The algorithm-fragment given in Step 2 in Section 4.2 ensures that all equivalent scenarios are pooled together in one equivalence class. □

From the previous definition, we can see that it is important that equivalent scenarios are compatible. Actually, this is not sufficient for being able to apply our framework. The idea is to handle equivalent scenarios with only one computation. The algorithms modify only the nonempty blocks and two equivalent scenarios have to be modified in the same way. Hence, for applying the framework equivalent scenarios must have the same set of nonempty blocks. However, there are reasonable exceptions to this, which will not be considered here in general, to keep the framework and its presentation less technical. The next definition summarises the two conditions discussed above.

Definition 14. *An equivalence relation R is called* proper *if*

- *for all equivalence classes C of R it holds that: for all scenarios $f, f' \in C$, the sets of nonempty blocks of f and f', respectively, are equal and*
- *R is preserved by the forget-, introduce- and join-node-algorithm.*

4.5 The Correctness of the Algorithms for Classes

In this section, we look at the correctness of the counterparts of the algorithms given in Section 3.2. As described earlier, only little modifications are needed to get algorithms for (representatives of) equivalence classes rather than (representatives of) scenarios. We can take the algorithms described in Section 3.2 and extend them by the while loop given in Step 2 in Section 4.2. While the algorithms are straightforward, the proof of their correctness is slightly more complicated.

We use the same terminology as in Section 3.2: i denotes a node of the tree T. Node i may have $0, 1$ or 2 children, depending on the type of i. These children are referred to as j or j_1, j_2. C_i denotes a representative of a class of i, which is computed using the representative(s) of class(es) C_j (C_{j_1}, C_{j_2}) of node(s) j (j_1, j_2), respectively. We will have a look at each type for node i.

When handling representatives, a lower index refers to the node the representative belongs to, and an upper index is used to differentiate different representatives belonging to the node considered. In general, the algorithms given in Section 3.2 can be used here as well. However, they have to be extended by the code-fragment of Step 2 in Section 4.2 to check for new equivalences. It is easy to see, that each of these 'new' algorithms can be performed in time $O(n_c^2 \cdot n_b^2 \cdot k^2)$.

Leaf Nodes. As seen before, for leaves i of T, the algorithm is rather simple, since $|X_i| = 1$. It is very similar to the description for leaves given in Section 3.2, because we have two scenarios for a leaf. For proper relations, these scenarios will never be in one equivalence class at a leaf, since their sets of nonempty blocks are not equal. Hence, we have two equivalence classes for a leaf. The remaining argumentation is very similar to the part in Section 3.2 concerning leaf nodes.

Introduce Nodes. Suppose i is an introduce node with child j, such that $X_i = X_j \cup \{v\}$. We use the representatives of the classes and their probabilities of node j to compute the representatives and their probabilities of node i. For any state of v (up or down, in L or not in L, in S or not in S) we compute the representatives and add them to the set of representatives of node i. This is done in an analogous way as described in Section 3.2.

Lemma 6. *Given the representatives of all equivalence classes of a proper equivalence relation and their probabilities of the child j of an introduce node i, the introduce-node-algorithm computes (the representatives of) all equivalence classes and their probabilities for node i correctly.*

Proof. We can compute the representatives and their probabilities for node i by considering every possible scenario for node i and creating equivalence classes according to the equivalence relation.

However, recall that v is the only vertex in $V_i \setminus V_j$. So, each scenario for node i can be obtained by taking a scenario for node j and extending it by specifying whether v is up or down. All possible scenarios for node j are represented by the representatives of equivalence classes of j, whose probabilities are known. Since the considered equivalence relation is proper, two scenarios which are equivalent at node j will result in equivalent scenarios of node i (for v being fixed to up or down). For that reason, it is sufficient to combine the representatives of classes of j with v to create the representatives of classes of i. To do this we consider both v being up and down for each class representative C_j of j and make the appropriate modifications. By doing this, we implicitly look at every scenario which is possible for G_i. For the correctness, we consider for a class representative C_j of j, two cases: v is up, or v is down:

Case I: v is up. For all $f \in [C_j]$, we extend f by $f(v) = 1$, i.e. for all $w \in V_j : f(w)$ is unchanged. The argumentation is very similar to the proof of Lemma 1. We have to use only one more argument. Since the relation is proper, all scenarios $f \in [C_j]$ have the same set of nonempty blocks. And hence, all these scenarios would be handled in the same way when treated individually. That is why we can handle them by a single pass of the outer loop of the algorithm when using the representative C_j. The probability of the new class with representative C_i is:

$$\Pr(C_i) = \sum_{f \in [C_i]} \Pr(f) = \sum_{f \in [C_j]} \Pr(f) \cdot p(v) = \Pr(C_j) \cdot p(v)$$

Here, $[C_i]$ contains scenarios, which are scenarios belonging to $[C_j]$ extended by $f(v) = 1$.

Case II: v is down. Similar to Case I, we extend all scenarios f by 'v is down'. Hence, no new connections between vertices of blocks or components can be made. Thus, we do not modify C_j to get the new class representative C_i. The probability of C_i is:

$$\Pr(C_i) = \Pr(C_j) \cdot (1 - p(v))$$

□

Forget Nodes. For a forget node i with child j, let $X_i = X_j \setminus \{v\}$. Note that $V_i = V_j$, thus i and j have the same scenarios. However, they might have different representatives. Let C_j and C_i be two representatives of the same scenario of node j and i, respectively. Because v does not appear in the blocks of C_i, C_j and C_i are different or equal, depending on the state of vertex v (up or down). Each scenario belongs to an equivalence class. Thus, we modify every class representative C_j of j in the following way to create a class representative C_i of node i. We simply delete v from every block of C_j containing v to obtain C_i. For a proper equivalence relation, all scenarios belonging to an equivalence class $[C_j]$ have the same set of nonempty blocks. Hence, by deleting v from the blocks of C_j, we implicitly delete v from all scenarios represented by C_j. This results in the new equivalence class C_i for node i. Because C_i and C_j represent the same scenarios, we have $\Pr(C_i) = \Pr(C_j)$.

Join Nodes. For join nodes i with children j_1 and j_2 we have: $X_i = X_{j_1} = X_{j_2}$. Note that $V_i = V_{j_1} \cup V_{j_2}$, and that $V_{j_1} \cap V_{j_2} = X_i$.

We proceed as described in Section 3.2. We combine compatible class representatives of nodes j_1 and j_2. After combining, we repair them with the procedure given in Section 4.2. By doing this, we get equivalence classes of node i.

Lemma 7. *Given all the representatives of equivalence classes of the children j_1 and j_2 of a join node i and the probabilities of these equivalence classes with a proper underlying equivalence relation, the join-node-algorithm computes the representatives and the probabilities of classes of node i correctly.*

Proof. We consider an equivalence class representative C_i of node i. By restricting a scenario $f \in [C_i]$ to V_{j_1} and V_{j_2}, respectively, it determines two scenarios f_1 of G_{j_1} and f_2 of G_{j_2}. Those two scenarios f_1 and f_2 belong to equivalence classes C_{j_1} and C_{j_2} of nodes j_1 and j_2, respectively. The representatives of these classes and their probabilities are already computed in our bottom up approach. Note that these representatives are compatible. Furthermore, we have due to equivalence (and hence compatibility) that any scenario in $[C_{j_1}]$ combined with any scenario in $[C_{j_2}]$ results in a scenario in $[C_i]$. This yields the Cartesian product of $[C_{j_1}]$ and $[C_{j_2}]$ with $|[C_{j_1}]| \cdot |[C_{j_2}]|$ combinations. Fortunately, since any such combination results in a scenario in $[C_i]$, we can compute all combinations by a single pass of the algorithm by using the representatives of $[C_{j_1}]$ and $[C_{j_2}]$. Hence, simply combining the representatives C_{j_1} and C_{j_2} is sufficient to handle all scenarios represented by them and to create (a subset of) $[C_i]$.

To create the entire set of scenarios $[C_i]$, we have to process all equivalence classes emerging from restrictions of scenarios of $[C_i]$ to V_{j_1} and V_{j_2}. However, the same argumentation as above holds for these as well, and the added code-fragment of Step 2 in Section 4.2 will check for equivalent classes and produce the final class $[C_i]$.

