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

Thomas Wolle

Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
thomasw@cs.uu.nl

Abstract. In this paper, we consider problems related to the network reliability problem, restricted to graphs of bounded treewidth. We look at undirected simple graphs with each vertex and edge a number in $[0, 1]$ associated. These graphs model networks in which sites and links can fail, with a given probability, independently of whether other sites or links fail or not. 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 computing for a large number of different properties Y whether Y holds for 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 S-terminal reliability problem can be solved in linear time 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 this paper, we investigate the problem to compute the probability that a given network has a certain property. We use the following model: To each site (vertex) of the network, a value in $[0, 1]$ is assigned, which expresses the probability that this site is working. Failures of links (edges) between sites can be simulated by introducing further vertices, see Section 5.1 for details. 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 [8]. When restricting the set of input networks to graphs of bounded treewidth, linear time algorithms are possible.

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

P. Bose and P. Morin (Eds.): ISAAC 2002, LNCS 2518, pp. 137–149, 2002.

Much work has been done on algorithms for graphs of bounded treewidth. A general description of a common 'bottom up' method for such graphs can be found in [3] or [4]. The (special case of the) S-terminal reliability problem where only edges can fail and nodes are perfectly reliable was considered by Arnborg and Proskurowski [1]. They show 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 [7]. Rosenthal introduced in [9] a decomposition method, which was applied by Carlier and Lucet to solve network reliability problems with edge and node failures on general graphs [5]. Carlier, Manouvrier and Lucet solved with this approach the 2-edge connected reliability problem for graphs of bounded pathwidth [6]. We will generalize both the type of the problems which can be solved, and the method used in [5] and [6].

For graphs of bounded treewidth with associated non-failure probabilities, we present a generic framework which is much more general than previously presented work. A large number of questions can be answered, which can be more complicated than the classical S-terminal reliability problem. These questions ask for the probability that the graph of the 'up' elements has a certain property. We allow that there are two types of special vertices: 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, cover the connectivity of vertices of the set S among each other, 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 our algorithm. This starts with linear time and can increase to exponential time (for solving a problem for which all possible information is necessarily preserved, and hence no diminishing of information and running time is possible).

2 Definitions

Definition 1. *A* tree decomposition *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 tree decomposition $((I, F), \{X_i \mid i \in I\})$ is $\max_{i \in I} |X_i| - 1$. The* treewidth *of a graph G is the minimum width over all tree decompositions of G.*

A tree decomposition (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 exactly one child* j *with* $X_i = X_j \cup \{v\}$ *for some vertex* $v \in V$.
- Forget nodes i *have exactly 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 tree decomposition can be converted into a nice tree decomposition of the same width in linear time (see [4]). 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, E the set of edges and k its treewidth. $T = (I, F)$ is a nice tree decomposition with width k of the graph G. The term *vertex* refers to a vertex of the graph and *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 is:

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

Clearly, there are $2^{|V|}$ scenarios. The elements in $L \subseteq V$ are representing 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 model 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. The following definition summarises this paragraph.

Definition 2. *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*,* $n_S = |S|$

3 The Technique

3.1 More Special Definitions

We assume to have a nice tree decomposition $(T, X) = ((I, F), \{X_i \mid i \in I\})$ of $G = (V, E)$. As described in many articles, e.g. [3] and [4], we will use a bottom up approach with this tree decomposition. We will compute partial solutions for the subgraphs corresponding to the subtrees of T 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. The following definition provides us the terminology for the subgraphs.

Definition 3. *Let $G = (V, E)$ be a graph and $(T = (I, F), X = \{X_i \mid i \in I\})$ a nice tree decomposition of G.*

- $V_i = \{v \in X_j \mid j = i$ *or* j *is an 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. Every scenario f causes G_i to be decomposed into one or more connected components $O_1, ..., O_q$, i.e. $G[V_i^{f=1}] = O_1 \cup ... \cup O_q$. For these components, we consider the intersection with X_i These intersection sets $B_1 = O_1 \cap X_i, ..., B_r = O_q \cap X_i$ are called *blocks*. Thus, each scenario f specifies a multiset of blocks. (It is a multiset since more than one component may have an empty intersection with X_i.) 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 '$\#L\text{flags}(B)$' to refer to the number of L-flags of block B (for S-flags, we use $\#S\text{flags}(B)$).

