Department of Computing

Efficient Calculation of Reliability and Performability of Large Computer Networks

Johannes Ulrich Herrmann

This thesis is presented for the Degree of
Doctor of Philosophy
of
Curtin University

April 2013

Declaration

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgment has been made.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university

Johannes Herrmann

Date

Dedication

To my family, who have supported me throughout this work and thereby made it possible. Especially to my wife who has had the patience to accept my seemingly never-ending late nights in front of the computer, and to my mother who shouldered so many burdens that I might be free to carry on this work.

Also to my supervisor, whose constant questions and feedback have shaped this thesis into something far better than what it would have been without his input.

Thank you.

Abstract

Communication in computer networks utilizes a variety of network connectivity models. The most common connectivity models, discussed in literature, are unicast, multicast and broadcast. In unicast models, a communication from the source *s* is required to reach a particular target *t*. For multicast the communication must reach a given group of targets T, while for broadcast the communication must reach every device in the network.

In practice, network devices (nodes) and/or communication lines (links) between devices can fail with some probability due to component wear out, design fault, catastrophic events, loss of battery, *etc*. Thus, the reliability measures of communication networks are important in order to gauge the performance of the network. For some types of network, such as Wireless Sensor Networks (WSN), devices may fail far more frequently than the links between them. Conversely, for networks such as some fixed line networks and radio broadcast networks, communication links can be interrupted far more often than the failure rate of the routing devices. Thus, network reliability measures in the literature consider a network model with fallible devices and fallible links, fallible devices and perfect links, or fallible links and perfect devices.

The reliability measures of a network vary, from a simple requirement for connectivity to time performance guarantees. A number of measures have been proposed, including its Reliability (REL), Expected Hop Count (EHC) and Expected Message Delay (EMD). REL gives the probability that the network is connected but ignores any properties of the connecting path. By contrast EHC estimates how many devices the communication must pass through and EMD estimates the delay of the communication. All are important in the design of modern networks. Computing REL has been shown to be NP-Hard, and thus it is unlikely that a truly efficient algorithm can be found (note that both EHC and EMD are at least as hard as REL).

A number of algorithms have been proposed in the literature to solve REL and EHC for unicast, multicast and broadcast connectivity models for networks with link only failures, node only failures, or link and node failures. No current solutions exist for EMD; all current solutions that address network delay use assumptions that result in the computation of EHC instead. Further, the reliability measures for a number of other connectivity models (*e.g.*, Global Positioning System (GPS),

which requires a communication to reach at least three satellites out of all of the available satellites) have not been addressed. In addition, each solution aims to solve only a specific connectivity and failure model. For example, a solution for EHC may function only for unicast networks with perfect links while a different solution may function for multicast networks with fallible links and either perfect or fallible devices. The best existing solutions use Ordered Binary Decision Diagrams (OBDD). One efficient OBDD-based solution utilizes the concept of boundary sets (BS) as an elegant way to track the effect of component availability and failure; however, BS notation only applies to undirected networks. For this solution and others, space performance has been identified as the limiting factor of network reliability.

This thesis introduces two new unified algorithms, Augmented OBDD (OBDD-A) and Hybrid OBDD (OBDD-H), which are able to solve REL, EHC and EMD for all link/device failure models, for unicast, multicast and broadcast connectivity models as well as a number of others that have been thus far unsolved. Both algorithms use OBDD, with the OBDD-H algorithm being based on BS notation and thus being restricted to undirected networks, while the OBDD-A algorithm introduces a new notation and is applicable to both directed and undirected networks. Both algorithms presented include appropriate data within the nodes of the decision diagrams instead of storing data in a separate structure or implicitly using node position in the OBDD.

The algorithms introduced in this thesis have several key advantages over existing approaches. Firstly, at most two levels of the OBDD are kept in memory at any one time instead of storing the entire diagram. This greatly decreases memory usage; for families of networks with identical inter-connectivity of devices, both algorithms have constant space complexity as well as linear time complexity for REL. Further, they are the first algorithms to address the EMD problem, and can be efficiently applied to a wide range of network models instead of only one or two.

All algorithms and algorithm components are presented as pseudo-code and have been implemented for testing on a range of networks. The performance results from testing agree with the theoretical analysis of complexity presented. In particular, testing verifies a number of theoretical predictions including the time and space-performance of both algorithms for families of networks with identical inter-connectivity. In addition, testing establishes the behaviour of the various metrics considered as network properties change, including the behaviour of the previously unaddressed message delay metric. Finally, testing shows that the

reliability of even very large networks, such as the 4×40,000 grid network, is efficiently computable by both algorithms across different network models.

Acknowledgements

While the work in this thesis is primarily my own work, I would like to acknowledge the following contributions:

- My supervisor, Dr Sieteng Soh, provided the initial impetus behind this work through his work on network reliability and performability. He has tirelessly provided feedback that has shaped this research and extensively edited submitted papers he co-authored. In addition he has given extensive feedback on the thesis document itself, including formatting and content arrangement.
- My co-supervisor, Dr Geoff West, and Prof Suresh Rai also provided feedback to papers they co-authored and some general feedback on the direction of the thesis.
- Matjž Škorjanc was hired to finalize part of the implementation of the algorithm. He also did some additional work on output and input formatting of the implementation.
- A number of people assisted with checking the thesis for spelling and grammatical issues. These were Sophie Divliaev, James Fleming and Yean-Wei Ong.

Contents

List of Figures

List of Tables

Presented Work

This thesis is based upon ongoing research, some of which has been published in journals or presented at conferences. These works are shown below. Note that the most recent paper has been reviewed and accepted for publication, but this thesis has been completed before the date of conference and thus the paper has not been presented at this time.

