Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 291 (2013) 15–25

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Memory-efficient Bounding Algorithm for the Two-terminal Reliability Problem

Minh Lê[a,1] Max Walter[b,2] Josef Weidendorfer[a,3]

[a] *Lehrstuhl für Rechnertechnik und Rechnerorganisation*
TU München
München, Germany

[b] *Siemens AG*
Nürnberg, Germany

Abstract

The terminal-pair reliability problem, *i.e.* the problem of determining the probability that there exists at least one path of working edges connecting the terminal nodes, is known to be NP-hard. Thus, bounding algorithms are used to cope with large graph sizes. However, they still have huge demands in terms of memory. We propose a memory-efficient implementation of an extension of the Gobien-Dotson bounding algorithm. Without increasing runtime, compression of relevant data structures allows us to use low-bandwidth high-capacity storage. In this way, available hard disk space becomes the limiting factor. Depending on the input structures, graphs with several hundreds of edges (*i.e.* system components) can be handled.

Keywords: terminal pair reliability, partitioning, memory migration, factoring

1 Introduction

The terminal-pair reliability problem has been extensively studied since the 1960s [12]. It determines the reliability of a binary-state system whose redundancy structure is modelled by a combinatorial graph. The edges correspond to the system components and can be in either of two states: failed or working, whereas the nodes are assumed to be perfect interconnection points. Though all components fail statistically independent, the problem was shown to be NP-complete [9]. Many algorithms have been developed over time. They can be categorized into the following classes:

(i) Sum of disjoint product (SDP) [6,14]

[1] Email:lem@in.tum.de

[2] Email:max.walter@siemens.com

[3] Email:weidendo@in.tum.de

doi:10.1016/j.entcs.2012.11.015

(ii) Cut and path-based state enumerations with reductions, [15,10,17]

(iii) Factoring algorithm with reductions [1,7,11]

(iv) Edge Expansion Diagram (EED) using Ordered Binary Decision Diagram (OBDD) [4]

The methods using SDP require enumeration of minimal paths or cuts of the network in advance. Therefore class (i) is related to class (ii). The vital drawback of methods from class (i) is that the computational effort in disjointing the minimal path or cut sets grows rapidly with the network size. Instead, it is more recommended to apply (iii) or (iv) [4]. By exploiting isomorphic subgraph structures, (iv) turns out to be quite efficient for large recursive network structures. However, the efficiency of the OBDD-based methods depends largely on BDD variable ordering which itself has time complexity of $\mathcal{O}(n^2 \cdot 3^n)$ [5]. Unfortunately, both aforementioned methods lack the ability of providing any valuable results in case of non termination. Considering that in general a reliability engineer is satisfied with a good approximate result (to a certain order of magnitude), the bounding algorithm suggested by Gobien and Dotson [15] using reductions proves to be a suitable method. Based on Boolean algebra it determines mutually disjoint success and failure events. Yoo and Deo underlined the efficiency of this method in direct comparison with four other efficient algorithms [16]. However, no attention has been paid to the rapidly increasing memory consumption of this method. In other words, the accuracy of the computed bounds are restricted to the size of the available memory. Hence, in this work we propose a way to overcome this limitation without significantly deteriorating the computation time. This is done by migrating the associated data structures held in memory to low bandwidth high-capacity storage. As a result we can cope with inputs of larger dimensions and additionally obtain more accurate bounds. Furthermore, the memory consumption can be seen as negligible as only the initial input graph and probability maps are stored in memory. After giving the definition of the two-terminal reliability problem and the idea of the appropriate Gobien Dotson algorithm in section 2, we will first explain how to optimize the memory consumption of this approach and subsequently migrate the memory content to hard disk in section 3. In section 4 we illustrate some results of the modified algorithm performed on several benchmark networks. Finally the results are summarised and an outlook is given in the last section.