Let C_{j_1} and C_{j_2} be two compatible equivalence class representatives of node j_1 and j_2, respectively. $[C_{j_1}]$ and $[C_{j_2}]$ are combined to get class $[C_i]$ of node i. Then we have for the probability of C_i of node i:

$$\Pr(C_i) = \sum_{f \in [C_i]} \Pr_{V_i}(f)$$

This product can be split into factors with regard to the children of node i. However, the set X_i is contained in both sets V_{j_1} and V_{j_2}. Hence, this product would contain a factor for the state probabilities of vertices of X_i twice, which we correct by an appropriate division:

$$\Pr(C_i) = \sum_{f \in [C_i]} \Pr_{V_{j_1}}(f) \cdot \Pr_{V_{j_2}}(f) \cdot \frac{1}{\Pr_{X_i}(f)}$$

Since the latter factor is constant for all $f \in [C_i]$ for a proper equivalence relation, we have:

$$\Pr(C_i) = \frac{1}{\Pr_{X_i}(f)} \cdot \sum_{f \in [C_i]} \Pr_{V_{j_1}}(f) \cdot \Pr_{V_{j_2}}(f)$$

Because the equivalence class $[C_i]$ contains all scenarios, which result from the Cartesian product of compatible classes $[C_{j_1}]$ and $[C_{j_2}]$ of j_1 and j_2, respectively, we have:

$$\sum_{f \in [C_i]} \Pr_{V_{j_1}}(f) \cdot \Pr_{V_{j_2}}(f) = \sum_{f \in [C_i]} \Pr_{V_{j_1}}(f) \cdot \sum_{f \in [C_i]} \Pr_{V_{j_2}}(f) = \sum_{f \in [C_{j_1}]} \Pr_{V_{j_1}}(f) \cdot \sum_{f \in [C_{j_2}]} \Pr_{V_{j_2}}(f)$$

And hence, this results in:

$$\begin{aligned}
\Pr(C_i) &= \frac{1}{\Pr_{X_i}(f)} \cdot \sum_{f \in [C_{j_1}]} \Pr_{V_{j_1}}(f) \cdot \sum_{f \in [C_{j_2}]} \Pr_{V_{j_2}}(f) \\
&= \frac{1}{\Pr_{X_i}(f)} \cdot \Pr(C_{j_1}) \cdot \Pr(C_{j_2})
\end{aligned}$$

After computing the new classes of node i, which contain scenarios corresponding to the Cartesian product of scenarios of compatible classes of j_1 and j_2, we have to check for new equivalences. The correctness of the procedure we use for this, is easy to see. □

We are now ready to bring everything together.

Theorem 1. *Let R be a proper equivalence relation of scenarios that is fine enough for property Y which can be checked using representatives. Then we can use our framework to compute the probability that G_i has property Y.*

Proof. The correctness of the computation of the representatives and their probabilities follows from the correctness of the particular algorithms for the four types of nodes. Lemma 6 and 7 show this for introduce- and join-nodes. The situation is trivial for leaf- and forget-nodes. The theorem follows now from Lemma 4 and 5. □

The running time depends on the relation R. For a fixed size of X_i, if R has a finite number of equivalence classes, then the algorithm is linear time. If the number of equivalence classes is bounded by a polynomial in the number of vertices of G, then the algorithm uses polynomial time. For further discussion, see Section 7.3.

5 Which Problems fit into this framework

In this section, we look at examples of solvable problems which yield different running times. Later in this section, we look at relations (and the resulting running times) that meet all conditions described in the previous section, because they are generated by specific operations or because they are a combination of relations.

5.1 Solvable Problems

It is not possible to give an exhaustive list of problems that can be handled with this approach, because many equivalence relations can be used, and very often, many different questions can be handled with a single relation. Note that we assume in this entire section that a treedecomposition of bounded width of the graph is given. Fortunately, Theorem 1 gives information about solvable problems. We simply can say that every problem that asks for the probability of a property Y of G can be solved, if Y can be checked using classes. Therefore we should find an equivalence relation R as coarse as possible, but still fine enough for Y. Furthermore, to apply the framework described in this paper, R must be preserved by the algorithms. Below, we list some example properties.

To obtain relations that allow algorithms with linear running time, we can restrict the maximum number of blocks and the number of flags per block to be constant. With such relations we can answer questions like: 'What is the probability that all clients are connected to at least one server?' or 'What is the probability that all servers are useful, i.e. have a client connected to them?' We can also use only one kind of special vertices, e.g. only servers. With such a relation we are able to give an answer to 'What is the probability that all servers are connected?', which is the classical L-terminal reliability problem. With additional ideas and modifications, it is also possible to answer the following question in linear time: 'What is the expected number of components that contain at least one vertex of S (of L; of S and L)?'

A possible assumption would be to consider the number of servers n_S to be small. We can use a different S-flag for each server. In this case, a relation which does not conflate empty blocks with S-flags, but only with L-flags, can be utilised. Further, each block can have multiple flags of each type. Unfortunately, this relation leads to a maximum number of classes bounded exponentially in n_S. With this relation we can answer the question with which probability certain servers are connected, and with what probability server x has at least y clients connected to it (x, y are integers). We can also determine the 'most useless' server, i.e. the server with smallest expected number of clients connected to it. Of course, this relation enables us to compute the expected number of components with at least one server.

Polynomial running time in n_L and n_S is needed by relations which bound n_b by a constant, however the number of flags per block is only bounded by n_L and n_S. Such relations enable properties like: at least x clients are not connected to a server, or at most y server are not connected to a client, as well as at least x clients are connected to at least one server while at least y servers are connected to at least one client.

5.2 Generating Relations

The relation which is the finest we can have, is the relation $\hat{R}$, which assigns to each scenario its own equivalence class. This relation is clearly preserved by the algorithms. We show below, that if we modify $\hat{R}$ by applying a sequence of specific operations to get a relation R, then R is preserved by the algorithms as well. The goal is to 'construct' relations which are 'automatically' preserved.

As a reminder, a representative is a multiset of blocks, not necessarily representing a scenario. With an operation $h : \mathcal{C} \to \mathcal{C}$ (where $\mathcal{C}$ is the set of multisets of blocks), which takes the representation, i.e. the set of blocks of an equivalence class as input, we can define a new equivalence relation R_h. We give a list of a few basic operations, with which many useful relations can be created. Therefore, we can start with the finest relation which equivalence classes represent exactly one scenario. The general form of an operation is: $h_{action}^{block-sel}$, whereby *block-sel* selects the blocks for the *action*, which is one of the following:

- '*del*': We *del*ete all blocks that are selected by *block-sel*.
- '*dmulS*': Here, we *d*rop *mul*tiple S-flags and keep only one. That means each block selected by *block-sel* is replaced by the same block at which we delete multiple S-flags and we keep only one of them.
- '*dmulL*': The same as $dmulS$, but here we *d*rop *mul*tiple L-flags.
- '*con*': We *con*flate all selected blocks and *con*catenate their lists of flags. That means all blocks selected by *block-sel* are united to one block with as many S- and L- flags as those of the united blocks together.

As already mentioned, blocks are selected by *block-sel*, which is a sequence of the following symbols:

- '$=\emptyset$' or '$\neq\emptyset$', which determines that either empty blocks or nonempty blocks are selected, respectively.
- 'S' or '$\not S$', which determines that either blocks that must have at least one S-flag or blocks that must not have any S-flag are selected.
- 'L' or '$\not L$', which is the same as above for L-flags.

We give the set of those block-selectors that are considered in this paper.

$$BSel = \{\neq\emptyset, \neq\emptyset\not S\not L, \neq\emptyset S\not L, \neq\emptyset\not S L, \neq\emptyset SL, =\emptyset, =\emptyset\not S\not L, =\emptyset S\not L, =\emptyset\not S L, =\emptyset SL\}$$

For the sake of clarity, we will look at some examples:

- $h_{del}^{=\emptyset\not S\not L}(C)$ = the multiset of blocks consisting of exactly all blocks of C but without empty blocks without any flag:

$$\begin{array}{cl} h_{del}^{=\emptyset\not S\not L} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S} \qquad \} \end{array}$$

- $h_{dmulS}^{=\emptyset SL}(C)$ = the multiset of blocks consisting of exactly all blocks of C but for each empty block with at least one S-flag and at least one L-flag, we drop (or delete) all but one S-flag:

$$\begin{array}{cl} h_{dmulS}^{=\emptyset SL} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{LLL}, \ \{\ \}_{LL}^{S}, \ \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ \} \end{array}$$

- $h_{dmulL}^{\neq\emptyset\not S L}(C)$ = the multiset of blocks consisting of exactly all blocks of C but each nonempty block $\{...\}_{L...L}$ with at least one L-flag and no S-flag is replaced by the same block $\{...\}_L$ with only one L-flag:

$$\begin{array}{cl} h_{dmulL}^{\neq\emptyset\not S L} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \ \{v,w\}_{L}, \ \{\ \}_{LL}^{SSS}, \{\ \}^{SS}, \{\ \}^{S}, \{\ \}\ \} \end{array}$$