- J. Herrmann and S. Soh, "Digital Channel Capacity Calculation Using Augmented Ordered Binary Decision Diagrams," 2013 IEEE 16th Int. Conf. Computational Science and Engineering, Sydney, Australia, 2013.
- J. U. Herrmann, et al., "Computing Performability for Wireless Sensor Networks," International Journal of Performability Engineering, vol. 8, Mar 2012, pp. 131-140.
- J. U. Herrmann and S. Soh, "Comparison of Binary and Multi-Variate Hybrid Decision Diagram Algorithms for K-Terminal Reliability," in 34th Australasian Computer Science Conference (ACSC2011), Perth, Australia, 2011.
- J. U. Herrmann, et al., "On Augmented OBDD and Performability for Sensor Networks," International Journal of Performability Engineering, vol. 6, Jul. 2010, pp. 331-342.
- J. U. Herrmann, "Improving Reliability Calculation with Augmented Binary Decision Diagrams," in IEEE 24th International Conference on Advanced Information Networking and Applications (AINA 2010), Perth, Australia, Apr. 2010, pp. 328-333.
- J. U. Herrmann and S. Soh, "A Space Efficient Algorithm for Network Reliability," in 15th Asia-Pacific Conference on Communications (APCC2009), Shanghai, China, 2009.
- J. U. Herrmann, et al., "Using Multi-valued Decision Diagrams to Solve the Expected Hop Count Problem," in IEEE 23rd International Conference on Advanced Information Networking and Applications Workshops (AINA 2009), Bradford, UK, 2009, pp. 419-424.
- J. U. Herrmann, et al., "Communication Network Analysis Using Augmented OBDDs," in The 9th Postgraduate Electrical Engineering and

Computing Symposium (PEECS 2008), Perth, Western Australia, 2008, pp. 93-98.

- J. U. Herrmann, et al., "An OBDD Approach for Computing Expected Hop Count of Communication Networks," in in The 8th Postgraduate Electrical Engineering and Computing Symposium (PEECS 2007), Perth, Western Australia, Nov 2007, pp. 79-84.

Notation

Term	Meaning
(v_f,v_t) and $\{v_f,v_t\}$	The directed and undirected edge, respectively, from vertex v_f to vertex v_t. When an edge may be either directed or undirected, parentheses are used. Note that, as described in Section 2.10.4, an undirected edge is always written with $f<t$.
BS	Boundary Set. This abbreviation could refer to either the boundary set itself or (more commonly) algorithms utilizing boundary sets.
CI_i, VI_i and VS_i	These are terms used for the information stored in node N_i of an OBDD-A (*q.v.*). CI_i is the Condition Information, VI_i is the Vertex Information and VS_i is the Vertex Set. The definitions of these are given in Section 3.2.
DD or OBDD	Decision Diagram or Ordered Binary Decision Diagram, as per Section 2.8. In addition, some algorithms are referred to as, for example, OBDD-based to indicate that the algorithm makes use of an OBDD.
F_k	The boundary set of level k of the decision diagram.
OBDD-A	The Augmented OBDD algorithm (and the diagram created by such an algorithm) as discussed in Chapter 3 and Section 5.3.
OBDD-H	The Hybrid OBDD algorithm (and the diagram created by such an algorithm) as discussed in Chapter 4 and Section 5.4.
OBDD-A<suffix> or OBDD-A<suffix>	The suffix behind the OBDD-A or OBDD-H indicates a form of the algorithm for a particular component failure model or metric. The component failure models are 'e' for fallible edges, 'v' for fallible vertices and 've' for both vertices and edges being fallible. The metric abbreviation used is 'p' for performability versions of the OBDD-A and OBDD-H, which can compute REL, EHC and EMD. The algorithms discussed in Chapters 3 and 4 are restricted to the computation of REL and do not have a performability suffix. For example, OBDD-Ape refers to the performability version of the OBDD-A for network models with fallible edges and perfect vertices.
Set, partition and block.	A set may be partitioned into a number of subsets. This work refers to each of these subsets as a block, and refers to the group of blocks of a set as the *partition* of that set.
Vertex, edge, node, link	'Vertex' and 'edge' are used when discussing the graph-based network models, while 'node' and 'link' are for decision diagrams
WSN	Wireless Sensor Network or Wireless Sensor Networks, depending on context.

Chapter 1
Introduction

1.1. Problem Overview

Networks, including computer communication, power distribution or road transport, play an important role in everyday life. The former, in particular, are rapidly increasing in importance in areas including business, leisure, defence and politics, and are the focus of this thesis.

The structure and connectivity requirements of networks vary widely depending on their application. The differences include the interconnectedness of the links and junctions of the network, as well as the requirements on the traffic passing through the network. Some traffic may be sent from multiple redundant sources and/or be required to reach one or more of multiple targets. For example, five copies of an important message may be sent in parallel from five different sources, but it may only be required that one of these reach any one of the valid destinations (or targets).

One challenge of network design and analysis is determining whether traffic is able to pass through the network from the source(s) to the destination(s) given the possibility that routing devices or communication links (or both) may fail. When one path through the network becomes unavailable, traffic is rerouted through an available path where possible. Ideally, networks are designed so that such rerouting will maintain connectivity in most cases. The probability that the network is connected at any given time, given the source and target requirements, is referred to as its reliability. Hence both designing new networks and analysing existing ones can be greatly assisted by the computation of their reliability.

In addition, the performance of a network is often measured by how quickly traffic can pass through the network from the source(s) to the target(s). The network may have constraints on traffic flow or delays on links and/or junctions of links that affect its performance. This performance is determined not just by the structure of the network, but also by which components of the network are available, since rerouting traffic may force it along a less optimal (*i.e.*, longer) path. There are a number of metrics that combine the performance and reliability of the network; these are collectively known as the network's performability. A key performability

measure is the Expected Message Delay (EMD), which estimates how long a message will take to pass through a network.

Many approaches have been proposed for both network reliability and performability. Unfortunately, both problems are known to be NP-Hard, and thus the time and space complexity of existing solutions restricts the networks to which they can be applied. Recently, Ordered Binary Decision Diagrams (OBDD) have been used to provide solutions that are efficient in terms of processing time for computing network reliability. Unfortunately these solutions apply only to a very small subset of possible problems; that is, networks with undirected links and perfect nodes that use only multicast or broadcast communication protocols. While the OBDD has not been used to solve performability, less efficient solutions exist for both reliability and some performability calculations on similarly restricted families of networks and protocols.

This thesis focuses on providing efficient OBDD-based algorithms for the computation of network reliability and performability that are superior to existing algorithms. The space performance of the algorithms presented is superior to existing algorithms while the time performance is at least comparable. Further, the algorithms presented are flexible enough to compute the reliability and/or performability (especially the message delay) of a wide selection of networks, irrespective of device or communication failure and connectivity requirements.