Throughout the paper, we use the following acronyms:

RBD Reliability Block Diagram
bfs breadth first search
bw bandwidth

2 Preliminary

Definition 2.1 The redundancy structure of a system to be evaluated is modeled by an undirected multigraph $G := (V, E)$ with no loops, where V stands for a set of

vertices or nodes and E a multiset of unordered pairs of vertices, called edges. In G we specify two nodes s and t which characterize the terminal nodes. We define two not necessarily injective maps, f and g, where $f : E \to C$ assigns the edges to the system components and $g : E \to V^2$ assigns each edge to a pair of nodes. Each component $c \in C$ represents a random variable with two states: failed or working. The probability of failure $q_c = 1 - p_c$ is given for each $c \in C$. The system's terminal pair reliability $R_{s,t}$ is the probability that the two specified terminal nodes are connected by at least one path consisting of only edges associated with working components.

In this model we claim the statistical independence of component failures. However, we indicate, that the approach presented in section 3 can be easily adapted to dependent failures [8]. Furthermore, we solely stay with series and parallel reductions in order to have a comparison to approaches where other reduction techniques, such as polygon-to-chain [7] or triangle reductions [13], cannot be applied, e.g. when considering node failures [3,11]. They surely would lead to a notable decrease of computation time and number of subproblems, but our main focus is to convey the idea of getting around the limitation of memory without compromising runtime.

2.1 The Gobien Dotson approach

Let P be a path $P = [e_1, e_2, \ldots, e_r]$ of length r ($r \in \mathbb{N}$) in the network graph G from s to t and $e_i \in E$ for $1 \leq i \leq r$. Following [10], the reliability expression of G can be obtained by recursively applying the factoring theorem for each of the r edges in path P. Starting with the first edge e_1, we define the set of edges which are mapped to the same component as e_1 : $\mathcal{E}_1 = \{e \in E | f(e) = f(e_1)\}$. As a result we have:

$$R_{s,t}(G) = p_1 \cdot R_{s,t}(G * \mathcal{E}_1) + q_1 \cdot R_{s,t}(G - \mathcal{E}_1),$$

where $*/-$ stands for a contraction/deletion of all edges in $\mathcal{E}_1$ and $q_i = 1 - p_i$, $1 \leq i \leq r$ is the failure probability of component $f(e_i)$. Then the term $R_{s,t}(G * \mathcal{E}_1)$ will again be expanded by factoring on edges in $\mathcal{E}_2$ which is analogously defined. Overall it follows that:

$$\begin{aligned}(1)\quad R_{s,t}(G) = {} & q_1 \cdot R_{s,t}(G - \mathcal{E}_1) \\ & + p_1 q_2 \cdot R_{s,t}(G * \mathcal{E}_1 - \mathcal{E}_2) \\ & + \ldots \\ & + p_1 p_2 \cdots p_{r-1} q_r \cdot R_{s,t}(G * \mathcal{E}_1 * \mathcal{E}_2 * \ldots * \mathcal{E}_{r-1} - \mathcal{E}_r) \\ & + \prod_{k=1}^{r} p_k\end{aligned}$$

So we have r subproblems respectively subgraphs deduced from path P. If there exist $i, j \in \{1, 2, \ldots, r\}$, $i \neq j$ but $f(e_i) = f(e_j)$, then $\mathcal{E}_i = \mathcal{E}_j$. Hence, the number of subproblems would decrease by one. Again, for each subproblem this equation can be recursively applied. Thus, for each subproblem we are looking for the topologically shortest path to keep the number of subproblems low. This is done by breadth-first search since all edges have length one. In each subgraph reductions

can be performed if possible. According to [15] all s-t paths correspond to success events (system is in a working state) and all s-t cuts to failure events (system is in a failed state). Suppose $\mathcal{S}$ to be a disjoint exhaustive success collection of success events S_i, $1 \leq i \leq N$ in G. The terminal pair reliability of G is then represented by

$$R_{s,t}(G) = \sum_{i=1}^{N} \mathbb{P}(S_i).$$

The last addend of equation 1 is the probability of the first success event S_1, thus $\mathbb{P}(S_1) = \prod_{k=1}^{r} p_k$. All the other $\mathbb{P}(S_i)$, $2 \leq i \leq N$ are the probabilities of the remaining s-t paths found in the subgraphs. Analogously it holds for the exhaustive failure collection $\mathcal{F} := \{F_i,\ 1 \leq i \leq M\}$

$$R_{s,t}(G) = 1 - \sum_{i=1}^{M} \mathbb{P}(F_i),$$

where the $\mathbb{P}(F_i)$ terms are the probabilities for the s-t cuts. For $u < |\mathcal{S}|$, $v < |\mathcal{F}|$ and $u, v \in \mathbb{N}$ the lower and upper bounds for the reliability are

$$\sum_{i=1}^{u} \mathbb{P}(S_i) \leq R_{s,t}(G) \leq 1 - \sum_{i=1}^{v} \mathbb{P}(F_i).$$

Following this inequation, the lower bound increases everytime a new s-t path has been found respectively the upper bound decreases for every additional s-t cut. It becomes an equation as soon as all s-t cuts and paths are found. We refer to [8] for the application of the described approach to a short example network.

