Fundamenta Informaticae 33 (1998) 1–13
IOS Press

Models For Dependable Computation with Multiple Inputs and Some Hardness Results*

Yongge Wang
Department of Combinatorics and Optimization
University of Waterloo, ON, N2L 3G1, Canada
ygwang@cacr.math.uwaterloo.ca

Yvo Desmedt
Department of Computer Science
Florida State University, Tallahassee, FL 32306-4530, USA
desmedt@cs.fsu.edu and
Information Security Group
Royal Holloway – University of London

Mike Burmester
Information Security Group
Royal Holloway – University of London
Egham, Surrey TW20 OEX, UK
m.burmester@rhbnc.ac.uk

Abstract. We consider the problem of dependable computation with multiple inputs. The goal is to study when redundancy can help to achieve survivability and when it cannot. We use AND/OR graphs to model fault tolerant computations with multiple inputs. While there is a polynomial time algorithm for finding vertex disjoint paths in networks, we will show that the equivalent problem in computation systems with multiple inputs is **NP**-hard. Our main results are as follows. (1) We present a general model for fault tolerant computation systems with multiple inputs: AND/OR graphs. (2) We show that it is **NP**-hard to find two vertex disjoint solution graphs in an AND/OR graph. It follows that in the general case redundancy cannot help to achieve survivability, assuming **P**$\neq$**NP**.

*Research supported by DARPA F30602-97-1-0205. Part of this work was done while the authors were at the University of Wisconsin–Milwaukee.

Keywords: Dependable computation, complexity theory, **NP**-hardness.

1. Introduction

Redundancy has been utilized to achieve reliability, for example to achieve fault tolerant computation and to achieve reliable communication in networks (see e.g., [1, 3, 4]). The goal in this paper is to study when redundancy can help to achieve survivability and when it cannot. The techniques to achieve reliable networks (see e.g., [1, 3, 4]) immediately extend to computations solely based on one input functions in which redundant hardware or software (servers) are used to compute intermediate and end results. However, from this result we cannot conclude that the techniques to achieve redundant computation will extend under similar circumstances to the general case. Indeed, a computation does not need to be based on one input functions. Many computation systems require multiple inputs. For example, an intelligent agent in a mobile code program (see e.g., [9, 16, 19]) must meet other agents to carry out the computation. Similarly, a financial institution may need both the input of its computer system and its communication system to function properly. In general large-scale infrastructures (telecommunications networks, power plants, control systems, etc) have multiple inputs. For a discussion on dependable infrastructure systems with multiple inputs, the reader may consult the 1997 interim report of the National Academy of Sciences [9].

In this paper we focus on the following problem: does redundancy in the general case help to achieve dependability (and survivability) or does it not? A computation (or a system) is dependable (see, e.g., [12, 13]) if it can tolerate both accidental and malicious (Byzantine) faults. We consider a special type of computation systems which can be modeled by AND/OR graphs (defined below). While there are polynomial time algorithms for finding vertex disjoint paths in networks, we will show that the equivalent problem in computation systems with multiple inputs is **NP**-hard. It follows that the use of redundancy for dependable computation will generally fail, assuming $\mathbf{P} \neq \mathbf{NP}$.

The organization of this paper is as follows. In Section 2 we discuss the background and use the AND/OR graphs to model fault tolerant computations with multiple inputs. In Section 3 we prove our main result, that is, that the problem of finding two vertex disjoint solution graphs in an AND/OR graph is **NP**-hard.

2. Background and models

Faults in the context of reliability occur in a probabilistic way and are, at their origin, independent of each other and of the overall state of the system. A typical such fault in a circuit occurs when the output of a gate is independent of its input, for example when the gate is stuck at 0, or when its output is a random string (noise). The impact of such faults can be controlled by using redundancy. Malicious (or Byzantine) faults are controlled by an adversary according to some plan which may exploit the possible weakness of the system. The adversary has at least as much power and knowledge about the structure of the system as the processors of the system have, and possibly more. For example the adversary may know the structure of the system whereas the processors may not. Such an adversary might try to use this information in an attempt to force the system to fail.

Dependability (see, e.g., [12, 13]) is a basic requirement for survivability. It stipulates that the system should survive in the presence of a malicious adversary.

Models

We can model some redundant computation systems by a directed graph (or multi-graph) in which the vertices are the processors and the edges are the links. Incoming edges of a vertex correspond to the inputs of the processor and outgoing edges to its outputs. The graph has at least one input (source) vertex and at least one output (sink) vertex. This is a high level model and is appropriate for many applications. However it is inadequate if processors need different types of inputs, e.g., from different sensors. Also subroutines often have more than one input.