– $h_{con}^{=\emptyset S \not L}(C)$ = the multiset of blocks consisting of exactly all blocks of C but all empty blocks with at least one S-flag and no L-flag are replaced by one empty block with as many S-flags as those of the replaced blocks together:

$$\begin{array}{rl} h_{con}^{=\emptyset S \not L} & (\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS},\ \{\ \}^{SS},\ \{\ \}^{S}, \{\ \}\ ) \\ = & \{\ \{u\}, \{v,w\}_{LLL}, \{\ \}_{LL}^{SSS}, \{\ \}^{SSS}, \qquad \{\ \}\ \} \end{array}$$

Compiling all operations which we will consider, we get the set $\mathcal{H}$:

$$\mathcal{H} = \{h_{del}^{=\emptyset \not S \not L}, h_{del}^{=\emptyset S \not L}, h_{del}^{=\emptyset \not S L}, h_{del}^{=\emptyset S L}, h_{dmulS}^{=\emptyset S \not L}, h_{dmulS}^{=\emptyset S L}, h_{dmulS}^{\neq\emptyset S \not L}, h_{dmulS}^{\neq\emptyset S L},$$

$$h_{dmulL}^{=\emptyset \not S L}, h_{dmulL}^{=\emptyset S L}, h_{dmulL}^{\neq\emptyset \not S L}, h_{dmulL}^{\neq\emptyset S L}, h_{con}^{=\emptyset S \not L}, h_{con}^{=\emptyset \not S L}, h_{con}^{=\emptyset S L}\}$$

Note that in this list the only operations mapping nonempty sets are operations that drop multiple flags, i.e. that do not change any vertex of the block.

So far, h is an operation which takes a multiset of blocks as input and gives a multiset of blocks as output. In the following definition, we extend the applicability of h to equivalence relations R. Note, that the relation $\hat{R}$ is the finest relation possible. It assigns to each scenario its own equivalence class.

Definition 15. *Let $\mathcal{R}_0 = \{\hat{R}\}$ and $h \in \mathcal{H}$. For $R \in \mathcal{R}_i$ we define a new equivalence relation $h(R)$ in the following way: Let C^1 and C^2 be two representatives of two equivalence classes $[C^1]$ and $[C^2]$ of R, respectively. Then*

$$(C^1, C^2) \in h(R) \Longleftrightarrow h(C^1) = h(C^2)$$

Furthermore, we have:

$$\mathcal{R}_{i+1} = \{R' | R' = h(R) \text{ for } R \in \mathcal{R}_i \text{ and } h \in \mathcal{H}\}$$

and

$$\mathcal{R} = \bigcup_{i=0}^{\infty} \mathcal{R}_i$$

If $(C^1, C^2) \in h(R)$, it might be convenient to choose the representative C of the new equivalence class $[C]$ with $C^1, C^2 \in [C]$ as follows: $C = h(C^1) = h(C^2)$. However, the algorithms have to be modified to maintain this. As an example, if the forget-node-algorithm creates a new empty block without any flags and if we use the operation $h_{del}^{=\emptyset \not S \not L}$ to define the new relation, then we have to modify the forget-node-algorithm to delete the new empty block.

Thus, applying an operation to the multiset of blocks of the representation of an equivalence class immediately results in the multiset of blocks of an equivalence class of the new relation. Now, the idea is to consider operations $h \in \mathcal{H}$, and prove that $h(R)$ is preserved by the algorithms if $h(R)$ is obtained by applying a sequence of operations of $\mathcal{H}$ to $\hat{R}$. We require some intermediate lemmas.

In the upcoming proofs, we use the following terms: For A a multiset of blocks, $blocks^{\alpha}(A)$ is the multiset of all blocks of A selected by $\alpha \in BSel$. We call these sets *categories*. Here are a few examples:

– $blocks^{\neq\emptyset}(A)$ is the set of all nonempty blocks of A.

- $blocks^{\neq\emptyset\not{S}L}(A)$ is the set of all nonempty blocks of A with at least one L-flag and no S-flags.
- $blocks^{=\emptyset\not{S}\not{L}}(A)$ is the multiset of all empty blocks of A with no flags.

Lemma 8. *Let h be a sequence of operations of $\mathcal{H}$, A be a multiset of blocks, and $\alpha \in BSel$.*

$$blocks^{\alpha}(h(A)) = h(blocks^{\alpha}(A))$$

Proof. This is easy to see for a single operation $h \in \mathcal{H}$. By applying this repetitively for each operation in a sequence h, we obtain the statement of the lemma. $\square$

As a reminder, the symbol '$\cup$' between multisets denotes the multiunion. (See the paragraph about blocks in Section 2.2 for more details.)

Lemma 9. *Let A be a multiset of blocks and let h be a sequence of operations of $\mathcal{H}$.*

$$\begin{aligned} h(A) = {} & h(blocks^{\neq\emptyset\not{S}\not{L}}(A)) \cup h(blocks^{=\emptyset\not{S}\not{L}}(A)) \\ & \cup h(blocks^{\neq\emptyset S\not{L}}(A)) \cup h(blocks^{=\emptyset S\not{L}}(A)) \\ & \cup h(blocks^{\neq\emptyset\not{S}L}(A)) \cup h(blocks^{=\emptyset\not{S}L}(A)) \\ & \cup h(blocks^{\neq\emptyset SL}(A)) \cup h(blocks^{=\emptyset SL}(A)) \end{aligned}$$

Proof. The multiset A of blocks can be partitioned into:

$$\begin{aligned} A = {} & blocks^{\neq\emptyset\not{S}\not{L}}(A) \cup blocks^{=\emptyset\not{S}\not{L}}(A) \\ & \cup blocks^{\neq\emptyset S\not{L}}(A) \cup blocks^{=\emptyset S\not{L}}(A) \\ & \cup blocks^{\neq\emptyset\not{S}L}(A) \cup blocks^{=\emptyset\not{S}L}(A) \\ & \cup blocks^{\neq\emptyset SL}(A) \cup blocks^{=\emptyset SL}(A) \end{aligned}$$

It follows from the definition of the categories $blocks^{\alpha}(A)$ above (for $\alpha \in BSel \setminus \{\neq\emptyset, =\emptyset\}$) that this is indeed a partition. Each single operation handles only the blocks of one category and leaves the other categories untouched. Therefore, it is easy to see that the claim is true for h being a single operation. Now, the result follows from Lemma 8. $\square$

Lemma 10. *Let A and B be multisets of blocks and let h be a sequence of operations of $\mathcal{H}$.*

$$h(A) = h(B) \iff$$

$$\forall\alpha \in BSel : h(blocks^{\alpha}(A)) = h(blocks^{\alpha}(B))$$

Proof. The '$\Leftarrow$' direction follows from Lemma 9. To see the '$\Rightarrow$' direction, let $D \in h(A)$. Clearly, $D \in h(B)$ and there is an $\alpha \in BSel \setminus \{\neq\emptyset, =\emptyset\}$, with $D \in blocks^{\alpha}(h(A))$. Since $h(A) = h(B)$, we have $D \in blocks^{\alpha}(h(B))$. Because this is true for all $D \in blocks^{\alpha}(h(A))$, we have: $blocks^{\alpha}(h(A)) \subseteq blocks^{\alpha}(h(B))$, and by symmetry: $blocks^{\alpha}(h(A)) = blocks^{\alpha}(h(B))$. By Lemma 8, we conclude: $h(blocks^{\alpha}(A)) = h(blocks^{\alpha}(B))$. Since D was chosen arbitrarily, α was chosen arbitrarily as well, and we see that the last equation is true for all $\alpha \in BSel$. $\square$

Lemma 11. *Let A, B be multisets of blocks and $h \in \mathcal{H} \setminus \{h^{=\emptyset S\not{L}}_{con}, h^{=\emptyset\not{S}L}_{con}, h^{=\emptyset SL}_{con}\}$ Then it holds that:*

$$h(A \cup B) = h(A) \cup h(B)$$

Proof. This is easy to see, because such an h modifies a multiset of blocks locally, namely block by block, which means that each block is exchanged by another one of the same category or deleted. $\square$

It is not possible to give an easy to prove statement as in Lemma 11 if $h_{con}^{=\emptyset S\not L}$, $h_{con}^{=\emptyset \not S L}$ or $h_{con}^{=\emptyset SL}$ is involved. In this case, it is possible that several blocks are exchanged by one other block. We will have a closer look at a sequence containing $h_{con}^{=\emptyset S\not L}$. Operation $h_{con}^{=\emptyset S\not L}$ only affects empty blocks with at least one S-flag and no L-flags. As we have seen above, there is no interference between different categories. Thus, in the next lemma, we restrict ourself to this category and only to operations which have an effect to this category.