Definition 4. *A* block *B of the graph G_i is a (perhaps empty) subset of X_i possibly extended by L- and S-flags. For $\{v_1, ..., v_t\} \subseteq X_i$, we write $B = \{v_1, ..., v_t\}$ or $B = \{v_1, ..., v_t\}_{S...S}^{L...L}$, respectively.*

Classes. For each scenario f, we associate the multiset of its blocks to node i. We call this representation of f its *class* C^f. To G_i we associate a multiset of classes, one for each possible scenario. Note that two different scenarios can have identical classes, but these will be considered as different objects in the multiset of the classes of G_i. Later in this paper, equivalence relations on scenarios (and hence on classes) are defined, and thus equivalence classes will be formed that

can correspond to more than one scenario. The most obvious of these is to assume that scenarios are equivalent when their classes are identical, but also less obvious equivalence relations will be considered. It will be seen that the algorithmic techniques for computing the collection of classes 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.

Let $G[V_i^{f=1}] = O_1 \cup ... \cup O_q$. Then we have the following blocks $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$, which simply make up the class $C^f = (B_1, ..., B_q)$. A block B of a class C^f 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. The next definition summarise this.

Definition 5. *A* class $C^f = (B_1, ..., B_q)$ *of* G_i *is a multiset of blocks of* G_i*, such that scenario* f *specifies* $B_1 = O_1 \cap X_i, ..., B_q = O_q \cap X_i$*. At this,* $\#Lflags(B_j) = |O_j \cap L|$ *and* $\#Sflags(B_j) = |O_j \cap S|$*, for* $j = 1, ..., q$*, and* $G[V_i^{f=1}] = O_1, ..., O_q$*.*

We denote by n_b the maximum number of blocks of a class over all classes, and by n_c the maximum number of classes of a node over all nodes. The numbers n_c and n_b are crucial points to the algorithm running time as we will see. Thus, in the subsequent Section 4.1 we look at equivalence relations between classes to reduce their number.

Example classes. Let $X_i = \{u, v, w\}$. Some of the possible classes for i 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 components with empty intersection with X_i and this component contains a vertex of S

Here, the classes 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 4.1).

3.2 The Method for Network Reliability Problems

As already mentioned, we compute the classes for every node of the nice tree decomposition using a bottom up approach. However, we will not only compute the classes C but their probabilities $\Pr(C)$ as well. For a class C of node i, $\Pr(C)$ is the probability that G_i is in a scenario f which is represented by C. Lemma 1 and similar statements for leaf, forget and join nodes, show us that

therefore we only need the (already computed) classes of the children of the node, insofar they exist. Once we have the probabilities of all classes of all nodes, and especially of the root, we easily can solve problems by summing up the requested probabilities.

Starting with the leaves of the nice tree decomposition, we compute for each node all classes that are possible for this node, and their corresponding probabilities. In this computation, the classes of its child(ren) play at this a decisive role. When 'walking up' the tree decomposition to a node i the classes will be refined and split into other classes representing the states of the vertices of X_i and their connections. Once more, we refer to Section 4.1 in which equivalence relations for reducing the number of classes are described. Then we will see that classes are not only refined, while walking up the tree, but at the same time, some classes will 'collapse' into one (equivalence) class, since they become equivalent.

3.3 How to Compute the Classes and Their Probabilities?

Due to space constraints, we only discuss introduce nodes here.

Introduce Nodes. Let i be an introduce node with child j, such that $X_i = X_j \cup \{v\}$. We use the classes and their probabilities of node j to compute the classes and their probabilities of node i. For any state of v (up or down, in L, in S, or not) we compute and add the classes to those of i. This is done by considering the following two cases for each class C_j of j, and modifying it to get a class C_i of i:

- 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 merge the blocks of this class as described below. The probability of the resulting new class C_i is: $\Pr(C_i) = p(v) \cdot \Pr(C_j)$.
- v is down; We will not make any changes to the class C_j to get C_i, since no new connections can be made and no flags were added. However, we compute the probability: $\Pr(C_i) = (1 - p(v)) \cdot \Pr(C_j)$.

Lemma 1. *There is an algorithm that computes the classes and their probabilities of an introduce node i correctly, given all classes and their probabilities of the child j of node i.*

Such an algorithm performing the description above, can be implemented to run in time $O(n_c \cdot n_b^2 \cdot k^2)$.