2.1. A model based on directed multi-graphs with colored edges

We can model a redundant computation system with multiple inputs by a directed multi-graph with colored edges. The different colors indicate the different types of inputs. The graph must have at least one input vertex and one output vertex.

There are several possible applications for this model. For example, subroutines whose inputs have the same color need only use one input (when there are no faults). If the colors are different then the processor must use one input for each of the input colors to carry out its computation (or whatever it is supposed to do).

Definition 2.1. A directed multi-graph with colored edges $G(V, INPUT, \mathit{output}; E)$ is a directed graph with a set of vertices V, a set $INPUT \subset V$ of input vertices, one output vertex, and a set of colored directed edges E. The input vertices have no incoming edges and the output vertex has no outgoing edges.

2.2. AND/OR graphs

AND/OR graphs have been used to model problem solving processes in artificial intelligence (see e.g., Nilsson [14]). In this paper, we will use AND/OR graphs to model fault tolerant computations with multiple inputs. An *AND/OR graph* is a directed graph with two types of vertices, labeled $\wedge$-*vertices* and $\vee$-*vertices*. The graph must have at least one input (source) vertex and one output (sink) vertex. The output vertex may be regarded as an $\vee$-vertex (without loss of generality). More specifically, we have the following definition.

Definition 2.2. An AND/OR graph $G(V_\wedge, V_\vee, INPUT, \mathit{output}; E)$ is a directed graph with a set $V_\wedge$ of $\wedge$-vertices, a set $V_\vee$ of $\vee$-vertices, a set $INPUT$ of input vertices, an output vertex $\mathit{output} \in V_\vee$, and a set of directed edges E. The vertices with no incoming edges are input vertices and the vertex with no outgoing edges is the output vertex.

It should be noted that our definition of AND/OR graphs is different from the standard definitions in artificial intelligence (see e.g., [14]), in that the directions of the edges are opposite. The reason is that we want to use the AND/OR graphs to model redundant computation systems.

It can be shown that this AND/OR graph model is equivalent to the directed graph model with multi-colored edges in the sense that there is a polynomial time reduction from each model to the other (which will preserve the semantics of the AND/OR graph and the directed graph with multi-colored edges, where semantics means the function that the model computes and the redundancy of each unit). However an AND/OR graph seems to be a more powerful mathematical tool for the study of redundant computation systems.

The application given in Section 2.1 can also be used for this model. In this case, for processors which need all their inputs in order to operate could be represented by $\wedge$-vertices, whereas processors which can choose (using some kind of voting procedure) one of their "redundant" inputs could be represented by $\vee$-vertices. In the replicated agent computation with voting (see Schneider [19]), a **meet** operation may be considered as an $\wedge$-vertex and the operation of choosing one agent from its replicas may be considered as an $\vee$-vertex.

2.3. Solution graphs

As mentioned earlier, redundancy plays a key role in achieving reliability. It is used in reliable communications (with error-correcting codes), in network reliability and in fault tolerant computation. Assume that we use the AND/OR graph to model a fault tolerant computation. Then, information (for example, mobile codes) must flow from the input vertices to the output vertex, and a valid computation can be described by a *solution graph* (the exact definition will be given below). However, if some insider vertices are faulty or even malicious, then the output vertex cannot be certain that the result will be correct. If we assume that at maximum k vertices are malicious, then the theory of fault tolerant computation (see e.g., Beimel and Franklin [1], Dolev [3], and Dolev et al. [4]) tells us that, if there are $2k+1$ vertex disjoint paths of information flow from the inputs to the output then the *output* will always succeed in getting the correct result by taking a majority vote on the results computed via $2k+1$ vertex disjoint solution graphs from the inputs, provided that the *output* knows the layout of the graph. It follows that in order to achieve dependable computation with redundancy, it is necessary to find an appropriate number of vertex disjoint solution graphs in a given AND/OR graph.

A path in the traditional sense is not useful in our contexts, since for an $\wedge$-vertex we need all the inputs to carry out the computation. We therefore give a formal definition of a solution graph in an AND/OR graph.

Definition 2.3. Let $G(V_\wedge, V_\vee, INPUT, output; E)$ be an AND/OR graph. A solution graph $P = (V_P, E_P)$ is a minimum subgraph of G satisfying the following conditions.

1. $output \in V_P$.
2. For each $\wedge$-vertex $v \in V_P$, all incoming edges of v in E belong to E_P.
3. For each $\vee$-vertex $v \in V_P$, there is exactly one incoming edge of v in E_P.