3 The memory efficient Gobien Dotson approach

In this section we will first explain how to keep the memory consumption of this approach as low as possible. Best to our knowledge, by the help of an event queue, the subgraphs are recursively created (see [10]) instead of being stored. Each event contains a sequence of edges after which the original graph is partitioned. Unfortunately, this approach does not incorpoorate any reductions. Additionally, the events contain redundant information by sharing the same sequence of precedent edges. In order to remedy this redundancy and the lack of reductions, we propose the use of a so-called the *delta tree*. It keeps track of all changes made to the original graph due to reductions and partitioning.

Even though memory consumption is kept as low as possible, the limitation is soon reached by large graph sizes due to the exponential growth of this problem. The main idea is to migrate the *delta tree* to hard disk. The data to be written is arranged in a certain way in order to comply with the hard disk's sequential read and write access (see 3.2).

3.1 The delta tree

All the intermediary results of this method can be stored in a recursion tree called the *delta tree*, T_Δ, whose structure is as follows: Starting with the root node, we store all reductions performed on the original input graph herein. In general, each node of the tree stores all consecutively performed reductions on a certain subgraph. The number of child nodes equals the length of the shortest s-t path found at the parent node. The edges connecting the parent node with the child nodes contain the information for partitioning the respective subgraph represented by the parent node. In the course of the algorithm the tree emerges level by level according to breadth first search order. Each leaf of the tree represents a subgraph or task to be proceeded. This subgraph can be reconstructed by tracing back the *delta path*, P_Δ, from leaf to root. Apart from the subgraph, the appropriate edge probability map, *EdgeProbMap* (epm), and the accumulated path/cut terms can be reconstructed with the help of P_Δ. Below, we show how the relevant information can be stored in T_Δ.

Each series or parallel reduction involves two edges, e_1 and e_2, whereas the first one's probability p_{e_1} is readjusted with respect to the performed reduction: $p_{e_1} := p_{e_1} \cdot p_{e_2}$ (for series-) or $p_{e_1} := p_{e_1} + p_{e_2} - p_{e_1} \cdot p_{e_2}$ (for parallel reduction). Edge e_2 will be contracted in case of a series reduction and deleted in case of a parallel reduction. In general, the contraction of an edge e contains the following steps: First delete e, then merge the border nodes of e to one node. In both cases (series- and parallel), edge e_2 is removed from the graph. Edge e_1 remains in the graph and p_{e_1} is captured in the *EdgeProbMap*. The operation $\circ$ is introduced for distinguishing a contraction ($\circ = +$) from a deletion ($\circ = -$). To distinguish e_1 and e_2, any reduction term red comprises a semicolon, its string representation is as follows:

$$red := "e_1; \circ e_2"$$

Any delta tree node n of a graph containing l reductions is represented by:

$$n := "red'_1 red'_2 \dots red'_l"$$

The reductions are separated by an acute accent.

Based on the shortest path of length r, the r subproblems are each derived by edge deletion and contraction operations. All edges which are to be contracted are standing upfront followed by the last edge e_r which is to be deleted (by the $-$ operation). Again those edges can be separated by a semicolon. Suppose we have found a path of length r at a node n in the *delta tree*, then the r *delta tree* edges e_Δ emerging from parent node n are defined as follows:

$$e^1_\Delta := " - e_1" \quad e^2_\Delta := "e_1; -e_2" \quad \dots \quad e^r_\Delta := "e_1; e_2; \dots; e_{r-1}; -e_r"$$