1.2. Motivation

1. Network reliability is useful for the analysis of existing networks and the design of new networks. A number of methods exist for solving network reliability, as discussed in Chapter 2, but their time and space complexities are exponential due to network reliability being an NP-Hard problem. Networks are becoming larger and more inter-connected, so more efficient methods of computing network reliability are required. In particular, the limiting factor of several existing algorithms is space complexity.

2. Current solutions generally apply only to restricted source/target combinations. In addition, most solutions allow for the failure of either routing devices or communication links, but not both. When both devices and links can fail, the exponential complexity increase makes even problems of moderate size prohibitive. Finally, some solutions have additional requirements, such as all communication links being undirected. A flexible

method of computing network reliability for as many network models as possible is required.

3. Existing OBDD approaches are only able to compute network reliability, but more information may be required to properly assess the 'goodness' of a network. In particular, the EMD of a network is increasingly important due to an increase of streaming applications and others that require communication guarantees. Further, while EMD has been discussed in the reliability literature, the solutions presented have only applied to a special case, the Expected Hop Count (EHC) that considers only homogenous link delays and not device delays. While OBDD approaches have been shown to be efficient for reliability, they have not been applied to EHC. An application of OBDD to EMD would be beneficial; this application should also solve EMD.

1.3. Aims and Approaches

The objectives of this thesis are as follows:

Aim 1: To improve the efficiency of computing network reliability for computer communication networks. In particular, to make use of the OBDD approach as it has been shown to be superior to other approaches in a number of areas.

Chapter 3 introduces the Augmented Ordered Binary Decision Diagram (OBDD-A), which compares positively with existing solutions in terms of memory requirements, and is competitive with other solutions, in terms of processing time. Chapter 4 introduces the Hybrid Ordered Binary Decision Diagram (OBDD-H) to incorporate the efficient boundary set notation into the OBDD-A to improve performance for undirected networks.

Aim 2: To compute reliability for different network communication models, specifically those with multiple source and/or target devices. At the same time, to compute reliability for different network component failure models, including efficiently allowing for the failure of both network nodes and links. Ideally, the algorithms should be able to compute reliability for any sort of network, irrespective of its communication models or underlying topology.

Chapter 2 introduces five communication models that cover the majority of possible source-target connectivity requirements. The algorithms in Chapters 3 and 4 are capable of computing the reliability of networks with multiple source and/or target devices and fallible network links and devices.

Aim 3: To compute EMD for different network communication models, including being able to solve problems that have arbitrary delays in both links and devices.

As discussed in Chapter 2, the existing works address only a restricted version of EMD (*i.e.*, EHC) and are not applicable for the general EMD problem. Chapter 5 addresses the application of both the OBDD-A and OBDD-H to solving EMD, and thus also EHC.

Aim 4: For each of the above, this work aims to first produce a valid algorithm and then to implement the algorithm to show that it functions as intended. Testing must also demonstrate the performance of each algorithm.

Each algorithm and modification has been implemented and tested. Details are given at the end of each section after any examples. The implementation demonstrates the correctness and efficiency of the algorithms and discusses performance-related issues.

1.4. Contribution and Significance

This thesis makes three main contributions to the field of computer communication reliability:

1. The OBDD-A and OBDD-H algorithms described in Chapters 3 and 4 of this thesis have computation time that is at least competitive with the leading algorithms in the field. The key concept of the inclusion of network state probability in diagram nodes allows the computation of reliability with a single pass. This, in turn, significantly improves the space complexity of both algorithms including having constant space complexity (and linear time complexity) when the problem is restricted to families of networks that have fixed inter-connectivity (*i.e.*, fixed width[1]).
2. A number of network communication models are formalized in Chapter 2 of this thesis, including models that have not previously been addressed. Both the algorithms presented are able to analyse networks under the various models presented, as well as all component failure models. This enables the OBDD-A and OBDD-H to compute the reliability and performability for networks with communication and component failure models unsolvable by existing algorithms. The boundary set notation used by the OBDD-H is extended in order to enable the algorithm to solve a wider range of models than the original notation permits.

[1] The width of a network is defined in Section 2.10.4 and is dependent on the ordering of the vertices of the network.

3. In Chapter 5, both the OBDD-A and the OBDD-H algorithms are shown to be able to compute the EMD of a wide range of networks, demonstrating the flexibility of both algorithms. Further, the information generated is sufficiently general that it can be easily adapted to a range of other metrics, as discussed in Chapter 6.

1.5. Thesis Outline

This thesis is organized as follows. Chapter 2 provides the background on the relevant problems and existing solutions. Further, the details relating to the implementation and testing of the algorithms presented are introduced. Chapter 3 presents the OBDD-A for computing the reliability of the network model with a single source and target, and perfect vertices. This is extended to the OBDD-H in Chapter 4, and both algorithms are extended for the computation of EMD in Chapter 5. Finally, Chapter 6 summarizes the findings and presents a number of avenues of research that are yet to be explored.

Chapter 2
Background and Literature Review

2.1. Chapter Overview

Various networks of different types are prevalent around the world. Cities and countries have transportation and power distribution networks, as well as a variety of communication networks. The latter include computer communication networks, wired and wireless telecommunication networks and wireless sensor networks.

Each network consists of a series of network links which intersect at network nodes. Depending on the type of the network, links may take the form of roads, shipping lanes, transmission wires or wireless signals. The nodes where these intersect can be road intersections, transmission stations or routers. This thesis considers only computer communication networks; however, the algorithms presented also apply to the other networks.

The network model and notation are introduced in Section 2.2, and a number of communication and component failure models are discussed in Sections 2.3 and 2.4, respectively. Section 2.5 discusses the most common approaches to computing the reliability of a communication network, and Section 2.6 discusses several more general metrics for computing network 'goodness'. Existing approaches for solving reliability and the other metrics are discussed in Section 2.7, with Sections 2.8 and 2.9 focusing on two of these; Ordered Decision Diagrams and Boundary Sets, respectively. Section 2.10 discusses network topologies used in this thesis, and Section 2.11 discusses details relevant to the algorithm implementation. Finally, Section 2.12 summarizes this chapter.

2.2. Network Model and Notation

We consider a computer communication network to be composed of communication devices (*e.g.*, routers, transmitters or sensors), which are connected by wired or wireless communication links. The network is modelled using a graph G=(V, E) whose set of vertices V represent communication devices and whose set of edges E represent communication links between the devices. The network is used to transmit information from one or more source devices to one or more target (or sink) devices, through communication links and one or more intermediate devices.