4. There is a sequence of vertices $v_1, \ldots, v_n \in V_P$ such that $v_1 \in INPUT, v_n = output$, and $(v_i \rightarrow v_{i+1}) \in E_P$ for each $i < n$.

Moreover, two solution graphs P_1 and P_2 are vertex disjoint if $(V_{P_1} \cap V_{P_2}) \subseteq (INPUT \cup \{output\})$.

Note that every vertex in a solution graph has some outgoing edges due to the minimum property of a solution graph. Since an AND/OR graph without $\wedge$-vertices may be considered as a normal digraph, the solution graphs in such AND/OR graphs are exactly the same as the standard paths in digraphs. It is also easy to see that there is only one solution graph in an AND/OR graph which has only one $\vee$-vertex (that is, the output vertex) and in which the *output* vertex has only one incoming edge.

If we would have an efficient way to find the vertex disjoint solution graphs in a given AND/OR graph then redundancy would help us to achieve dependable computation with multiple inputs as follows: carry out the computation through all vertex disjoint solution graphs and let the output vertex decide whether the result is correct or not by majority vote. However, our results in this paper show that it is **NP**-hard to compute the vertex disjoint solution graphs in AND/OR graphs.

2.4. Related works

Finding disjoint paths in a graph is a classical problem in graph theory. A fundamental characterization was found in 1927 by K. Menger: the maximum number of pairwise disjoint paths between a given "source" and a given "sink" in a graph is equal to the minimum size of a "cut" separating source and sink. In 1956, Ford and Fulkerson [5] published a direct labeling method for the more general problem of finding the maximum flows in a graph. This result implies the *max-flow min-cut theorem*, which has been used to find vertex disjoint paths between **one** source and **one** sink in a graph. Lynch proved in 1975 that the following problem is **NP**-complete: given a planar graph and pairs $(r_1, s_1), \ldots, (r_k, s_k)$ of vertices, find k pairwise vertex disjoint paths connecting r_i and s_i for $i = 1, \ldots, k$ respectively. Robertson and Seymour [18] have shown that if the number k is fixed (that is, k is not a part of the input), then there is a polynomial time algorithm which solves this problem (when the graph is planar, here is a linear time algorithm, see Wagner and Weihe [20]). In the case of digraphs, Fortune, Hopcroft and Wyllie [6] have shown that this problem is **NP**-complete even for $k = 2$ pairs of vertices.

There have been also many results in disjoint path and tree methods for the design of *very large-scale integrated (VLSI) circuits* (see, e.g., Korte, Lovasz, Prömel, and Schrijver [10]). Many problems in VLSI circuits design reduce to finding vertex disjoint trees (Steiner trees) in a graph, each spanning a given set of vertices. If each such set consists of just two vertices, we have a vertex disjoint paths problem. It has been shown that the general problem of finding vertex disjoint Steiner trees in a graph is **NP**-complete. Note that our results, though different, are related to these vertex disjoint paths problems.

Achieving processor cooperation in the presence of faults is a major problem in distributed systems. Popular paradigms such as Byzantine agreement have been studied extensively. Dolev

[3] (see also, Dolev et al. [4]) showed that Byzantine agreement is achievable only if the number of faulty processors in the system is less than one-half of the connectivity of the system's network. Hadzilacos [8] has shown that even in the absence of malicious failures, $k+1$ connectivity is required to achieve agreement in the presence of k faulty processors. All these results assume that the processors have one type of input, while our approach is more general in this respect.

3. Finding vertex disjoint solution graphs in an AND/OR graph is NP-hard

3.1. A brute-force algorithm for finding vertex disjoint solution graphs

We first present a general algorithm for finding k vertex disjoint solution graphs in an AND/OR graph, and show that for a certain subclass of AND/OR graphs, this algorithm is efficient.

Definition 3.1. For an AND/OR graph $G(V_\wedge, V_\vee, INPUT, output; E)$, a directed line l from v_1 to v_m in G is a sequence of vertices $v_1, \dots, v_m$ such that $(v_i \to v_{i+1}) \in E$ for all $i < m$. For a directed line $l : (v_1, \dots, v_m)$ in G, we have the following definitions.

$$\#_\vee(l) = |\{v_i : i < m, v_i \in V_\vee\}|$$

$$\#_\wedge(l) = |\{v_i : i > 1, v_i \in V_\wedge\}|$$

$\#_\vee(G) = \max\{\#_\vee(l) : l \text{ is from an input vertex to the output vertex in } G\}$,
and
$\#_\wedge(G) = \max\{\#_\wedge(l) : l \text{ is from an input vertex to the output vertex in } G\}$.
We also define $\deg_{\text{in}}(G)$ to be the maximum of the indegrees of all vertices (except the *output*) in G.