Every *delta path* P_Δ of recursion level m is an alternating sequence of T_Δ-nodes n_i (commencing with root n_0) and T_Δ-edges e^i_Δ, $0 \le i \le m$:

$$P_\Delta := "n_0 > e^0_\Delta > n_1 > e^1_\Delta > \dots > e^{m-1}_\Delta > n_m > e^m_\Delta"$$

3.2 The modified algorithm

In this part we describe the whole modified algorithm which generates an output file *FNext* by taking an input file *FPrev* at each recursion level. After initializing the input graph, the files and all appropriate maps in procedure 1, procedure 2 is invoked for processing the initial graph. There we first check the connectivity of the graph. If the graph is connected, we look for possible reductions in line 12. The changed probabilities due to reductions are updated in *epm* at line 13. Additionally, the performed graph manipulations caused by reductions are captured in a string as described above and again this string is contributed to *line* (line 14). Furthermore, *line* is enriched with the respective subproblems according to the shortest path *sp*. Finally, *line* is written to *FNext*.
Now we define the rules for writing T_Δ to *FNext*. Each line in *FNext* (*linebranch*) represents a branch of T_Δ comprising k leafs. The string representation therefore is defined as follows

$$linebranch := "n_0 > e_\Delta^0 > n_1 > e_\Delta^1 > \ldots > e_\Delta^{m-1} > n_m : e_\Delta^{sub_0}, e_\Delta^{sub_1}, \ldots, e_\Delta^{sub_k}"$$

Hereby the k *delta tree* edges e_Δ^{sub} are separated by a comma and attached by a colon. In procedure 2 the first part of *linebranch* - to the left of the colon - is aggregated by *aggregateLeaf(e)* with the respective subproblems. At the end of the for-loop the completed *linebranch* is written to file *FNext*. After all *linebranches* of the current T_Δ level have been written, the *FNext* file is renamed to *FPrev* and then taken as input for the next recursion in procedure 3. A *delta path* (task P_{Δ_i}) with $i \in \{0, 1, \ldots, k\}$ is derived from *linebranch* and is represented as follows

$$P_{\Delta_i} := "n_0 > e_\Delta^0 > n_1 > e_\Delta^1 > \ldots > e_\Delta^{m-1} > n_m > e_\Delta^{sub_i}"$$

In procedure 3, each P_Δ deduced from *linebranch* is processed by procedure 4 which reads P_Δ and reconstructs the subgraph and the appropriate accumulated terms *acc*. At the end of each level of T_Δ, the current bounds are printed out.

Procedure 1 Main

1: $Init()$; //Initialize $inputGraph, epmInit, FNext, FPrev$
2: $FPrev = computeRel(inputGraph$,new $List(), epmInit, "", FNext)$;
3: $FNext =$ new File;
4: $bfsLevel(FPrev)$;

4 Experimental results

The modified approach was implemented in JAVA and tested on nine example networks on an Intel Xeon 2.97 GHz machine. Each network (nw.) in Table 1 is annotated with its number of edges. The terminal nodes are colored in black. Nw.4-6 are featured with parameter N which stands for the number of horizontal edges in each row and $N - 1$ is the number of vertical edges in each column. In nw.9, N

Procedure 2 computeRel

Input: RBD $Graph$,List acc,EdgeProbMap epm,String $line$,File $FNext$

```
1: bool PorC; //Condition variable for determining path/cut
2: bool b = Graph.findPath();
3: if b == false then
4:    double tmp = computeProduct(acc);
5:    if PorC == true then
6:       Paths = Paths + tmp;
7:    else
8:       Cuts = Cuts + tmp;
9:    end if
10:   return
11: end if
12: SPRed red = Graph.reduce() //Preprocessing: Reduce graph.
13: epm = red.getEdgeProbMap(); //Update edge probabilities
14: line = line + red.getString(); //Extend line with reduced terms
15: List sp = BFS.shortestPath(Graph); //Find shortest path
16: for each e ∈ sp do
17:    line = line + aggregateLeaf(e);
18: end for
19: FNext.write(line);
20: return FNext
```