Merging of Class Blocks. We describe a procedure that is a subroutine of the introduce- and join-node-algorithm. It is used to maintain the structure of classes (and has nothing to do with equivalence classes). Due to an introduction of a new vertex (see above) or joining two classes a class can contain two 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 hurts the proper structure of a class. Hence, we *merge* such non-disjoint blocks B_a and B_b into one block B. We have to add to B the number of L flags of both blocks B_a and 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. An algorithm for merging can be implemented to run in $O(k^2 \cdot n_b^2)$ time.

4 Results

Our results can be divided into two groups. The first group does not use an equivalence relation between classes. Its consideration is a kind of warm up for the second group, which uses equivalence relations between classes (see Section 4.1).

So far, a class 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. The blocks of C reflect the components and the flags the number of vertices of S and L in each component of the graph $G[V_i^{f=1}]$. Hence, any question which asks for the probability of a certain property concerning the components and connectivity of vertices of S and L can be answered. Clearly, only graphs can have such properties, but we will generalise this to scenarios and classes.

A scenario f for node i *has* property Y, if $G[V_i^{f=1}]$ has property Y. A class C of node i *has* property Y, if the scenario represented by C has property Y.

If we want to use classes for solving problems, it is also very important to be able to make the decision whether a class has a certain property just by considering the information given by the representation of the class (namely the blocks of the class). A class C of node i *shows* property Y, if the information given by the representation of C is sufficient to determine that C has Y.

The following definition describes all the properties that can be *checked* with this approach. Therefore an equivalence between showing and having is necessary.

Definition 6. *A property Y of G_i can be* checked *by* using classes *of i, if for all classes C of i hold:*

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

After computing all classes for node i in a bottom up manner, we can use these classes to easily solve several problems for G_i.

Lemma 2. *Let Y be a property that can be checked by the classes of i. The probability that the subgraph of G_i induced by the up vertices has property Y equals the sum over the probabilities of the classes of node i which show this property Y.*

From the definitions and lemma above, we easily conclude that each question that asks for the probability of a property Y of G_i can be answered, if Y can be checked using the classes of i.

Since there are $O(2^{|V_i|})$ scenarios, this results in a method with exponential running time, because the number of classes is a crucial point for the running time of the algorithm. Thus, in the next Section 4.1, we consider how to reduce the number of classes. However, the more classes per node we 'allow' the more information we conserve during the bottom up process and hence the more problems can be solved.

4.1 Reducing the Number of Classes

In this section we will have a global look at the strategy. The correctness and restrictions follow in subsequent sections. We will use the terms class and equivalence class, which can cause confusion. Classes refer to 'objects' containing only one scenario, while equivalence classes may represent several classes or scenarios, respectively. Indeed, it is the case that equivalence classes can be used in the algorithms instead of classes.

By using an equivalence relation R between classes it is possible to reduce the number of objects, i.e. equivalence classes, handled by the algorithms. Another interpretation is that we define an equivalence relation between scenarios, since so far each class represents only one scenario. It is sensible to define classes to be equivalent, only if they belong to the same node. Equivalent classes must be 'similar' with regards to the way they are processed by the algorithms. We have to show that R is 'preserved' by the algorithms, i.e. if we have two equivalent classes of a node as input to an algorithm, then the resulting classes are also equivalent. (This is described in detail in Section 4.3.) Additionally, we have to take care that the equivalence relation R we have chosen, is suitable to solve our problem.

Then we are able to use the algorithms with the equivalence classes (or better: with their representatives) instead of classes. We have to modify the algorithms slightly, since we have to check after processing each node, if two classes or two equivalence classes became equivalent. If this is the case, we join them into one equivalence class with new probability the sum of the joined (equivalence) classes. The following list of steps summarises what we have to do.

- Step 1: We define an equivalence relation R which is fine enough to solve our problem (see Section 4.2). On the other hand, R should be 'coarse' to reduce the number of equivalence classes and hence the running time.
- Step 2: We must modify our algorithms to maintain equivalence classes, i.e. it is necessary to check for equivalent (equivalence) classes after processing each node. If we find such two (equivalence) classes, we keep only one of them with accumulated probability. For doing this, we add some code to the algorithms, which can be implemented to need $O(n_c^2 \cdot n_b^2)$ steps.
- Step 3: We have to show the correctness of using the representatives of R instead of classes, i.e. that the algorithms preserve R (see Section 4.3).

– Step 4: We analyse the running time by estimating the maximum number of equivalence classes and the maximum number of blocks.