The graphical representation of the network is shown in Figure 2.1. Each vertex of the graph is represented by a circle and labelled with its index; vertex v_j is labelled with integers j=0, 1, 2, …, |V|-1. Edges in the graph are shown as arrows[2], and are likewise labelled with non-negative integer indices. The arrows show the edge to be either undirected (*e.g.*, $e_2=\{v_1,v_2\}$is the undirected edge between v_1 and v_2), or directed in the direction of the arrow (*e.g.*, $e_3=(v_1,v_3)$ is the directed edge from v_1 to v_3). The source vertices (in this case v_0) are denoted by inner circles surrounding their indices. Target (sink) vertices (in this case v_3) are denoted by a double outer circle. For networks with multiple source and target vertices, let S⊆V and T⊆V be the set of source and target vertices, respectively. If target vertices are divided into g groups T_i, we have $T=\bigcup_{i=0}^{g-1} T_i$ and $\forall i,j\left(T_i \cap T_j = \emptyset\right)$. Hence, the target groups partition T; similarly, source vertices can be formed into groups S_i that partition S. When |S|=1 the source vertex is denoted by *s*; similarly, the target vertex is denoted by *t* when |T|=1.

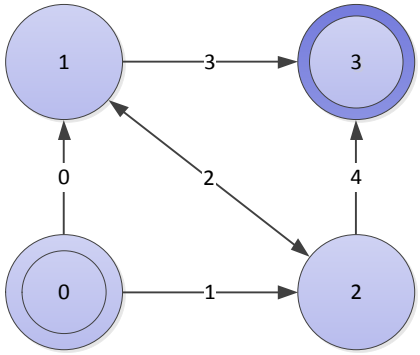


Figure 2.1: Graph Model of a Simple Communication Network

Each vertex and edge may have attributes (*e.g.*, reliability, delay), depending on the application being considered. For any part of the network that may fail, its corresponding graph component (edge or vertex) can be assigned a probability of being operational of $Pr(e_i)$ and $Pr(v_i)$ for an edge e_i and vertex v_i, respectively; $0\leq Pr(e_i), Pr(v_i)\leq 1.0$. In this thesis, each component can be either available (up) or failed (down); this component failure model and others are discussed in Section 2.4.

For an application involving the likely delay of information through a communication network, each e_i and v_i can be assigned a delay value $D(e_i)$ and $D(v_i)$ respectively; $D(e_i), D(v_i)\geq 0$.

[2] When a network is entirely undirected, the arrows may be omitted from each edge.

An (*x*,*y*) *simple path* between vertices v_x and v_y is formed by the set of available edges and/or vertices, such that no vertex is traversed more than once. Any proper subset of a simple path does not result in a path between the vertex pair. We write a simple path using the subsets of the edges traversed by the path. For example, the simple path in Figure 2.1 that starts at v_0 and reaches v_3 by traversing e_0 and e_3, is written {0,3}. The *pathset*, $P_{ST,}$ is a set whose elements are ($s \in S$, $t \in T$) simple paths (also called minpaths or (*s*,*t*)-minpaths). For example, for S={0} and T= {3} in Figure 2.1, P_{ST}={ {0,3}, {1,4}, {1,2,3), {0,2,4} }.

Similarly, an (*x*,*y*) *simple cut* between vertices v_x and v_y is formed by a set of edges and/or vertices, such that any simple path between v_x and v_y must include at least one edge or vertex from that cut. Any proper subset of a simple cut does not result in a cut between the vertex pair. A *cutset*, is a set whose elements are ($x \in S$, $y \in T$) simple cuts.

2.3. Network Communication Models

Each network carries some amount of load. With a communication network, this load is composed of information, which may take several forms; for a computer communication network such as the Internet, the information takes the form of packets of data. The load of the network is abstracted away by using end-to-end or connectivity tracking.

End-to-end tracking models the flow through the network by tracking information on the path(s) taken. Flow is assumed to start at any network device marked as a source, and move through the network until it reaches a device marked as a target. Depending on the application, the actual path taken may not be tracked, although some aspect of the path (*e.g.*, most recent devices reached) will be.

For example, if the application requires the number of hops taken, then the hop count would be tracked for every path, while other information would be excluded. When tracking flow through a network where multiple choices are possible for a single transmission, we assume that the optimal path is taken (*e.g.*, the shortest path or the path with the greatest bandwidth). The optimal path may change as network components (devices and/or links) fail.

Connectivity tracking records which vertices are connected, given components known to be active; unlike end-to-end tracking, it does not record any information on what path is taken to make up the connections.

Table 2.1: Various Communication Network Models and Their Associated Problem Statements

Models	Description	Application
Model 1: One-to-one (s,t)	Single source to single target. Known as Unicast.	Network communication between two devices once route has been established.
Model 2: One-to-all	Single source to all targets.	
Model 2.1: (s,T)	Single source to all targets; targets are selected from all possible vertices (T⊆V). Known as Multicast.	UDP uses multicast for network communication. Also used in stock exchanges and for multimedia content delivery.
Model 2.2: (s,V)	Single source to all vertices. Known as Broadcast.	A computer communication device broadcasts its identity to all other devices on the network.
Model 3: One-to-any	Single source to any of multiple targets.	
Model 3.1: (s,1-of-T)	Single source to any one of multiple targets. Known as Anycast.	Mobile phone query to nearby towers; may receive several replies but requires at least one.
Model 3.2: (s, c-of-T)	Single source to a given number of multiple targets.	Global Positioning System requires three satellites to determine locations.
Model 3.3: (s, c_i-of-T_i)	Single source to a given number each of multiple target groups.	Multimodal broadcast containing instructions for several different sensory types (*e.g.*, temperature, pressure, humidity) for an environmental monitoring WSN).
Model 4: Any-to-one	Any of multiple sources to single target.	
Model 4.1: (1-of-S,t)	One of multiple sources to single target.	Bush-fire or intrusion detection in a single sensor field.
Model 4.2: (c-of-S,t)	A given number of each of multiple sources to single target.	A detection system (*e.g.* for fires or undersea earthquakes) may require multiple sensors to detect an event in order to reduce false alarms.
Model 4.3: (c_i-of-S_i,t)	Several of each of multiple groups of sources to single target.	A smart house with sensor groupings around various fixtures and appliances may require a sufficient number of sensors to be able to communicate with the base station.
Model 5 : Any-to-any	Multiple sources to multiple targets.	
Model 5.1: (1-of-S, 1-of-T)	One of multiple sources to one of multiple targets.	Sensor network with redundant terminals or base stations. Group of search aircraft or ships searching for site marked by multiple beacons.
Model 5.2: (c^s-of-S, c^t-of-T)	Several of multiple sources to several of multiple targets.	Critical network requiring large amount of redundancy.
Model 5.3: ($c^s_{i,j}$-of-S_i, $c^t_{i,j}$-of-T_j)	Several of each of multiple groups of sources to several of multiple groups of targets.	Sensors with multi-sensory abilities communicating with each other and sensory-specific base stations.