Lemma 12. *Let A, B, C, D be multisets of empty blocks with at least one S-flag and no L-flags. For h a sequence of operations of $\{h_{del}^{=\emptyset S\not L}, h_{dmulS}^{=\emptyset S\not L}, h_{con}^{=\emptyset S\not L}\}$, we have:*

$$h(A) = h(B) \ \wedge \ h(C) = h(D) \implies h(A \cup C) = h(B \cup D)$$

Proof. We consider a number of cases:

- Case 1: If $h_{con}^{=\emptyset S\not L}$ is not in the sequence h, the result follows directly from Lemma 11.
- Case 2: If there is any $h_{del}^{=\emptyset S\not L}$ in the sequence h, we have the situation, that there are no empty blocks with at least one S-flag and no L-flags anymore, and the result trivially holds.

  Hence, we suppose in the following cases that $h_{con}^{=\emptyset S\not L} \in h$ and $h_{del}^{=\emptyset S\not L} \notin h$.

- Case 3: $h(X) = ...h_{dmulS}^{=\emptyset S\not L}(...h_{con}^{=\emptyset S\not L}(...(X)...)...)...$; i.e. first we apply $h_{con}^{=\emptyset S\not L}$ and then $h_{dmulS}^{=\emptyset S\not L}$. Sequence h will turn all empty blocks with at least one S-flag and no L-flags into a single empty block with a single S-flag. Then we know that $h(A) = h(B) \in \{ \ \{ \ \}^S, \emptyset \ \}$ and $h(C) = h(D) \in \{ \ \{ \ \}^S, \emptyset \ \}$. (We have $h(A) = h(B) = \emptyset$ iff A and B do not contain any empty block with at least one S-flag and no L-flags. The same also holds for sets C and D.) If $A \cup C$ contains an empty block with only S-flags, then $B \cup D$ as well and we have: $h(A \cup C) = h(B \cup D) = \{ \ \}^S$ otherwise we have: $h(A \cup C) = h(B \cup D) = \emptyset$.
- Case 4: $h(X) = ...h_{con}^{=\emptyset S\not L}(...h_{dmulS}^{=\emptyset S\not L}(...(X)...)...)...$; i.e. first we apply $h_{dmulS}^{=\emptyset S\not L}$ and then $h_{con}^{=\emptyset S\not L}$. The result of $h(X)$ in this case is $\{ \ \}^{S...S}$, whereat the number of S-flags equals the number of empty blocks in X with at least one S-flag and no L-flags. Let a be the number of empty blocks of A with at least one S-flag and no L-flags. We define b, c, d in the same way. Then we trivially have: $a = b \wedge c = d \Longrightarrow a + c = b + d$, which implies $h(A \cup C) = h(B \cup D)$.
- Case 5: only $h_{con}^{=\emptyset S\not L}$ and no $h_{dmulS}^{=\emptyset S\not L}$ is applied. Let a be the total number of all S-flags of all empty blocks of A with at least one S-flag and no L-flags. We define b, c, d in the same way. Since $h(A)$ is exactly one empty block with a S-flags and no L-flags, we have: $a = b \wedge c = d \Longrightarrow a + c = b + d$. Hence $h(A \cup C) = h(B \cup D)$.
- Case 6: $h(X) = ...h_{con}^{=\emptyset S\not L}(...h_{dmulS}^{=\emptyset S\not L}(...h_{con}^{=\emptyset S\not L}(...(X)...)...)...)...$; i.e. first we apply $h_{con}^{=\emptyset S\not L}$, then $h_{dmulS}^{=\emptyset S\not L}$ and then $h_{con}^{=\emptyset S\not L}$ again. Here the outermost $h_{con}^{=\emptyset S\not L}$ has no effect and hence we have already considered this case above.
- Case 7: $h(X) = ...h_{dmulS}^{=\emptyset S\not L}(...h_{con}^{=\emptyset S\not L}(...h_{dmulS}^{=\emptyset S\not L}(...(X)...)...)...)...$; i.e. first we apply $h_{dmulS}^{=\emptyset S\not L}$, then $h_{con}^{=\emptyset S\not L}$ and then $h_{dmulS}^{=\emptyset S\not L}$ again. In this case the innermost $h_{dmulS}^{=\emptyset S\not L}$ has no effect and thus already handled above.

$\square$

We can easily see that lemmas similar to the previous one are true, for considering blocks with at least one L-flag and no S-flags (or for blocks with at least one S-flag and at least one L-flag). The proofs are very similar and hence we omit them. We generalize the previous lemma which is the last intermediate step.

Lemma 13. *Let A, B, C, D be multisets of blocks. For h a sequence of operations of $\mathcal{H}$, we have:*

$$h(A) = h(B) \ \wedge \ h(C) = h(D) \implies h(A \cup C) = h(B \cup D)$$

Proof. As discussed above, operations for different categories of blocks do not affect each other, and hence we can consider each category separately. Thus, the result follows easily from Lemma 9, 10, 12, and the variants of Lemma 12, discussed above. □

Lemma 14. *Let R be an equivalence relation with $R \in \mathcal{R}$. R is preserved by the introduce-, forget-, and join-node-algorithm.*

Proof. Let h be the sequence of operations with $h(\hat{R}) = R$. In many equations below, we make use of Lemmas 12 and 13. We consider each algorithm separately:

- The forget-node-algorithm:
  Node i is a forget node with child j. Let C_j^1, C_j^2 be representatives of node j and let C_i^1, C_i^2 be representatives of node i which are the results of the forget-node-algorithm of representatives C_j^1, C_j^2, respectively. We have:

$$\begin{aligned} h(C_j^1) = h(C_j^2) \implies & \mathit{blocks}^{\neq\emptyset}(C_j^1) = \mathit{blocks}^{\neq\emptyset}(C_j^2) \ \wedge \\ & \mathit{blocks}^{\neq\emptyset}(C_i^1) = \mathit{blocks}^{\neq\emptyset}(C_i^2) \ \wedge \\ & h(\mathit{blocks}^{=\emptyset}(C_j^1)) = h(\mathit{blocks}^{=\emptyset}(C_j^2)) \end{aligned} \tag{1}$$

  Now, we can distinguish two cases:
  Case 1: No new empty block arises. Then we have:

$$\mathit{blocks}^{=\emptyset}(C_j^1) = \mathit{blocks}^{=\emptyset}(C_i^1) \ \wedge \ \mathit{blocks}^{=\emptyset}(C_j^2) = \mathit{blocks}^{=\emptyset}(C_i^2)$$

  Together with equation 1, we see:

$$\begin{aligned} h(C_j^1) = h(C_j^2) & \implies h(\mathit{blocks}^{=\emptyset}(C_j^1)) = h(\mathit{blocks}^{=\emptyset}(C_j^2)) \\ & \implies h(\mathit{blocks}^{=\emptyset}(C_i^1)) = h(\mathit{blocks}^{=\emptyset}(C_i^2)) \\ & \implies h(C_i^1) = h(C_i^2) \end{aligned}$$

  Case 2: A new empty block B^* arises. Here, we have:

$$\mathit{blocks}^{=\emptyset}(C_i^1) = \mathit{blocks}^{=\emptyset}(C_j^1) \cup B^* \ \wedge \ \mathit{blocks}^{=\emptyset}(C_i^2) = \mathit{blocks}^{=\emptyset}(C_j^2) \cup B^*$$

  Applying Lemma 13 with $A = \mathit{blocks}^{=\emptyset}(C_j^1), B = \mathit{blocks}^{=\emptyset}(C_j^2)$ and $C = D = B^*$, we have:

$$\begin{aligned} & h(\mathit{blocks}^{=\emptyset}(C_j^1) \cup B^*) = h(\mathit{blocks}^{=\emptyset}(C_j^2) \cup B^*) \\ \implies & h(\mathit{blocks}^{=\emptyset}(C_i^1)) = h(\mathit{blocks}^{=\emptyset}(C_i^1)) \\ \implies & h(C_i^1) = h(C_i^2). \end{aligned}$$

  In both cases, we have $h(C_i^1) = h(C_i^2)$ and hence R is preserved by the forget-node-algorithm.