Given an AND/OR graph G and an edge $v \to output$, the following algorithm outputs the set of all solution graphs in G which pass through the edge $v \to output$.

Algorithm I

Input: An AND/OR graph G and an edge $v \to output$.
Output: The set $PATH_v$ of all solution graphs in G which pass through the edge $v \to output$.

1. Let $V_1 = \{output, v\}$, $E_1 = \{v \to output\}$, and $V_i = E_i = \emptyset$ for $i > 1$.
2. Set $i = 1$.
3. For each $u \in V_i \setminus INPUT$ such that u has no incoming edges in E_i, we distinguish the following two cases.

   Case 1. $u \in V_\wedge$. If $u_1 \to u, \dots, u_s \to u$ are the incoming edges of u in E, then let $V_i = V_i \cup \{u_i : i = 1, \dots, s\}$ and $E_i = E_i \cup \{u_i \to u : i = 1, \dots, s\}$.

   Case 2. $u \in V_\vee$. If $u_1 \to u, \dots, u_s \to u$ are the incoming edges of u in E, then let $V_i = V_i \cup \{u_1\}$, $E_i = E_i \cup \{u_1 \to u\}$, $V_{i_0+j} = V_i \cup \{u_{j+1}\}$, and $E_{i_0+j} = E_i \cup \{u_{j+1} \to u\}$, where $j = 1, \dots, s-1$ and $i_0 = \max\{t : V_t \neq \emptyset\}$.

4. If there exists a positive integer i_0 and $v \in V_{i_0} \setminus INPUT$ such that v has no incoming edges in E_{i_0}, then let $i = i_0$ and go to Step 3.
5. Let $PATH_v = \{(V_i, E_i) : V_i \neq \emptyset$ and (V_i, E_i) is a solution graph$\}$. □

The following lemma gives an upper bound on the size of $PATH_v$.

Lemma 3.1. $|PATH_v| \leq \deg_{\text{in}}(G)^{\#_\vee(G)\cdot \deg_{\text{in}}(G)^{\#_\wedge(G)}}$.

Proof:
The search process in **Algorithm I** can be considered as a search on a tree (the tree can be constructed while the search is going on) with depth at most $\#_\vee(G) + \#_\wedge(G)$. Each node on the tree has at most $\deg_{\text{in}}(G)$ sons. Some nodes are marked as $\vee$-nodes if they correspond to the $\vee$-vertices of the AND/OR graph, and others marked as $\wedge$-nodes. If we consider this tree as an AND/OR graph with the root node as the *output* vertex and with the leaves as the *input* vertices, then it is clear that $|PATH_v|$ is less than or equal to the number of solution graphs in this AND/OR tree corresponding to the original AND/OR graph. Whence it suffices to show that there are at most $\deg_{\text{in}}(G)^{\#_\vee(G)\cdot \deg_{\text{in}}(G)^{\#_\wedge(G)}}$ solution graphs in such kind of AND/OR tree. We define the root of the tree to be the node of *depth* 0, and inductively we define the depth of a node to be greater by one than the depth of its parent. An AND/OR tree is called *regular* if it satisfies the following properties:

1. The depth of the tree is $\#_\wedge(G) + \#_\vee(G)$.
2. Each node has $\deg_{\text{in}}(G)$ sons.
3. All nodes with depth less than or equal to $\#_\wedge(G)$ are $\wedge$-nodes.
4. All nodes with depth greater than $\#_\wedge(G)$ are $\vee$-nodes.

It is straightforward to see that a regular AND/OR tree has exactly $\deg_{\text{in}}(G)^{\#_\vee(G)\cdot \deg_{\text{in}}(G)^{\#_\wedge(G)}}$ solution graphs. Whence it suffices to show that a regular AND/OR tree has the largest number of solution graphs. This can be proved by induction on the depth of the AND/OR tree. That is, we can inductively move all $\wedge$-vertices from high depth to low depth. The following Claim guarantees that the "movement" will not decrease the number of solution graphs. For convenience, if v is a node on an AND/OR tree G_t we let *sub*(v) denote the subtree of G_t rooted at v.

Claim 3.1. Let G_t be an AND/OR tree and v be an $\vee$-vertex on G_t with the following properties:

1. The sons of v are: $u_1, \ldots, u_n$.
2. The sons of u_i $(i \leq n)$ are: $u_{i,1}, \ldots, u_{i,m}$.
3. For each $i \leq n$ and $j \leq m$, there are $p_{i,j} \geq 1$ solution graphs in *sub*$(u_{i,j})$ of G_t.