Table 2.1 summarizes the various network communication models with possible applications that use them. One example application is the wireless sensor network (WSN), in which sensor nodes sense input from an event, and forward appropriate

data towards one or more target devices. Data dissemination in WSNs can be categorized into *commands* and *events* [1]. *Commands* are issued from one task manager and sent to one or more target sensors (nodes) in the field.

For such communications, Model 1: Unicast (*s*,*t*), Model 2: Multicast and Broadcast (*s*,T), and Model 3: Anycast (*s*,*c*-of-T) communication models are typically used, for a task manager *s* and a target sensor *t* or a set of target sensors T in the field. *Events* are sensed by one or more sensor nodes and relayed to the task manager *t*. If a single event is sensed, the unicast(*s*,*t*) is used; here '*s*' is the sensor node and '*t*' is the task manager.

If multiple sensors detect one or more events, one communication type in the Model 4: Many-to-one (S,*t*) is used, where S is the set of multiple sensors. Model 5: Many-to-many (S,T) is used for networks with multiple source and target vertices, such as a network with multiple sensors and multiple monitoring stations, or a network with redundancy in both sensors and base stations. Detailed descriptions of the various communication models are given below.

2.3.1. Models 1 and 2

In Unicast(*s*,*t*), a single source node *s* attempts to communicate with a single target node *t*. Broadcast(*s*,V) is used in network communications where a source node *s* broadcasts information to all target nodes V. WSNs in particular make frequent use of unicast and broadcast communication [2] for disseminating user commands/queries to one or all target sensors. Model 2 also includes Multicast(*s*,T), where a single device broadcasts to each of a set of target devices T. The difference between Multicast (Model 2.1) and Broadcast (Model 2.2) is that, for the latter, the communication is sent to every device in the network (T=V), where V is the set of network devices; Multicast selects the target devices, but does not necessarily select all of them (T⊆V). The literature contains vast amounts of research on the network reliability of Unicast [3-14], Multicast [15-17] and Broadcast [9, 18, 19].

2.3.2. Models 3 and 4

Model 3 includes three sub models: 3.1:(*s*,1-of-T), 3.2:(*s*, *c*-of-T) and 3.3:(*s*, c_i-of-T_i) and Model 4 also includes three sub models: 4.1:(1-of-S,*t*), 4.2:(*c*-of-S,*t*), and 4.3:(c_i-of-S_i,*t*). The sub models (*s*,1-of-T) and (1-of-S,*t*) come about due to the failure prone nature of WSN devices [2]. Sensor nodes may fail for a number of reasons, ranging from power failure to direct sabotage. Communications between them may also fail if the power becomes insufficient to transmit the required

distance, if an obstruction is moved between the devices, due to deliberate jamming, or for other reasons. One common method to counter-act this propensity for failure is to increase the number of nodes deployed, even if only information from one sensor node is needed. This model also applies to some other networks, such as mobile or ad-hoc networks [2, 20] where a (possibly portable) device must be able to contact a communication hub; if multiple hubs are present, the device must be able to contact one suitable hub for successful communication.

The sub models (*s*, *c*-of-T) and (*c*-of-S,*t*) apply to WSN, in which a number of sensor nodes detect the same event; these sensors send redundant information, and the target requires *c* of them to arrive before it can start reacting to the occurring event. For example, consider a WSN with many redundant target sensors for monitoring a remote forest for the existence of bush fires. Each target sensor is expected to send an alarm signal to the target as soon as it detects signs of a fire, although it may be required that *c* sensors detect a fire before an alarm is sounded in order to minimize false positives.

Sub models 3.3 and 4.3 are more complex, and generally deal with multi-modal data. One device may be required to communicate with several groups of devices, needing to reach one or more of each of these groups. For example, if a device is capable of communicating with both wireless and Bluetooth [2], successful communication may require contact with one or more of each type. In a more general model (*i.e.*, (c_i-of-S_i,*t*)), the sensors are grouped, where each sensor group is monitoring a different class or location of phenomena, or *vice versa*. This model is applicable to a number of different scenarios, such as climate monitoring and various surveillance systems [1].

Regardless of the application, data dissemination may be event/time driven, and reliable and timely event detection is critical so that appropriate actions can be performed as quickly as possible. In general, reliable event detection at the station is based on collective information provided by sensor nodes. Section 3.6.5 gives an example of a security system requiring models 3.1, 3.2 and 3.3.

Models 3 and 4, and their corresponding sub models (*e.g.*, 3.1 and 4.1), are the reverse of each other; one can be modelled from the other by reversing the direction of the network. For example, consider the network shown on top in Figure 2.2. The network has $S=\{v_0,v_1\}$ and $t=v_9$, which is an instance of Model 4.2 since $c=2$. When the direction of all directed edges are reversed, the target vertex is made into a source vertex and *vice versa*, the result is the network shown on the

bottom of Figure 2.2. Solving this network with c=2 using Model 3.2, we obtain an identical result to solving the original network under Model 4.2.