Procedure 3 bfsLevel

Input: File $FPrev$

```
1: for each linebranch ∈ FPrev do
2:    //proceed each subproblem (path)
3:    for each sub ∈ linebranch do
4:       RBD rbd = readTaskBranch(sub);
5:       FNext = computeRel(rbd, accumlation, epm, line, FNext);
6:    end for
7: end for
8: if FNext.IsEmpty() then
9:    return
10: end if
11: FPrev = FNext;
12: FNext = new File;
13: print "upper bound for Unreliability = 1 − Paths()";
14: print "lower bound for Unreliability = Cuts()";
15: bfsLevel(FPrev);
```

represents the number of squares in the K4-ladder (see [2]). In order to compare the results with related papers we claim that every edge fails with a probability of 0.1 in all networks. Table 2 shows the bounds computed with regard to a relative accuracy of ten percent. The last column $d(T_\Delta)$ indicates the depth or level of the

Procedure 4 readTaskBranch

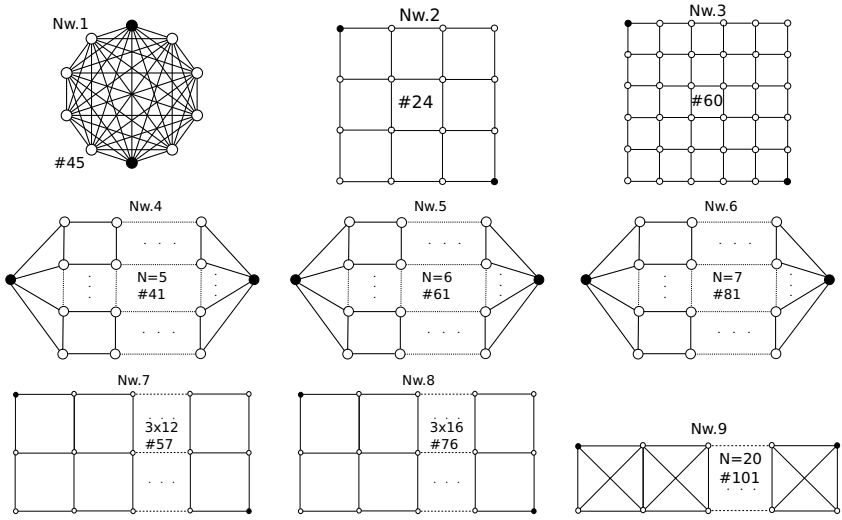
Input: String *sub*

```
String[] deltapath = sub.split(>);
for each i ∈ deltapath do
    processNode(deltapath[i]); //reductions
    i = i + 1;
    processEdge(deltapath[i]); //merge&delete operations
end for
epm.update(); //recreated EdgeProbMap
acc.update(); //recreated accumlation List
return rbd; //reconstructed graph
```

delta tree after which the respective bounds were obtained. The file size, the number of tasks and the average disk IO bandwidth are also listed. It can be observed that apart from the number of components, the runtime highly depends on the structure of the network. Comparing the results of nw.3 and nw.4 in Table 2 their runtimes are roughly the same. It is a wrong conclusion to expect a much faster solution for nw.4 regarding its 1.5 times higher bandwidth. The reason therefore lies in their graph structures: The shortest *s-t* path for nw.3 is twice as long as the one for nw.4 which means that nw.3 generates in general more subproblems in each level than nw.4. This leads to a higher computation time for processing all tasks for each level. Hence, for nw.3 more time must be spent to wait for the data to be written on hard disk leading to a lower bandwidth. Another observation concerning the impact of the graph structure is between nw.6 and nw.9: nw.6 has 20 components less than nw.9 but we needed about four times longer in order to achieve bounds with the same relative accuracy. Though we start with a lower number of subproblems (length of shortest *s-t* path) in nw.6, this number still remains high after several recursion levels for many child nodes in T_Δ leading to a high number of tasks for each level. For nw.6 272 million subproblems are stored in 21.4 GB which means that in average 84 Bytes are needed to encode a task. Comparing the bandwidth of nw.4-6, we notice that the larger the network becomes the lower the average disk bandwidth. The simple reason is that it takes more time to perform graph manipulations on larger graphs and more subproblems evolve due to the increasing length of the shortest *s-t* path leading to a higher latency. For some networks it was not possible to obtain the exact results within two days. Those that we could finish are listed in Table 3. $d_{\max}(T_\Delta)$ represents the maximal depth (or the total amount of levels) of the *delta tree*. Note that the depth of the tree is limited by the number of edges of the original input graph since the number of edges decreases by at least one, in each recursion. $d_w(T_\Delta)$ is the broadest level of the tree. The file size of this level also indicates the maximum disk space needed for the whole computation. We can make the following observation by comparing the two tables: For large networks we only need a little fraction of the total time and also of the maximum required disk space in order to reach satisfying bounds. For example, for nw.3 the fraction of disk space required is 0.071 percent and the time spent is only 0.018 percent of the overall time. Similarly, this can be observed for nw.5.