Furthermore let G'_t be an AND/OR tree constructed from G_t and v by replacing the subtree *sub*(v) on G_t with a subtree *sub*(v') with the following properties:

1. v' is an $\wedge$-vertex.

2. The sons of v' are the $\vee$-vertices: $u'_1, \ldots, u'_m$, where u'_i $(i \leq m)$ is the father of the subtrees: $sub(u_{1,i}), \ldots, sub(u_{n,i})$.

Then the number of solution graphs in G_t is less than or equal to the number of solution graphs in G'_t.

Proof of Claim: It is clear that there are exactly $\prod_j \sum_i p_{i,j}$ solution graphs in $sub(v')$ of G'_t, and it is straightforward to check that there are at most $\prod_j \sum_i p_{i,j}$ solution graphs in $sub(v)$ of G_t (the details are omitted here). The Claim is thus proved.

This completes the proof of the lemma. □

Let $PATH = \cup_{(v \to output) \in E} PATH_v$. Then it follows that, for an AND/OR graph G with $\deg_{\text{in}}(G) = 2$, $\#_\wedge(G) = k_0$, $\#_\vee(G) = \lfloor \log n \rfloor$, and $\deg_{\text{in}}(output) = m$, we have

$$|PATH| = \sum_{(v \to output) \in E} |PATH_v| \leq m(2^{2^{k_0} \lfloor \log n \rfloor}) \leq mn^{2^{k_0}}.$$

The following algorithm will tell us whether an AND/OR graph G has k vertex disjoint solution graphs.

Algorithm II

Input: An AND/OR graph G and a number k.
Output: k vertex disjoint solution graphs in G, if such solution graphs exist.

Let $v_1 \to output, \ldots, v_m \to output$ be an enumeration of all incoming edges of *output*. For each k-tuple $v_{i_1} \to output, \ldots, v_{i_k} \to output$ from the incoming edges of *output*, check whether there exist vertex disjoint solution graphs $P_1 \in PATH_{v_{i_1}}, \ldots, P_k \in PATH_{v_{i_k}}$, where $PATH_{v_{i_j}}$ can be computed using the Algorithm I. If the search succeeds, output the k vertex disjoint solution graphs. □

Since $output \in V_\vee$, it is easy to check that the Algorithm II uses at most

$$\binom{m}{k} \prod_{v \to output} |PATH_v| \tag{1}$$

steps. Hence for an AND/OR graph G and a number k, if the value in (1) is smaller than a given polynomial, then there is a polynomial time algorithm to compute k vertex disjoint solution graphs in G, if such solution graphs exist.

Lemma 3.2. *Let G be an AND/OR graph such that $\deg_{\text{in}}(G) = 2$, $\#_\wedge(G) = k_0$, and $\#_\vee(G) = \lfloor c \log n \rfloor$ for some constants k_0 and c. Then, for a constant k, there is a polynomial time algorithm to compute k vertex disjoint solution graphs in G, if such solution graphs exist.*

After having described the above exhaustive search algorithm, one may wonder whether there are more efficient algorithms for finding vertex disjoint solution graphs in AND/OR graphs. In the next section, we will show that this problem is **NP**-hard.

3.2. NP-completeness

We have shown in Section 3.1 that, if both $\binom{m}{k}$ and $\prod_{(v\to output)\in E}|PATH_v|$ are bounded by polynomials then there is a polynomial time algorithm to find k vertex disjoint solution graphs in the AND/OR graph G, if such solution graphs exist. The result in this section will show that if we do not impose any of these conditions, then the problem is **NP**-complete. We first define two problems on which at least one of the two conditions fails.

AOG1$\vee$ (AND/OR Graphs with 1 $\vee$-vertex).
Instance: A number k and an AND/OR graph G with $\#_\vee(G) = 0$.
Question: Do there exist k vertex disjoint solution graphs in G?

AOG2D (AND/OR Graphs with 2 Disjoint solution graphs).
Instance: An AND/OR graph G.
Question: Do there exist 2 vertex disjoint solution graphs in G?

Note that $\prod_{(v\to output)\in E}|PATH_v| = 1$ for AOG1$\vee$ and $\binom{m}{k}$ is a polynomial for AOG2D. The following two theorems show that both AOG1$\vee$ and AOG2D are **NP**-complete.