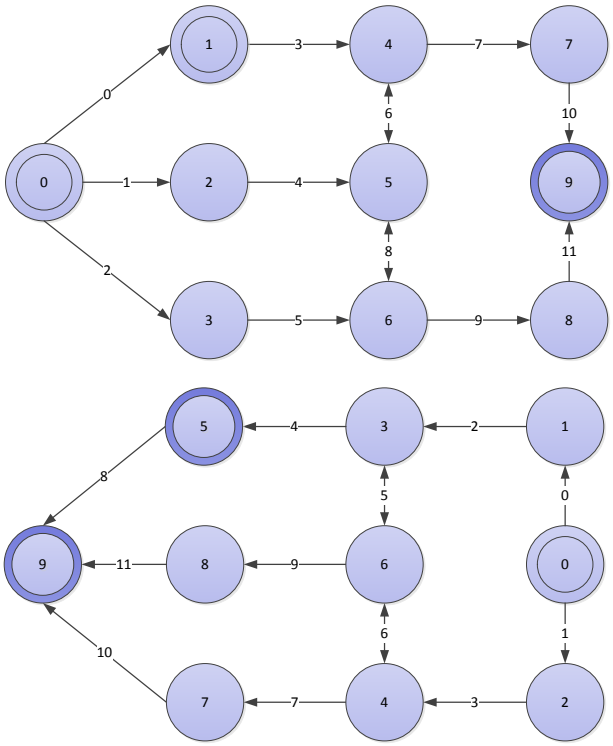


Figure 2.2: Model 4.2 vs. Model 3.2

Both models 3 and 4 are included for the sake of completeness. For simplicity, in this thesis, we use the sub models in Model 3 to also refer to their corresponding sub models in Model 4, (*e.g.*, comments regarding (*s*,1-of-T) also apply for (1-of-S,*t*)) and do not otherwise address Model 4.

As demonstrated, Model 3 and Model 4 are relevant in practice, but little work has addressed the use of the two models, except for work that forms part of this thesis [21, 22]. Note that AboElFotoh *et al.* [23] proposed the DSNREL problem, which is equivalent to computing the reliability of (1-of-S_i,*t*) or (*s*,1-of-T_i), depending on the direction of the communication, and are thus restricted forms of Model 3.3 or 4.3, respectively. To the best of our knowledge, no work exists that can solve the unrestricted versions of Model 3 or 4.

2.3.3. Model 5

Model 5 is the most general model, allowing multiple source and target vertices. Due to its complexity it has not been considered in reliability literature. It could be argued that this model has more sub-categories (*e.g.*, Model 5.2 also includes the models (c^s-of-S, 1-of-T) and (1-of-S, c^t-of-T) which could be listed separately); this

thesis keeps the number of sub-models minimal, since Model 5 is not addressed in detail in this work.

2.4. Component Failure Models

The reliability of a component may not be known, but can be estimated with accuracy depending on the amount of data available. The mean time to failure (MTTF) and mean time to repair (MTTR) of a component can be calculated from usage data. Using such statistics the availability (reliability) or unavailability (unreliability) can be calculated. A number of methods for achieving this are available in the literature, such as the work by Kirby and Schwartz [24] to calculate the unavailability of protective relays, which applies equally to the components of other networks. This thesis uses a stochastic network model, *i.e.*, each component of a network has a known reliability (operational) or unreliability (failed). Transitions from operational to failed, or *vice versa*, are not considered - other than those modelled by the definition of component reliability using MTTF and MTTR. Components which can take multiple states are also not considered. The assumptions are consistent with network reliability literature [4, 10, 15, 17, 18, 25-29].

For ease of duplication, most works in the literature assume $Pr(e_i)=0.9$ and/or $Pr(v_i)=0.9$, depending on whether edges or vertices (or both) are fallible. This practice is also followed in this thesis for easier comparison with other works. It should be noted that accurate simulation requires that individual components can take different probability values [7], and the algorithms presented in this thesis are designed to allow this. The performance of the implementation of the algorithms presented does not vary for any reliability values, but all data files used list the reliability of each fallible component as 0.9. [3]

In general, the failure of network components may be dependent either on the failure of other components, or on events. In particular, the failure of one component may increase the chance of failure of other components. For example, if one component of a network fails due to a power surge or power failure, nearby components may be statistically more likely to fail as well, since they are known to be 'close' to such an event. Similarly, if one wireless communication experiences interference and thus fails, it may be statistically more likely that nearby wireless communications fail due to the same interference.

[3] The implementations of the algorithms given in this thesis can be modified if $Pr(v_i)$ or $Pr(e_i)$ are allowed to take the values 0 or 1. In either case, the status of the component is known; an algorithm taking advantage of this is more efficient than one that does not. Both probabilities are unrealistic, however, and thus this work does not allow these values to be taken.

There are, however, a number of cases where dependent failures are not appropriate. For example, in a wired communication network with relatively long wires, it is unlikely that one event can cause multiple wires to fail. Hence, one failure would not increase (or decrease) the probability of other component failures. Similarly, one routing device in such a network failing is unlikely to affect the failure chance of others, and hence their failure probabilities are also independent. It could be argued that one event could affect all available repair crews, thus affecting the MTBR, but such issues are beyond the scope of this thesis.

While considering the problem of dependent failures is worthwhile, it adds a large amount of complexity to the problem of network reliability. Traditional work in the field of network reliability [4, 10, 15, 17, 18, 25-29] assumes independent failure of network components [16]. This thesis similarly assumes that all component failures are statistically independent.

The above models do not consider whether edges, vertices or both are susceptible to failure. Much of the literature [4-6, 15, 17, 18, 26, 30, 31] assumes that vertices are perfect, and it has been argued that this is appropriate for some real-life networks, such as radio broadcast networks [3]. This thesis considers vertex failure, edge failure, and the case where both edges and vertices fail. Hence, each network model given in Table 2.1 has three aspects labelled ‘v’ for vertex failure, ‘e’ for edge failure and ‘ve’ when both are fallible. For example, Model 3.3v refers to network Model 3.3 with fallible vertices and perfect edges.

2.5. Communication Network Reliability

2.5.1. Introduction

Research into network reliability began in the 1970s, and is an important factor in network design and operation [16]. The reliability of a graph, and hence the communication network it represents, is defined as the probability that it is connected. That is, messages or other traffic can pass from the source(s) to the target(s) by one or more paths of available vertices and edges. The availability of the vertices and edges depends on the *state* of the network.

A *network state* $\Omega=(V_U,E_U)$ of network $G=(V,E)$ is a partition of G, such that all vertices in $V_U \subseteq V$ and edges in $E_U \subseteq E$ are active and all other vertices and edges in G are inactive. The probability of a state $\Omega=(V_U,E_U)$ is computed as:

$$\Pr(\Omega)=\prod_{v_i\in V_U}\Pr(v_i)\prod_{v_i\notin V_U}(1-\Pr(v_i))\prod_{e_i\in E_U}\Pr(e_i)\prod_{e_i\notin E_U}(1-\Pr(e_i)) \quad (2.1)$$

since all failures are assumed to be statistically independent. A state is a *success state* if it meets the connectivity requirements of the relevant communication model described in Table 2.1. Assuming that both edges and vertices can fail, there are $2^{|V|+|E|}$ network states in G, but reliability (and performability, as discussed in Section 2.6) are computed only from the set of all success states, Ω_S.