– The introduce-node-algorithm:
Node i is an introduce node with child j. Let C_j^1, C_j^2 be representatives of node j. Also, let $C_i^{1,up}, C_i^{2,up}, C_i^{1,down}, C_i^{2,down}$ be representatives of node i which are the results of the introduce-node-algorithm of representatives C_j^1, C_j^2, with respect to newly introduced vertex being up or down.

$$\begin{aligned} h(C_j^1) = h(C_j^2) &\Longrightarrow blocks^{\neq\emptyset}(C_j^1) = blocks^{\neq\emptyset}(C_j^2) \\ &\Longrightarrow blocks^{\neq\emptyset}(C_i^{1,up}) = blocks^{\neq\emptyset}(C_i^{2,up}) \ \wedge \\ &\qquad blocks^{\neq\emptyset}(C_i^{1,down}) = blocks^{\neq\emptyset}(C_i^{2,down}) \end{aligned} \tag{2}$$

Furthermore, the algorithm does not modify the empty blocks of the classes:

$$blocks^{=\emptyset}(C_j^1) = blocks^{=\emptyset}(C_i^{1,up}) = blocks^{=\emptyset}(C_i^{1,down})$$

$$blocks^{=\emptyset}(C_j^2) = blocks^{=\emptyset}(C_i^{2,up}) = blocks^{=\emptyset}(C_i^{2,down})$$

Thus, we have:

$$\begin{aligned} h(C_j^1) = h(C_j^2) &\Longrightarrow h(blocks^{=\emptyset}(C_j^1)) = h(blocks^{=\emptyset}(C_j^2)) \\ &\Longrightarrow h(blocks^{=\emptyset}(C_i^{1,up})) = h(blocks^{=\emptyset}(C_i^{2,up})) \end{aligned}$$

Using this and equation 2, we see:

$$\begin{aligned} h(C_i^{1,up}) &= h(blocks^{=\emptyset}(C_i^{1,up}) \cup blocks^{\neq\emptyset}(C_i^{1,up})) \\ &= h(blocks^{=\emptyset}(C_i^{1,up})) \cup blocks^{\neq\emptyset}(C_i^{1,up}) \\ &= h(blocks^{=\emptyset}(C_i^{2,up})) \cup blocks^{\neq\emptyset}(C_i^{2,up}) \\ &= h(C_i^{2,up}) \end{aligned}$$

The statement $h(C_i^{1,down}) = h(C_i^{2,down})$ follows directly from:

$$blocks^{\neq\emptyset}(C_j^1) = blocks^{\neq\emptyset}(C_i^{1,down}) \ \wedge \ blocks^{\neq\emptyset}(C_j^2) = blocks^{\neq\emptyset}(C_i^{2,down})$$

We conclude that R is preserved by the introduce-node-algorithm.

– The join-node-algorithm:
Node i is a join node with children j_1, j_2. Let $C_{j_1}^1$, $C_{j_1}^2$, $C_{j_2}^3$, $C_{j_2}^4$ be compatible classes of nodes j_1 and j_2, respectively. Let C_i^{13} be the result of the join-node-algorithm when processing $C_{j_1}^1$ and $C_{j_2}^3$. C_i^{14}, C_i^{23} and C_i^{24} are defined in the same way. We have:

$$\begin{aligned} h(C_{j_1}^1) = h(C_{j_1}^2) &\Longrightarrow blocks^{\neq\emptyset}(C_{j_1}^1) = blocks^{\neq\emptyset}(C_{j_1}^2) \ \wedge \\ &\qquad h(blocks^{=\emptyset}(C_{j_1}^1)) = h(blocks^{=\emptyset}(C_{j_1}^2)) \end{aligned}$$

An analogous implication can be obtained with $h(C_{j_2}^3) = h(C_{j_2}^4)$. Let C be a multiset of blocks, then $REPAIR(C)$ is the result of the REPAIR-algorithm of Section 3 applied to C. Then we can see:

$$\begin{aligned} blocks^{\neq\emptyset}(C_i^{13}) &= REPAIR(blocks^{\neq\emptyset}(C_{j_1}^1) \cup blocks^{\neq\emptyset}(C_{j_2}^3)) \\ &= REPAIR(blocks^{\neq\emptyset}(C_{j_1}^2) \cup blocks^{\neq\emptyset}(C_{j_2}^4)) \\ &= blocks^{\neq\emptyset}(C_i^{24}) \end{aligned} \tag{3}$$

Applying Lemma 13 with $A = blocks^{=\emptyset}(C_{j_1}^1)$, $B = blocks^{=\emptyset}(C_{j_1}^2)$, $C = blocks^{=\emptyset}(C_{j_2}^3)$ and $D = blocks^{=\emptyset}(C_{j_2}^4)$, we get:

$$\begin{aligned} h(blocks^{=\emptyset}(C_i^{13})) &= h(blocks^{=\emptyset}(C_{j_1}^1) \cup blocks^{=\emptyset}(C_{j_2}^3)) \\ &= h(blocks^{=\emptyset}(C_{j_1}^2) \cup blocks^{=\emptyset}(C_{j_2}^4)) \\ &= h(blocks^{=\emptyset}(C_i^{24})) \end{aligned} \tag{4}$$

From equations 3 and 4 it follows that $h(C_i^{13}) = h(C_i^{24})$. We can see $h(C_i^{13}) = h(C_i^{24}) = h(C_i^{23}) = h(C_i^{14})$ by using symmetric arguments. Now it follows that R is preserved by the join-node-algorithm. $\square$

We conclude that all relations of $\mathcal{R}$ are preserved by the algorithms. As an example, we consider $R = h_{con}^{=\emptyset SL}(h_{con}^{=\emptyset \not S L}(h_{con}^{=\emptyset S \not L}(h_{del}^{=\emptyset \not S \not L}(\hat{R}))))$. With this relation $R \in \mathcal{R}$, we can answer the questions which ask for the probabilities of (among others) the following properties: 'at least half of the clients are connected to at least one server' or 'at most x servers are not connected to a client'. The classical S-terminal reliability problem can be solved with, e.g., the following relation: $h_{con}^{=\emptyset SL}(h_{con}^{=\emptyset S \not L}(h_{dmulS}^{\neq \emptyset SL}(h_{dmulS}^{\neq \emptyset S \not L}(h_{dmulS}^{=\emptyset SL}(h_{dmulS}^{=\emptyset S \not L}(h_{del}^{=\emptyset \not S L}(h_{del}^{=\emptyset \not S \not L}(\hat{R}))))))))$.

The next theorem summarises this section.

Theorem 2. *Let $R \in \mathcal{R}$ be an equivalence relation which is fine enough for property Y. There is an $O(n \cdot n_c^2 \cdot n_b^2 \cdot k)$ time algorithm for computing the probability that graph G has Y.*

Proof. The correctness follows directly from Theorem 1, Lemma 14 and the discussion of the global running time in Section 7.3. $\square$

Because of their special structure, we will analyse the running times of generated relations in the next section. We will see that applying certain operations bounds the number of classes n_c polynomially or by a constant.

5.3 Running Times of Generated Relations

In general, the running times of the procedures heavily depend on the chosen equivalence relations. The choice of the relation is a crucial point. Hence, it is very hard to make a statement about running times in general. For our generated or constructed relations, however, we are in a slightly better situation. The relation that can be used for the largest collection of properties $\hat{R}$ has exponentially many equivalence classes, and needs therefore exponential time. By applying our operations in $\mathcal{H}$, we reduce the number of equivalence classes by an 'roughly' determinable amount.

Instead of giving a list of all possible combinations of operations, we consider a few cases which appear especially useful. As an example, when using operation $h_{dmulS}^{=\emptyset S \not L}$, it can be reasonable to use operation $h_{dmulS}^{\neq \emptyset S \not L}$ (and $h_{dmulS}^{=\emptyset SL}$) as well. Furthermore, operation $h_{del}^{=\emptyset \not S \not L}$ can almost always be used, since empty blocks without flags cannot be used to make further connections, and they do not give any information about servers or clients.

When giving upper bounds for the number of representatives, we can distinguish between the number of possibilities due to the nonempty blocks and due to the empty blocks. For now, we only consider the number of possibilities due to the nonempty blocks, with no respect to flags, because this number is a constant and hence is not influenced by other input parameters apart from the treewidth k. Therefore, we have to consider at most $k+1$ elements per node which can be taken to form blocks and classes. A rough upper bound for this number is

$$\delta \ := \ \sum_{i=1}^{k+1} \binom{k+1}{i} \sum_{j=1}^{i} \left\{ \begin{matrix} i \\ j \end{matrix} \right\}$$

This can be shown as follows. The rightmost term gives the number of partitions of i elements into j nonempty subsets. It is referred to as the Stirling number of the second kind. See [6] for notations and details. With the inner summation, we get the number of partitions of i elements into at most i nonempty subsets. The binomial coefficient chooses i elements for partitioning among $k+1$ elements, which are the vertices of one node of the treedecomposition. Since not all nodes must have exactly $k+1$ vertices and not all vertices have to be up, we have to use the left summation. Although, this number can be rather large, it is a constant (which we refer to as δ) when k considered a constant. After this preparatory work, we consider a number of cases:

Case 1: The sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{dmulS}^{=\emptyset S \not L}, h_{dmulS}^{=\emptyset SL}, h_{dmulS}^{\neq \emptyset S \not L}, h_{dmulS}^{\neq \emptyset SL}$.