4.2 Information Content of Equivalence Classes

As at the beginning of Section 4, we introduce some notation for equivalence classes. Therewith, we want to express the information 'contained' in equivalence classes. 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 classes to be equivalent which may not be equal. By doing this, we lose the information that these classes were different and also why they were different. However, we reduce the number of classes, as intended. Clearly, using no equivalence relation or one which defines every class (scenario) to be in an extra equivalence class provides as much information as possible. Such a relation would be the finest we can have. 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 at the beginning of Section 4, we give the following definition for equivalence classes.

An equivalence class C^* of classes for node i *has* property Y, if each class belonging to C^* has property Y. An equivalence class C^* of node i *shows* property Y, if its representative shows Y. Despite the abstract level, we formalise when equivalence classes provide enough information for solving problems.

Definition 7. *An equivalence relation R is fine enough for property Y, if for all nodes i of the tree decomposition holds: For all equivalence classes C^* of node i must hold:*

$$(\forall f \text{ represented by } C^* : f \text{ has property } Y)$$

$$\vee$$

$$(\forall f \text{ represented by } C^* : f \text{ has not property } Y)$$

That means R is fine enough for Y, if there is no equivalence class which contains a scenario that has property Y and another one which does not have property Y. In the same flavour as Lemma 2, the next one tells us that under certain conditions, we can use equivalence classes to solve problems for G after computing all equivalence classes of all nodes until we reached the root r.

Lemma 3. *Let R be an equivalence relation which is fine enough for property Y that can be checked using classes. 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.*

4.3 When and Why Is Using Equivalence Relations Correct?

We already know, 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 algorithm. 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. On the other hand R should be coarse. 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 classes are equivalent before processed by an algorithm, they are also equivalent after that. To capture exactly the notion of 'preserving an equivalence relation', some definitions are required. However, in this extended abstract, we only look at the algorithm for introduce-nodes. The upper index x in C_j^x in this definition represents not a scenario, but an index to distinguish different classes of node j.

Definition 8. *Let i be an introduce node with child j with $X_i = X_j \cup \{v\}$. C_j^x is a class of node j which results in the classes $C_i^{x,up}$ and $C_i^{x,down}$ of node i, for v is up or down, respectively. An introduce-node-algorithm* preserves *the equivalence relation R if it holds:*

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

If we use only (the representatives of) the equivalence classes in the algorithms, we can process all classes (scenarios) represented by this equivalence class by only one 'central execution' of an algorithm. The result will be, again, equivalence classes. When and why is this correct? The next lemma, which is easy to see, tells us that indeed all classes (scenarios) represented by one equivalence class of R can be handled by just one process of an algorithm if R is preserved by this algorithms.

Lemma 4. *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.*

That means, we do not have to be careful whether an algorithm 'hurts' R by creating a result that actually is not an equivalence class of R. We are now ready to bring everything together.

Theorem 1. *Let R be an equivalence relation of classes that is fine enough for property Y which can be checked using classes. Furthermore, let R be preserved by the forget-, introduce- and join-node-algorithm. Then we can use our framework to compute the probability that the subgraph of G_i induced by the up vertices has property Y.*

The running time depends on the relation R. 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.

4.4 Solvable Problems

To give an exhaustive list of problems that can be handled with this approach, is beyond the scope of this paper, because many equivalence relations can be used, and with each such relation very often many questions can be answered. Fortunately, the last theorem tells us something 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. Nevertheless, we list some example properties.

To get relations that allow 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 S-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 reasonable assumption could 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 fewest average 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 Discussion

5.1 Edge Failures

As mentioned earlier edge failures can be simulated by vertex failures. Therefore, we place a new vertex on each edge. Its reliability 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 then $|V| + |E|$ 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 tree decomposition, 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 tree decomposition and does not increase its width, if the width is at least 2.

5.2 Treewidth vs. Pathwidth

In Section 4.3, 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 tree decomposition (T, X) of G whereas 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.

5.3 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'. 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. 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 our colleagues Hans Bodlaender and Frank van den Eijkhof for their very useful discussions and comments, and also thanks to Peter Lennartz and Ian Sumner.

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. 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
7. E. Mata-Montero: *Reliability of Partial k-tree Networks.* Ph.D. Thesis, Technical report: CIS-TR-90-14, University of Oregon, (1990)
8. 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
9. A. Rosenthal: *Computing the reliability of complex networks.* SIAM J. Appl. Math. 32, (1977), 384-393