A number of network communication models were introduced in Section 2.3. Each of these imposes different conditions on the reliability of the network. While the reliability of Models 1 and 2 has been addressed at length in the literature, other models (such as Model 5) have not been addressed. The component failure models most commonly used in reliability literature are discussed in Sections 2.5.2-2.5.4, and the performability measures other than reliability are discussed in Section 2.6. Approaches to computing reliability and performability are discussed in Section 2.7.

2.5.2. Two-Terminal Reliability

The simplest measure of network reliability is the probability that two points in the network – a single source vertex *s* and a single target vertex *t* – are connected. This probability is referred to as *two-terminal reliability* (or *s-t reliability* and abbreviated 2-REL or REL(*s*,*t*)), or simply terminal-reliability. Formally, 2-REL is the probability that at least one active (*s*,*t*)-minpath exists in the network. The problem of computing 2-REL has been shown to be #P-Complete [3].

2-REL is an instance of communication Model 1 in Table 2.1, and is applicable to a number of real-life network scenarios, mainly involving fixed communication links. For computer communication networks, 2-REL is commonly called unicast.

2.5.3. K-Terminal Reliability

K-terminal reliability (K-REL) is defined as the probability that the vertices in the set K are connected by one or more paths of active vertices and edges in the network. With undirected edges, this is equivalent to the problem of a source vertex *s* being connected to another |K|-1 vertices in T, with $s \cup T = K$. If edges are not required to be undirected the problems are not identical, but this distinction is not raised in the literature. K-REL is communication Model 2.1 in Table 2.1, and is a more general form of both 2-REL and ALL-REL, described later. For computer communication networks, K-REL is commonly called multicast.

2.5.4. All-Terminal Reliability

All-Terminal Reliability is a special case of K-Terminal Reliability where |T|=|K|=|V|, requiring every vertex in the network to be able to reach every other vertex. This network reliability measure is often called all-network reliability (ALL-REL). ALL-REL uses communication Model 2.2 in Table 2.1. For computer communication networks, ALL-REL is commonly called broadcast.

2.6. Communication Network Performability

2.6.1. Expected Hop Count and Message Delay

While network connectivity is a requirement for communication through the network, other measures of network 'goodness' may also be appropriate. For example, a WSN may need to minimize the number of sensor nodes that must forward a message in order to reduce power consumption, or a streaming application may require a minimum delay guarantee in order to function effectively.

The number of sensor nodes passing on a message can be measured in the number of steps (hops) that this message takes through the network. The expected value of the number of hops, the Expected Hop Count (EHC), is a good estimator for this.

Computing the EHC requires the probability of each success state (as per Eq. (2.1)), as well as the *length* of each success state, $\Omega \in \Omega_S$, denoted as $1 \leq L(\Omega) \leq n-1$. Thus, $L(\Omega)$ is the length (the number of hops or hop count) of the shortest minpath contained in Ω. We assume that the routing protocol in the network always finds the shortest available minpath [32]. When one minpath is unavailable (*e.g.*, because of vertex or edge failures) the router finds the next possible shortest minpath that is available.

The above holds for the EHC of Model 1, written EHC(s,t) and other models that require only one communication to reach the target, such as Model 3.1. If a model requires at least c messages from one or more sources to reach the target, then $L(\Omega)$ is the length of the c^{th} shortest such message. For example, if c=3 and state Ω contains minpaths of length 2,3,4 and 5 then $L(\Omega)$=4. This is because while some messages arrive earlier (with 2 and 3 hops in this case) the communication is not completed successfully until the c^{th} (*i.e.*, the 3rd) message arrives (taking 4 hops). This concept is discussed further in Section 5.2.

Formally, the EHC is given by:

$$\mathrm{EHC}(\Omega) = \frac{\sum_{\Omega \in \Omega_S} \left(\mathrm{L}(\Omega) \times \Pr(\Omega)\right)}{\sum_{\Omega \in \Omega_S} \Pr(\Omega)} \tag{2.2}$$

Note that the denominator in Equ. (2.2) is the sum of all success states, and thus gives REL. AboElFotoh [33] has shown that computing EHC for Model 'v' is #P-hard.

A generalization of EHC is EMD, the Expected Message Delay. Instead of counting the number of hops a message requires, EMD measures the delay of a message passing through the network. If the delay at each vertex is one and edges do not produce delay, the two problems are equivalent. The literature generally [32-34] uses the term 'message delay' even when this assumption is made.

The delay $\mathrm{D}(\Omega)$ of network state Ω is defined as the shortest (or c^{th} shortest $c \neq 1$) delay of any minpath in Ω. The delay of a minpath is the sum of the delays of all vertices and edges in that minpath. The EMD is given by:

$$\mathrm{EMD}(\Omega) = \frac{\sum_{\Omega \in \Omega_S} \left(\mathrm{D}(\Omega) \times \Pr(\Omega)\right)}{\sum_{\Omega \in \Omega_S} \Pr(\Omega)} \tag{2.3}$$

It can be seen that the EMD formula is very similar to EHC. While it could be argued that the implementations of EMD and EHC are likewise similar, even small differences can become important when dealing with exponential algorithms. Hence, if one claims to solve EMD, it is important to show the algorithm's performance for the general case; not just the special case of EHC [35].

2.6.2. Bandwidth and Flow

Another metric of increasing importance in modern networks is the amount of flow that can pass through the network. Given a network represented by graph G=(V,E) with source, *s*, and target, *t*, the maximum flow problem can be stated as "find the maximum value of $f_i(t)$ such that $\forall v \in \mathrm{V}$ $(v \neq s,t)$ $f_i(v) = f_o(\mathrm{v})$ and $\forall e \in \mathrm{E}$ $f(e) \leq f_{max}(e)$ where $f_i(v)$ and $f_o(v)$ are the flows into and out of vertex *v* respectively, and $f(e)$ and $f_{max}(e)$ are the flow passing through and the capacity of edge *e*".