Any nonempty block can have 0 or 1 S-flags and between 0 and n_L L-flags. These give at most $2 \cdot (n_L + 1)$ possibilities per nonempty block. When working with a treedecomposition of width k, there can be at most $k+1$ nonempty blocks in any representative. For $k+1$ blocks, this results in at most $(2 \cdot (n_L + 1))^{k+1}$ possibilities. Now, we look at the number of possible equivalence classes caused by the empty blocks. We denote the number of possibilities of empty blocks with at least one L-flag and no S-flag by δ'. This number only depends on n_L. Each of these blocks can have 0 or 1 S-flags and since we have at most n_L such blocks, this results in at most 2^{n_L} possibilities. Furthermore, we can have between 0 and n_S empty blocks with exactly one S-flag and no L-flags. (There are no empty blocks without flags.) Altogether, we have that the number of equivalence classes is bounded by:

$$\delta \cdot (2 \cdot (n_L + 1))^{k+1} \quad \cdot \quad \delta' \cdot 2^{n_L} \cdot (n_S + 1)$$

Example 1. With the relation

$$R = h_{del}^{=\emptyset \not S \not L}(h_{dmulS}^{=\emptyset S \not L}(h_{dmulS}^{=\emptyset SL}(h_{dmulS}^{\neq \emptyset S \not L}(h_{dmulS}^{\neq \emptyset SL}(\hat{R})))))$$

we can solve among others the following problem for every x: What is the probability that there are at least x clients connected to each server?

Corollary 1. *If we do not allow empty blocks without any flags and if we bound the number of S-flags per block to be at most one, we have $O(n_S \cdot 2^{n_L})$ classes per node, i.e. linear in n_S, but exponential in n_L.*

Case 2: The sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{dmulS}^{=\emptyset S \not L}, h_{dmulL}^{=\emptyset \not S L}, h_{dmulS}^{=\emptyset SL}, h_{dmulL}^{=\emptyset SL}$, $h_{dmulS}^{\neq \emptyset S \not L}, h_{dmulL}^{\neq \emptyset \not S L}, h_{dmulS}^{\neq \emptyset SL}, h_{dmulL}^{\neq \emptyset SL}$.

For a nonempty block, there are 4 possibilities: no flag, one S-flag, one L-flag, one of each flags. Hence, we have at most 4^{k+1} possibilities for at most $k+1$ nonempty blocks. One equivalence class can have at most n_S empty blocks with exactly only one S-flag. The situation is similar for blocks with only L-flags, and the number of empty blocks with one

S-flag and one L-flag is bounded by $\min(n_S, n_L)$. In this case, the number of equivalence classes is bounded by:

$$\delta \cdot 4^{k+1} \quad \cdot \quad (n_S + 1) \cdot (n_L + 1) \cdot (\min(n_S, n_L) + 1)$$

Example 2. With the relation $R =$

$$h_{del}^{=\emptyset \not S \not L}(h_{dmulS}^{=\emptyset S \not L}(h_{dmulL}^{=\emptyset \not S L}(h_{dmulS}^{=\emptyset SL}(h_{dmulL}^{=\emptyset SL}(h_{dmulS}^{\neq\emptyset S \not L}(h_{dmulL}^{\neq\emptyset \not S L}(h_{dmulS}^{\neq\emptyset SL}(h_{dmulL}^{\neq\emptyset SL}(\hat{R})))))))))$$

we can solve among others the following problem for every x and y: What is the probability that there are x connected components with at least one client and no server, while there are at least y connected components containing at least one server and no client?

Corollary 2. *If we do not allow empty blocks without any flags and if we bound the number of S-flags and L-flags per block to be at most one, respectively, we have $O(n_S \cdot n_L \cdot \min(n_S, n_L))$ classes per node.*

Case 3: The sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{con}^{=\emptyset S \not L}, h_{con}^{=\emptyset \not S L}, h_{con}^{=\emptyset SL}$.

For one nonempty block, we have at most $(n_S+1) \cdot (n_L+1)$ possibilities for the number of L- and S-flags. Considering $k+1$ nonempty blocks results in $((n_S+1) \cdot (n_L+1))^{k+1}$ possibilities. We have at most one empty block with S-flags and no L-flags, which gives n_S+1 possibilities. The situation is similar for a possible block with only L-flags, and if we have a block with both types of flags, then there are $n_S \cdot n_L$ possible flag distributions. Multiplying these bounds results in an upper bound for the number of equivalence classes:

$$\delta \cdot ((n_S + 1) \cdot (n_L + 1))^{k+1} \quad \cdot \quad (n_S + 1) \cdot (n_L + 1) \cdot (n_S \cdot n_L + 1)$$

Example 3. With the relation

$$R = h_{del}^{=\emptyset \not S \not L}(h_{con}^{=\emptyset S \not L}(h_{con}^{=\emptyset \not S L}(h_{con}^{=\emptyset SL}(\hat{R}))))$$

we can solve among others the following problem: What is the probability that at least x clients are not connected to a server while at least y servers have no client connected to it?

Corollary 3. *If we do not allow empty blocks without any flags and if we have at most one empty block with S-flags, at most one empty block with L-flags and at most one empty block with S- and L-flags, then we have $O(n_S^{k+3} \cdot n_L^{k+3})$ classes per node, i.e. polynomial in n_S and n_L.*

Case 4: The sequence h contains: $h_{del}^{=\emptyset \not S \not L}, h_{con}^{=\emptyset S \not L}, h_{con}^{=\emptyset \not S L}, h_{con}^{=\emptyset SL}, h_{dmulS}^{=\emptyset S \not L}$, $h_{dmulL}^{=\emptyset \not S L}, h_{dmulS}^{=\emptyset SL}, h_{dmulL}^{=\emptyset SL}, h_{dmulS}^{\neq\emptyset S \not L}, h_{dmulL}^{\neq\emptyset \not S L}, h_{dmulS}^{\neq\emptyset SL}, h_{dmulL}^{\neq\emptyset SL}$ and the $h_{con}^{\cdots}$ operations are applied earlier than the $h_{dmulS}^{\cdots}$ and $h_{dmulL}^{\cdots}$ operations.

This means, that after applying h each block can have at most one flag of each type. As in Case 2, we have for the nonempty blocks 4^{k+1} possibilities. There are only 8 possibilities for the empty blocks, since we only can have the following three blocks: $\{\ \}^S, \{\ \}_L^S, \{\ \}_L$, which may or may not exist in an equivalence class. Hence, we have 8 possibilities. Altogether, the number of equivalence classes is bounded by:

$$\delta \cdot 4^{k+1} \quad \cdot \quad 8$$

Example 4. With the relation

$$R = h_{dmulS}^{=\emptyset S \not L}(h_{dmulL}^{=\emptyset \not S L}(h_{dmulS}^{=\emptyset SL}(h_{dmulL}^{=\emptyset SL}(h_{dmulS}^{\neq\emptyset S \not L}(h_{dmulL}^{\neq\emptyset \not S L}(h_{dmulS}^{\neq\emptyset SL}(h_{dmulL}^{\neq\emptyset SL}($$

$$h_{del}^{=\emptyset \not S \not L}(h_{con}^{=\emptyset S \not L}(h_{con}^{=\emptyset \not S L}(h_{con}^{=\emptyset SL}(\hat{R})))))))))))$$

we can solve among others the following problem: What is the probability that each client is connected to a server while each server has a client connected to it?

Corollary 4. *If we do not allow empty blocks without any flags and if we bound the number of S- and L-flags per block to be at most one, respectively, and if we bound the number of empty blocks to be constant, then we have a constant number of classes per node.*

These four corollaries show, how applying operations from $\mathcal{H}$ can help to reduce the number of classes. Recall that this yields algorithms that compute the corresponding probabilities of the properties for graphs, given with a treedecomposition of width at most k. The running time of such an algorithm is $O(n)$ times the number of equivalence classes (i.e. representatives) per node. In the next section, we consider answering questions, which ask for the probability that the surviving subgraph has property Y_1 and property Y_2. For such a combination of properties, we can use a combination of equivalence relations.