Theorem 3.1. *AOG1$\vee$ is* **NP***-complete.*

Proof:
It is clear that AOG1$\vee \in$ **NP**. Whence it suffices to reduce the **NP**-complete problem CLIQUE to AOG1$\vee$. The CLIQUE problem is defined as follows.
Instance: A graph G and a number k.
Question: Does there exist a clique of size k in G? (A clique is a complete subgraph of G.)

The input $G = (V_G, E_G)$, to CLIQUE, consists of a set of vertices $V_G = \{v_1, \ldots, v_n\}$ and a set of edges E_G. In the following we construct an AND/OR graph $f(G) = MG(V_\wedge, V_\vee, INPUT, output; E)$ with $\#_\vee(f(G)) = 0$ (the input to AOG1$\vee$) such that there is a clique of size k in G if and only if there are k vertex disjoint solution graphs in $f(G)$.

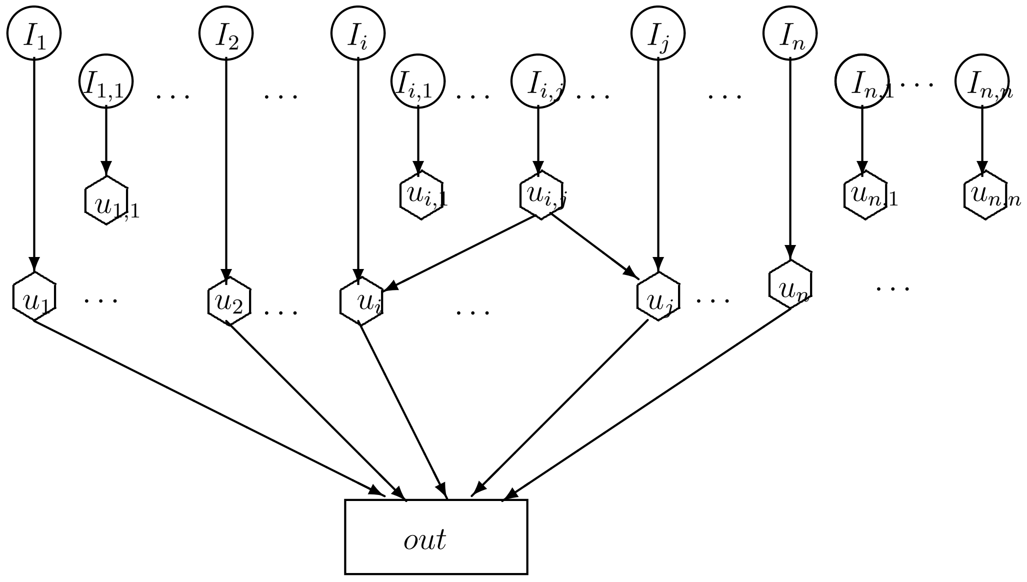
Let $INPUT = \{I_i, I_{i,j} : i, j = 1, \ldots n\}$, $V_\vee = \{output\}$, $V_\wedge = INPUT \cup \{u_{i,j} : i, j = 1, \ldots n\} \cup \{u_i : i = 1, \ldots, n\}$, and E be the set of the following edges.

1. For each $i = 1, \ldots, n$, there is an edge $I_i \to u_i$.
2. For each pair $i, j = 1, \ldots, n$, there is an edge $I_{i,j} \to u_{i,j}$.
3. For each pair $i, j = 1, \ldots, n$, such that the unordered pair $(v_i, v_j) \notin E_G$, there are four edges $u_{i,j} \to u_i$, $u_{i,j} \to u_j$, $u_{j,i} \to u_i$, and $u_{j,i} \to u_j$.
4. For each i, there is an edge $u_i \to output$.

Figure 1 shows the structure of the AND/OR graph $f(G)$.

It is clear that two solution graphs P_1 and P_2 in $f(G)$ which go through u_i and u_j respectively are vertex disjoint if and only if there is an edge (v_i, v_j) in E_G. Hence there is a clique of size k in G if and only if there are k vertex disjoint solution graphs in $f(G)$. $\square$

Figure 1. The AND/OR graph $f(G)$



In order to prove that AOG2D is **NP**-complete, we first define the **NP**-complete problem $3SAT$ as follows. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of *variables.* A *literal* is either a variable x_i or its *complement* $\bar{x}_i$. Thus the set of literals is $L = \{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. A *clause* C is a 3-element subset of L. We are given a set of clauses $C_1, C_2, \ldots, C_m$, each of which consists of 3 literals. The question is whether the set of variables can be assigned values T (true) or F (false), so that each clause contains at least one literal with a T value. A clause is satisfied under an assignment if the clause contains at least one literal with a T value. The concise statement of $3SAT$ is, therefore, the following:

Instance: A set of clauses.
Question: Is there an assignment of the literals such that all the clauses are satisfied.