We remark that for even smaller failure probability values - in the magnitude of 10^{-5} for highly reliable systems - the bounds would be obtained in an even shorter amount of time requiring less hard disk space.
To compare the implementation based on the delta tree and disk storage with an older implementation of the method [15] extended with series and parallel reductions, we have added another time column. For nw.5-9 the memory of 2GB was exhausted before the respective bounds could be achieved. For all the other networks we can observe that the runtime is shorter for the small sized nw.1 and nw.2 but it takes significantly more time for the larger networks three and four. This is due to the hold of all subgraphs and perfomed reductions in memory by method [15]: For small sized problems less memory is required to store each generated subgraph. Furthermore less subproblems evolve. As soon as the problem reaches a certain size, more storage is needed for each subgraph and also more subproblems are to be expected. Consequently the prohibitively rapid growth of memory demand leads to a negative impact on runtime.

Table 1
Benchmark Networks 1-9



5 Conclusion

In this work we stressed the problem of high memory consumption for the Gobien Dotson algorithm. Due to the exponential nature of the terminal-pair reliability problem, the demand for memory grows unacceptably with the size of the networks to be assessed. This imposes a limiting factor for reaching good reliability bounds since the computation must be interrupted because of memory shortage. Hence, we suggested to migrate the memory content to hard disk. The *delta tree* was encoded in a certain way to comply with the hard disk's sequential read and write behavior. This method even allows interruptions since the files created until the point of interruption can be reused to continue the computation at a later time. We found

Table 2
Bounds (relative accuracy=0.1)

Nw	**lb**	**ub**	**time**	**time[15]**	**file(MB)**	**#tasks**	**øbw**$\left(\frac{MB}{s}\right)$	d
1	$1.99 \cdot 10^{-9}$	$2.05 \cdot 10^{-9}$	0.68 s	0.41 s	0.14	1464	0.64	11
2	$2.47 \cdot 10^{-2}$	$2.50 \cdot 10^{-2}$	0.26 s	0.20 s	0.01	282	0.10	5
3	$2.42 \cdot 10^{-2}$	$2.47 \cdot 10^{-2}$	8.00 s	26.17 s	5.55	121,622	0.78	7
4	$9.12 \cdot 10^{-5}$	$9.14 \cdot 10^{-5}$	7.68 s	11.53 s	7.26	97,036	1.16	10
5	$1.32 \cdot 10^{-5}$	$1.36 \cdot 10^{-5}$	180 s	-	183.78	2,631,226	0.60	11
6	$1.88 \cdot 10^{-6}$	$1.91 \cdot 10^{-6}$	7.73 h	-	21,924.28	272,012,633	0.40	14
7	$3.81 \cdot 10^{-2}$	$3.85 \cdot 10^{-2}$	49.16 s	-	54.10	697,592	0.75	8
8	$4.36 \cdot 10^{-2}$	$4.38 \cdot 10^{-2}$	1.39 h	-	4,899.35	53,184,683	0.56	8
9	$4.34 \cdot 10^{-3}$	$4.41 \cdot 10^{-3}$	1.91 h	-	9,011.64	50,958,418	0.78	10

Table 3
Exact results

Nw	**Unreliability**	**time**	**time[15]**	$d_{\max}$	d_w	**file(MB)**	**#tasks**	**øbw**$\left(\frac{MB}{s}\right)$
1	$2.00 \cdot 10^{-9}$	9.79 s	6.26 s	36	20	1.65	12,574	1.99
2	$2.49 \cdot 10^{-2}$	0.35 s	0.17 s	9	5	0.01	282	0.10
3	$2.43 \cdot 10^{-2}$	12.45 h	-	25	16	20,429.39	62,358,421	2.43
4	$9.13 \cdot 10^{-5}$	41.75 s	47.52 s	20	12	14.52	138,814	1.65
5	$1.33 \cdot 10^{-5}$	39.55 h	-	30	20	30,613.73	203,132,939	1.43
7	$3.83 \cdot 10^{-2}$	0.61 h	-	22	12	521.65	4,135,084	1.25

that only a small fraction of the complete runtime and the maximum required space is needed to obtain reasonable and accurate bounds. It must be said that this fact depends very much on the probability of failure of the system components and is pertinent for highly reliable systems. Most of the additional time only contributes to minor improvements of the bounds.