5.4 Combining Relations

Once we have relations for solving problems, we can combine them to create relations for 'combined problems'. We know that many relations allow us to solve more than one problem. Assume that we are faced with a problem like 'What is the probability that the surviving subgraph has properties A and B?' Let us assume further that we already have relations R^A and R^B for computing the probabilities of A and B, respectively. We will see that we can use R^A and R^B to create a new relation for property $(A \wedge B)$.

Lemma 15. *Let R^A and R^B be two relations, that are proper and fine enough for property A and B, respectively. Let $R^{A\wedge B}$ be the relation, such that for all C^1 and C^2 that are representatives of scenarios f_1 and f_2, respectively, we have:*

$$(C^1, C^2) \in R^{A\wedge B} \iff (C^1, C^2) \in R^A \wedge (C^1, C^2) \in R^B$$

Then $R^{A\wedge B}$ is proper and fine enough for property $(A \wedge B)$.

Proof. We have to show three properties: $blocks^{\neq\emptyset}(C^1) = blocks^{\neq\emptyset}(C^2)$, the preservation by the algorithms and the fineness of $R^{A\wedge B}$.

The first property is easy to see, because two scenario representatives C^1 and C^2 can only be equivalent under $R^{A\wedge B}$ if they are equivalent under R^A and R^B. Since R^A and R^B are proper, we have: $blocks^{\neq\emptyset}(C^1) = blocks^{\neq\emptyset}(C^2)$.

To see the preservation by the algorithms, we look at a join node i with children j_1 and j_2. We use the notation introduced in Section 3.2 and utilised in Section 4.5. Then we have:

$$
\begin{aligned}
& (C^1_{j_1}, C^2_{j_1}) \in R^{A\wedge B} \quad \wedge \quad (C^1_{j_2}, C^2_{j_2}) \in R^{A\wedge B} \\
\Longrightarrow\ & \begin{bmatrix} (C^1_{j_1}, C^2_{j_1}) \in R^A \\ \wedge \\ (C^1_{j_1}, C^2_{j_1}) \in R^B \end{bmatrix} \wedge \begin{bmatrix} (C^1_{j_2}, C^2_{j_2}) \in R^A \\ \wedge \\ (C^1_{j_2}, C^2_{j_2}) \in R^B \end{bmatrix} \\
\Longrightarrow\ & \begin{bmatrix} (C^1_{j_1}, C^2_{j_1}) \in R^A \\ \wedge \\ (C^1_{j_2}, C^2_{j_2}) \in R^A \end{bmatrix} \wedge \begin{bmatrix} (C^1_{j_1}, C^2_{j_1}) \in R^B \\ \wedge \\ (C^1_{j_2}, C^2_{j_2}) \in R^B \end{bmatrix} \\
\Longrightarrow\ & \begin{bmatrix} (C^{11}_i, C^{12}_i) \in R^A \wedge \\ (C^{12}_i, C^{21}_i) \in R^A \wedge \\ (C^{21}_i, C^{22}_i) \in R^A \end{bmatrix} \wedge \begin{bmatrix} (C^{11}_i, C^{12}_i) \in R^B \wedge \\ (C^{12}_i, C^{21}_i) \in R^B \wedge \\ (C^{21}_i, C^{22}_i) \in R^B \end{bmatrix} \\
\Longrightarrow\ & \begin{bmatrix} (C^{11}_i, C^{12}_i) \in R^{A\wedge B} \wedge \\ (C^{12}_i, C^{21}_i) \in R^{A\wedge B} \wedge \\ (C^{21}_i, C^{22}_i) \in R^{A\wedge B} \end{bmatrix}
\end{aligned}
$$

Hence, $R^{A\wedge B}$ is preserved by the join-node algorithm. The preservation by the introduce-node and forget-node algorithm can be proven in an analogous way.

Next, we prove the fineness of the relations. We look at a node i of T. There we select an equivalence class representative C of $R^{A\wedge B}$. We will derive a contradiction: Assume we have two scenario representatives C^1 and C^2 with: C^1 has property $(A \wedge B)$ and C^2 does not have $(A \wedge B)$ and $C^1, C^2 \in C$. That means 'C^1 has A and C^1 has B' and 'C^2 does not have A or C^2 does not have B' and:

$$
\begin{gathered}
(C^1, C^2) \in R^{A\wedge B} \quad \Longrightarrow \\
(C^1, C^2) \in R^A \quad \wedge \quad (C^1, C^2) \in R^B \\
\Longrightarrow \\
\begin{bmatrix} [C^1 \text{ has } A \ \wedge \ C^2 \text{ has } A] \\ \vee \\ [C^1 \text{ has } \neg A \wedge C^2 \text{ has } \neg A] \end{bmatrix} \wedge \begin{bmatrix} [C^1 \text{ has } B \ \wedge \ C^2 \text{ has } B] \\ \vee \\ [C^1 \text{ has } \neg B \wedge C^2 \text{ has } \neg B] \end{bmatrix}
\end{gathered}
$$

From the assumption 'C^1 has $(A \wedge B)$', we have:

$$C^1 \text{ has } A \wedge C^1 \text{ has } B \Longrightarrow C^2 \text{ has } A \wedge C^2 \text{ has } B,$$

which is a contradiction to 'C^2 has $\neg(A \wedge B)$', and hence $R^{A\wedge B}$ is fine enough for $(A \wedge B)$, which completes the proof. □

Other Combinations. The previous lemma only considers the combination of two relations with the logical 'and'. A natural question to ask is whether combining relations can also be applied to the logical 'not' and the logical 'or'.

Given a graph property A and a relation R^A to compute the probability p^A that the surviving subgraph has property A. To compute the probability $p^{\neg A}$ that the surviving subgraph does not have property A, i.e. it has $\neg A$, we can simply compute $p^{\neg A} = 1 - p^A$ using R^A. Hence, the relations R^A and $R^{\neg A}$ for property A and $\neg A$, respectively, are identical.

We assume now, that we have two relations R^A and R^B for properties A and B, respectively. Defining $R^{A \vee B}$ in the way of Lemma 15 will not result in an equivalence relation, because due to the transitivity of such a relation, two scenarios would be equivalent under $R^{A \vee B}$ even if they are neither equivalent under R^A nor under R^B. However, using basic logical rules, we still can compute the probability $p^{A \vee B}$ that the surviving subgraph has property A or property B (whereat $p^{\neg A \wedge B}$ is the probability that the surviving subgraph does not have property A, but it has property B; $p^{A \wedge B}, p^{A \wedge \neg B}, p^{\neg A \wedge \neg B}$ are defined accordingly):

$$p^{A \vee B} = p^{A \wedge B} + p^{\neg A \wedge B} + p^{A \wedge \neg B} = 1 - p^{\neg A \wedge \neg B}$$

The two previous Lemmas 14 and 15 give us powerful tools for generating and combining relations. For these new relations, it is not necessary to prove the properness and fineness, respectively, as long as the original relations have these properties. We will look at examples of questions that can be answered with the framework:

- What is the probability that in the surviving subgraph not each client is connected to a server?
- What is the probability that in the surviving subgraph each client is connected to a server and that each server has a client connected to it?
- What is the probability that in the surviving subgraph all servers are connected or that each server is connected to at least 5 clients?

For these, we need equivalence relations for the single properties in each question. These single properties, however, fit into our framework and hence, due to Lemma 15, the three combined properties of the questions above fit as well.

6 Additional Ideas - The Expected Number of Components with a Certain Property

We give a description of an example relation R, which cannot be generated by the methods of the previous sections. The goal is to answer the questions: 'What is the expected number of components with at least one server and no clients?' (at least one client, no servers; at least one client and at least one server) This question differs from 'What is the probability that G has property Y?' Even though, our framework can be used to answer this question, since the representatives of the equivalence classes 'show enough information'. However, some modifications are necessary.