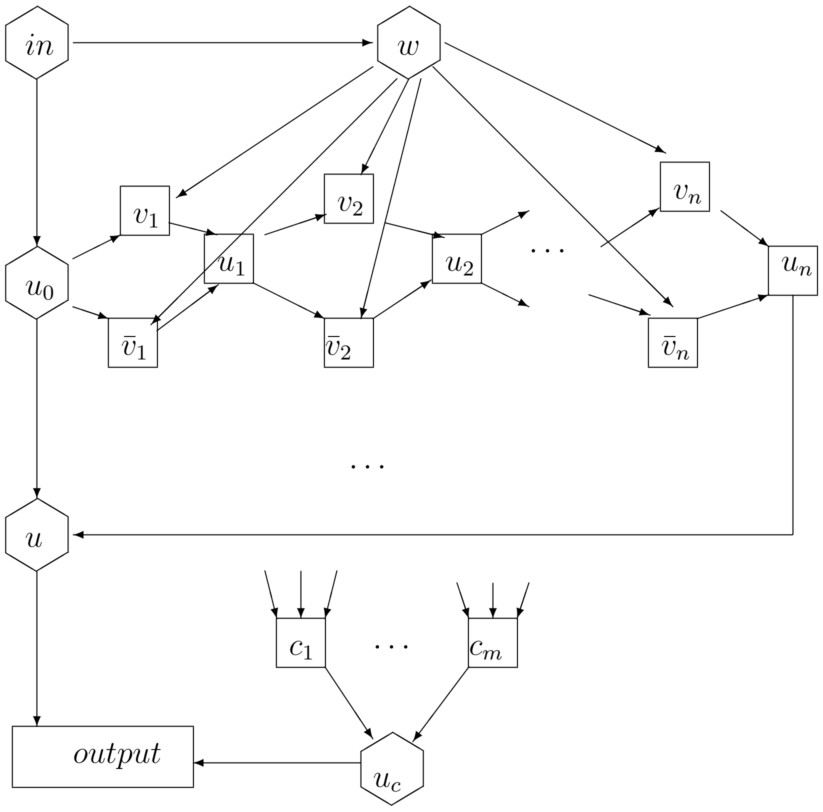
Theorem 3.2. *AOG2D is* **NP**-*complete.*

Proof:
It is clear that AOG2D $\in$ **NP**. Whence it suffices to reduce the **NP**-complete problem $3SAT$ to AOG2D.

The input C, to $3SAT$, consists of clauses $C_1, C_2, \ldots, C_m$, each a 3-element subset of the set of literals $L = \{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. In the following we construct an AND/OR graph $f(C)$ (the input to AOG2D) such that $f(C)$ has two vertex disjoint solution graphs if and only if C is satisfiable.

For each variable x_i we construct two $\vee$-vertices v_i and $\bar{v}_i$ and one $\vee$-vertex u_i, as shown in Figure 2. For the reason of convenience, we use hexagons to denote $\wedge$-vertices and rectangles to denote $\vee$-vertices. There is an input vertex *in* which is connected by two edges to u_0 and w respectively. u_0 is connected by three edges to the vertices v_1, $\bar{v}_1$ and u respectively, and w is connected by $2n$ edges to $v_1, \bar{v}_1, \ldots, v_n, \bar{v}_n$. The vertices for variables are connected in series: for $i < n$, both v_i and $\bar{v}_i$ are connected by edges to u_i, and u_i is connected by edges to v_{i+1} and to $\bar{v}_{i+1}$. u_n is connected by an edge to the $\wedge$-vertex u which is again connected to the output vertex. In addition, there are $\vee$-vertices $c_1, c_2, \ldots, c_m$ and an edge from each to the $\wedge$-vertex u_c. For each occurrence of x_i ($\bar{x}_i$), there is an edge from v_i ($\bar{v}_i$) to the vertex c_j, the clause in which it occurs. Lastly, there is one edge from u_c to the output vertex *output*.

Figure 2. The AND/OR graph $f(C)$



It is easy to see that if there are two vertex disjoint solution graphs P_1 and P_2 in $f(C)$, then we can assume that the first solution graph P_1 goes from *in* through u_0 and u to *output*, and that the second solution graph P_2 goes from *in* through w and u_c to *output*. Also it is clear that P_1 must use u_n and v_i or $\bar{v}_i$ for each $i \leq n$, but not both.