One may assume that by migrating the memory content to hard disk, afterwards the hard disk itself might be a bottleneck. However, by having a look at the measured bandwidth values, this is definitely not the case. On the contrary, the maximal average disk bandwidth of merely 2.43 MB/s shows that there is space for exploiting even further the writing speed of nowadays hard disks (of around 150 MB/s). In this context, we intend to parallelize the sequential algorithm and take advantage of the property that all tasks P_Δ are independent.

Acknowledgement

We are very thankful to A. Bode from TU München for supporting this work. Our thanks are also dedicated to M. Siegle from Universität der Bundeswehr München for many insightful discussions. We also thank the three anonymous reviewers for their helpful and detailed comments.

References

[1] A.Satyanarayana and M.K.Chang. Network reliability and the factoring theorem. In *Networks, vol.13, no.1, 107-120*, 1983.

[2] C.Tanguy. Asymptotic mean time to failure and higher moments for large, recursive networks. In *CoRR*, 2008.

[3] D.Torrieri. Calculation of node-pair reliability in large networks with unreliable nodes. In *IEEE Trans. Reliability, vol.43, no.3, 375-377*, 1994.

[4] F.M.Yeh, S.K.Lu, and S.Y.Kuo. Determining terminal-pair reliability based on Edge Expansion Diagrams using OBDD. In *IEEE Trans. Reliability, vol.48, no.3, 234-246*, 1999.

[5] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for Binary Decision Diagrams. In *SIAM Journal on Computing, vol.39, no.5, 710-713*, 1990.

[6] J.M.Wilson. An improved minimizing algorithm for sum of disjoint products. In *IEEE Trans. Reliability, vol.39, no.1, 42-45*, 1990.

[7] K.Wood. A factoring algorithm using polygon-to-chain reductions for computing k-terminal network reliability. In *Networks, vol.15, no.2, 173-190*, 1985.

[8] M.Lê and M.Walter. Bounds for Two-Terminal Network Reliability with Dependent Basic Events. In *Lecture Notes in Computer Science, vol.7201/2012, 31-45*, 2012.

[9] M.O.Ball. Computational complexity of network reliability analysis: An overview. In *IEEE Trans. Reliability, vol.35, no.3, 230-239*, 1986.

[10] N.Deo and M.Medidi. Parallel algorithms for terminal pair reliability. In *IEEE Trans. Reliability, vol.41, no.2, 201-209*, 1992.

[11] O.R.Theologou and J.G.Carlier. Factoring and reductions for networks with imperfect vertices. In *IEEE Trans. Reliability, vol.40, no.2, 210-217*, 1991.

[12] R.E.Barlow and F.Proshan. *Mathematical Theory of Reliability.* John Wiley & Sons, 1965.

[13] S.J.Hsu and M.C.Yuang. Efficient computation of terminal-pair reliability using triangle reduction in network management. In *ICC on Communications, vol.1, 281-285*, 1998.

[14] S.Soh and S.Rai. Experimental results on preprocessing of path/cut terms in sum of disjoint products technique. In *IEEE Trans. Reliability, vol.42, no.1, 24-33*, 1993.

[15] W.P.Dotson and J.Gobien. A new analysis technique for probabilistic graphs. In *IEEE Trans. Circuit & Systems, vol.26, no.10, 855-865*, 1979.

[16] Y.B.Yoo and N.Deo. A comparison of algorithms for terminal pair reliability. In *IEEE Trans. Reliability, vol.37, no.2, 210-215*, 1988.

[17] Y.G.Chen and M.C.Yuang. A cut-based method for terminal-pair reliability. In *IEEE Trans. Reliability, vol.45, no.3, 413-416*, 1996.