Definition of *R*. We consider two scenario representatives C^1 and C^2 of node i, The scenarios represented by C^1 and C^2 are defined to be equivalent, if they are equal on their nonempty blocks, i.e.:

$$(C^1, C^2) \in R \Longleftrightarrow$$

$$h_{del}^{=\emptyset \not S \not L}(h_{del}^{=\emptyset S \not L}(h_{del}^{=\emptyset \not S L}(h_{del}^{=\emptyset S L}(C^1)))) = h_{del}^{=\emptyset \not S \not L}(h_{del}^{=\emptyset S \not L}(h_{del}^{=\emptyset \not S L}(h_{del}^{=\emptyset S L}(C^2))))$$

Even though we can use the operations of Section 5.2 to define the considered equivalence relation, we are not able to use our framework without further modifications. This is because the representative C of an equivalence class $[C]$ of node i will have a special structure. All empty blocks will be replaced by three special blocks: $\{e_S(C)\}^S, \{e_L(C)\}_L$ and $\{e_{SL}(C)\}_L^S$. The nonempty blocks will be the same as the ones of the represented scenarios and $e_S(C), e_L(C)$ and $e_{SL}(C)$ are real numbers. We will only go into details for $e_S(C)$, since for $e_L(C)$ and $e_{SL}(C)$ it is analogue. The number $e_S(C)$ is the expected number of components of G_i, with empty intersection with X_i, with at least one server and no client given the fact that G_i is in one of the scenarios represented by C. The number of components of $G[V_i^{f=1}]$ with at least one server and no client and with empty intersection with X_i is $|\mathit{blocks}^{=\emptyset S\not L}(\mathit{blocks}_i(f))|$, which is the number of empty blocks with at least one S-flag and no L-flags of the representative for scenario f. Then we have:

$$e_S(C) = \sum_{f \in C} \Pr(f) \cdot |\mathit{blocks}^{=\emptyset S\not L}(\mathit{blocks}_i(f))|$$

Fineness and Preservation of *R*. The preservation is easy to see, since the algorithms only affect the nonempty blocks. Thus, in each case equivalent scenarios are treated in the same way and, hence, are equivalent after processing by the algorithms. A formal proof uses the same ideas and would be similar to the proof of Lemma 14.

The fineness of a relation is defined using the fact that scenarios can or cannot have specific properties. On the other hand, 'the expected number of components with at least on server and no client' is a number and not a property. That is why the term 'fine enough' is not really applicable for our question. However, we can use the equivalence classes of R in the following way. We consider a node i and all its equivalence classes $[C]$. Then, the expected number of components of G_i with at least one server and no client is:

$$\sum_{[C]:[C] \text{ is eq.class of } i} \Pr([C]) \cdot (e_S(C) + |\mathit{blocks}^{\neq\emptyset S\not L}(C)|)$$

Modification of the Algorithms. Since we have a different structured problem, we modify our algorithms slightly. These modifications maintain e_S from node to node (as well as e_L and e_{SL}). Introduce-nodes need no special treatment, because the set of empty blocks is not changed. This is different with forget-nodes. We consider an equivalence class $[C]$. If we create a new empty block with only S-flags, then we delete it and increase $e_S(C)$ by one. The correctness of this is easy to see, since all scenarios of $[C]$ would now have one more empty block with only S-flags. For a join-node i with children j_1 and j_2, we add the two e_S numbers of the 'source' representatives together to obtain $e_S(C)$ of the resulting class C. This can be understood by bringing to mind that an equivalence class C of i represents scenarios of G_i which contains as subgraphs G_{j_1} and G_{j_2}. Furthermore, the empty components of G_{j_1} and G_{j_2} are not influencing each other.

There is one thing left we have to do: the modification of the code-fragment in Step 2 in Section 4.2. If two equivalence classes $[C^1]$ and $[C^2]$ become equivalent, we have to modify $e_S(C)$ of the resulting equivalence class $[C]$, using $e_S(C^1)$ and $e_S(C^2)$ in the following way, which is not hard to see.

$$e_S(C) = \frac{e_S(C^1) \cdot \Pr(C^1) + e_S(C^2) \cdot \Pr(C^2)}{\Pr(C)}$$

The numbers $e_L(C)$ and $e_{SL}(C)$ are computed in the same way.

Running Time. The overall running time of the algorithms is $O(n_c^2 \cdot n_b^2 \cdot k^2)$. For our relation R we have $n_b \leq k+1+3$, since we have at most $k+1$ nonempty blocks and 3 special blocks, which actually do not play a role in class distinction. The analysis of the maximum number of equivalence classes is similar to Case 4 in Section 5.3. Hence, n_c is a constant.

Conclusion. The problem 'What is the expected number of components with at least one server and no clients?' does not fit into the framework as described in the previous sections. However, we have seen that with additional modifications of the framework we can obtain an algorithm to solve this problem in linear time on graphs with given treedecomposition of width at most k.

7 Discussion

We presented a framework for a variety of network reliability problems for graphs of bounded treewidth. The method itself is open enough for extensions which may enable faster running times and/or additional properties to check. Those properties that can be checked with classes, i.e. those properties for which our framework is applicable, concern connections between sets of vertices. It remains a further research topic, to develop a more powerful framework that can handle much more properties, e.g. properties like 'What is the probability that the surviving subgraph has a dominating set of size $\leq k$?'

7.1 Edge Failures

The framework described in this paper considers only vertex failures. As mentioned earlier edge failures can be simulated by vertex failures. To do this, we place a new vertex on each edge. The reliability of the new vertex will be the one of the corresponding edge. All edges of the resulting graph will be defined to be perfectly reliable. We clearly increase the number of vertices of the graph by doing this. The resulting graph has $|V|+|E|$ vertices. For arbitrary graphs it holds $|E| \leq \frac{|V|\cdot(|V|-1)}{2}$ which would mean a potential quadratic increment. When restricted to graphs of treewidth at most k, we have $|E| \leq k \cdot |V| - \frac{k\cdot(k+1)}{2}$ (see [2]). Hence, only an increment linear in $|V|$, if k considered a constant. The effect on the treewidth of a graph itself is also rather secondary. We consider an edge $\{u, v\}$. In our treedecomposition, we have one node i with $\{u, v\} \subseteq X_i$. When placing vertex w on this edge, we can simply attach a new node $\{u, v, w\}$ to the node i. It is easy to see that this results in a proper treedecomposition and does not increase its width, if the width is at least 2.

7.2 Treewidth vs. Pathwidth

In Section 4.4, we required a relation R to be preserved by all three algorithms. We can refrain from this demand, if we look at graphs of bounded pathwidth. A path decomposition of G is a treedecomposition (T, X) of G whereat T is a path. Hence, join-nodes do not appear and thus R has not to be preserved by the join-node-algorithm. This may enable more relations, but at the other hand we have to use a path-decomposition.

7.3 Running Times

The running time for our algorithm heavily depends on the chosen relation R. We can see it is very important to choose a relation 'as coarse as possible'. All considered algorithms run in time $O(n_c^2 \cdot n_b^2 \cdot k^2)$ per node. In [7], Kloks shows that there are nice treedecompositions with at most $4 \cdot n$ nodes (thus linear in $n = |V(G)|$). Hence, we have the global running time $O(n \cdot n_c^2 \cdot n_b^2 \cdot k^2)$.

Furthermore, the framework provides enough possibilities for using additional tricks, ideas and modifications. It may be necessary to use them to get the best running time, as shown in Section 6. One general possibility to decrease the running time would be to delete classes that can never have the required property as soon as possible. However, the performance gain is not easy to analyse.

Acknowledgements

Many thanks to my supervisor Hans Bodlaender and to my colleague Frank van den Eijkhof for their very useful discussions and comments, and also thanks to Peter Lennartz, Peter Verbaan and Pauline Vollmerhaus.

References

1. S. Arnborg, A. Proskurowski: *Linear time algorithms for NP-hard problems restricted to partial k-trees.* Discrete Appl. Math. 23, (1989), 11-24.
2. H. L. Bodlaender: *A linear-time algorithm for finding tree-decompositions of small treewidth.* SIAM J. Comput. 25/6 (1996), 1305-1317.
3. H. L. Bodlaender: *A tourist guide through treewidth.* Acta Cybernet. 11 (1993), 1-23.
4. H. L. Bodlaender: *Treewidth: Algorithmic techniques and results.* In I. Privara and P. Ruzicka, editors, Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS '97, LNCS 1295, (1997), 19-36.
5. J. Carlier, C. Lucet: *A decomposition algorithm for network reliability evaluation.* Discrete Appl. Math. 65, (1996), 141-156.
6. R. L. Graham, D. E. Knuth, O. Patashnik: *Concrete Mathematics.* Addison-Wesley Publishing Company, Amsterdam, (1989).
7. T. Kloks: *Treewidth. Computations and Approximations.* LNCS 842, (1994).
8. C. Lucet, J.-F. Manouvrier, J. Carlier: *Evaluating Network Reliability and 2-Edge-Connected Reliability in Linear Time for Bounded Pathwidth Graphs.* Algorithmica 27, (2000), 316-336.
9. E. Mata-Montero: *Reliability of Partial k-tree Networks.* Ph.D. Thesis, Technical report: CIS-TR-90-14, University of Oregon, (1990).
10. J. S. Provan, M. O. Ball: *The complexity of counting cuts and of computing the probability that a graph is connected.* SIAM J. Comput. 12/4, (1983), 777-788.
11. A. Rosenthal: *Computing the reliability of complex networks.* SIAM J. Appl. Math. 32, (1977), 384-393.