For the second solution graph P_2, there is exactly one edge entering c_u ($u \leq m$) since u_c is an $\wedge$-vertex. If this edge comes from v_i for $i \leq n$, then P_1 must use $\bar{v}_i$.

Thus if the answer to $f(C)$ with respect to AOG2D is positive, then we can use the paths P_1 and P_2 to assign a satisfying assignment of the literals as follows: if P_1 goes through v_i, assign

$x_i = F$, and if through $\bar{v}_i$, $x_i = T$. In this case the answer to C with respect to $3SAT$ is also positive.

Conversely, assume there is a satisfying assignment of the variables. If $x_i = T$, let P_1 use $\bar{v}_i$; if $x_i = F$, use v_i. Now, let ξ be a 'true' literal in C_u. If $\xi = x_i$ then v_i is free of the solution graph P_1 and we can use it to find a solution graph from w to c_u; if $\xi = \bar{x}_i$, use $\bar{v}_i$. Finally, use the m edges entering u_c and the edge from u_c to *output* to form the solution graph P_2. $\square$

Acknowledgment

The authors would like to thank an anonymous referee for valuable comments.

References

[1] Beimel, A. and Franklin, M.: Reliable communication over partially authenticated networks. In: *Proceedings of the WDAG '97, Lecture Notes in Computer Science 1320*, Springer Verlag, 1997, 245–259.

[2] Burmester, M. and Desmedt, Y. and Wang, Y.: Using approximation hardness to achieve dependable computation. In: *Proceedings of Second International Conference on Randomization and Approximation Techniques in Computer Science, Lecture Notes in Computer Science 1518*, Springer Verlag, 1998, 172–186.

[3] Dolev, D.: The Byzantine generals strike again. *J. of Algorithms*, **3**, 1982, 14–30.

[4] Dolev, D. and Dwork, C. and Waarts, O. and Yung, M.: Perfectly secure message transmission. *J. of the ACM*, **40**(1), 1993, 17–47.

[5] Ford, L. and Fulkerson, D: *Flows in Networks.* Princeton, NJ: Princeton University Press, 1962.

[6] Fortune, S. and Hopcroft, J. and Wyllie, J.: The directed subgraph homeomorphism problem. *Theoret. Comput. Sci.*, **10**, 1980, 111–121.

[7] Garey, M. and Johnson, D.: *Computers and Intractability: A Guide to the Theory of* **NP**-*Completeness.* San Francisco: W. H. Freeman and Company, 1979.

[8] Hadzilacos, V.: *Issues of Fault Tolerance in Concurrent Computations.* PhD thesis, Harvard University, Cambridge, MA, 1984.

[9] *Information Systems Trustworthiness – Interim Report.* Computer Science and Telecommunications Board Commission on Physical Sciences, Mathematics, and Applications National Research Council. April 1997.

[10] Korte, B. and Lovasz, L. and Prömel, H. and Schrijver A. (Eds.): *Paths, Flows, and VLSI Layout.* Springer-Verlag, 1990.

[11] Lamport, L. and Shostak, R. and Pease M.: The Byzantine generals problem. *J. of the ACM*, **32**(2), 1982, 374–382.

[12] Neumann, P.: Are dependable systems feasible? *Commun. of the ACM*, **36**(2), 1993, 146.

[13] Nicola, V. and Nakayama, M. and Heidelberger, P. and Goyal, A.: Fast simulation of highly dependable systems with general failure and repair processes. *IEEE Transactions on Computers*, **42**(12), 1993, 1440-1452.

[14] Nilsson, N.: *Principles of Artificial Intelligence.* Tioga, 1980.

[15] Perl, Y. and Shiloach, Y.: Finding two disjoint paths between two pairs of vertices in a graph. *J. of the ACM*, **25**(1), 1978, 1–9.

[16] Riecken, D.: Intelligent agents. *Commun. of the ACM*, **37**(7), 1994, 19–21.

[17] Reiter, M. and Stubblebine, S.: Path independence for authentication in large-scale systems. In: *Proceedings of the 4th ACM Conference on Computer and Communication Security*, ACM Press, 1997.

[18] Robertson, N. and Seymour, P: Graph minors XIII: The disjoint paths problem. *J. Comb. Theory, Ser. B*, **63**(1), 1995, 65–110.

[19] Schneider, F.: Towards fault-tolerant and secure agentry. In: *Proceedings of the WDAG '97, Lecture Notes in Computer Science 1320*, Springer Verlag, 1997, 1–14.

[20] Wagner, D. and Weihe, K.: A linear time algorithm for multicommodity flow in planar graphs. *Proc. First European Symposium on Algorithms*, 1993, 384–395.