The value $f_i(t)$ represents the maximum amount of flow that can pass through the network at any given time, which can also be considered as a steady-state flow. Examples of the maximum flow problem include cars in a road network, electricity in a power network, and messages in a telecommunication network [36].

The maximum flow problem is of obvious interest when considering the performability of a computer communication network; many types of network require bandwidth guarantees for such applications as video streaming and telephony. When considering WSNs such as a military network designed to monitor enemy movement, sufficient information flow is critical.

There are two main approaches to the network flow problem; the augmented path and preflow-push methods. Several polynomial-time algorithms exist for both categories, with modern preflow-push algorithms outperforming augmented path algorithms [37].

Edmonds and Karp [38] showed that the Ford and Fulkerson algorithm [39] runs in time $O(nm^2)$ for a network with n vertices and m edges. Independently, Dinic [40] introduced layered networks, an $O(n^2m)$ algorithm. This bound was improved to $O(n^3)$ by Karzanov [41], who introduced the concept of preflows in a layered network. Further improvements have been made, with the dynamic tree approach by Sleator and Tarjan [42] being $O(n \log m)$, and the extension of the blocking-flow approach by Goldberg and Rao [43] requiring $O\left(min\left(n^{2/3}, m^{1/2}\right)\right).m.log\left(n^2/m\right).logU$, where U is the upper bound for the integral edge capacities.

The maximum flow algorithm assumes that all graph edges and vertices are available. In many types of networks this may not always be the case. We define $f(\Omega)$ as the maximum flow for the network resulting from network state Ω. The Capacity Related Reliability (CRR) [8] is the probability that a network can achieve a given network flow, called the minimum flow $f_{\min}$. The CRR of the network G=(V,E) can be calculated by summing the probabilities of all network states $\Omega=(E_U,V_U)$ which have $f(\Omega)\geq f_{\min}$. While enumerating all network states is infeasible for networks of even moderate size, Soh and Rai [44] have presented an efficient way of calculating the CRR of a network using cutsets.

The problem of computing the CRR of a network is introduced as an example of a problem that the algorithms presented in this thesis are not suitable for solving [45]. This is addressed in the conclusion to Chapter 6.

2.6.3. Other Performability Metrics

As discussed in Section 2.4, the assumption that failures are independent may not be appropriate for certain networks. In particular, some networks may be susceptible to having multiple components fail due to a single event, such a power

failure, communication jamming, *etc*. Such events cause what are referred to as *common cause failures* (CCF). Failure to allow for CCF may cause a network's reliability to be over-estimated [46].

This thesis uses the model of independent component failure, and thus does not address CCF. Network reliability is useful for network design and upgrade, problems for which CCF is not usually considered. Solutions utilizing the Ordered Binary Decision Diagram (OBDD) have been proposed for CCF [46-49], and the possible extension of the algorithms in this thesis to CCF are discussed in Chapter 6.

A concept that is specific to sensory networks such as WSN is *coverage*, which is defined as the ability to sense every point in the region(s) of interest by one or more sensors [46], which must be able to communicate with the appropriate base station. A related concept is *K-coverage*, which is the ability of a network to cover each point in the region with at least K sensors [46].

Computing the reliability of a network using Models 3.3 or 4.3 from Table 2.1 is equivalent to computing the sensor coverage of a network, if we define the source or target groups appropriately. Each such group would represent sensors covering one area of interest. Hence, the algorithms introduced in this thesis can be used to solve sensor coverage problems, but since the focus of the thesis is network reliability this concept will not be addressed further.

The network coverage problem should not be confused with the concept of *fault coverage*, which is the probability that a faulty component can be covered by functioning components, allowing the system to recover from the fault [50]. Fault coverage has been addressed in the literature, with both binary [48, 50-53] and multi-valued [54] decision diagrams being used. The extension of the algorithms presented in this thesis to fault coverage is beyond the scope of the thesis, but is discussed in Chapter 6.

2.7. Computing Network Reliability and Performability

Algorithms to compute K-REL have been known since at least 1958 [55]. Like 2-REL, interest increased in the 1970s with a number of approaches proposed [56, 57]. Wood [55] proposed a unified mathematical framework regarding factoring algorithms to solve K-REL, and Bienstock [58] produced a number of ways to speed up the computation of K-REL using lattice theory. Yeh, Lu and Kuo [17] use a three-step process to compute K-REL, first using a heuristic approach to find a good variable ordering.

There are a number of different approaches to computing network reliability and performability: state enumeration, decomposition, inclusion-exclusion, the sum of disjoint products (SDP) [10], factoring [10], approximation, OBDD [30], and Boundary Set (BS) [15, 18]. Except for OBDD and BS, each of these is briefly introduced in this section, and more details of their applications to the various network reliability problems are discussed in Sections 2.7.1 to 2.7.6. OBDD is a factoring algorithm, which is discussed in Section 2.8 and the BS is an efficient OBDD algorithm discussed in Section 2.9.

2.7.1. State Enumeration

The simplest approach, state enumeration, is to generate all possible states of the network, and test each for meeting connectivity constraints. For a network with n fallible components, each of which can be either functioning or failed, there are 2^n possible states. Although this approach can be used to solve any of the network connectivity models, including 2-REL, K-REL, and ALL-REL, it is infeasible for networks of even moderate size. It can also be used to solve corresponding performability problems, such as EMD.

2.7.2. Decomposition methods

Decomposition methods [12] work by decomposing the network into smaller sub-networks which are more easily solved [59]. For example, Tanguy uses decomposition on the Brecht-Colburn ladder [9], generalized fan [9] and directed ladder networks [7], and then uses transfer matrices to find the reliability of a network. Deng and Singh [60] use decomposition to compute the reliability of inter-connected power systems. These approaches are useful for these specific networks, but there is a lack of efficient decomposition algorithms for general networks. The decomposition method has been used for 2-REL [7, 9, 12] and ALL-REL [9].

2.7.3. Inclusion-Exclusion

Another approach is to use the inclusion-exclusion principle, or its improved version [61]. However, this requires the enumeration of all the success and failure sets, and these are exponential in the size of the network [55]. While this is an improvement on state enumeration, the approach is not useful for large networks. The improved version of Inclusion-Exclusion has been used to solve both 2-REL and *K*-REL [61].

2.7.4. Sum of Disjoint Products

The Sum of Disjoint Products (SDP) method generates the paths and/or cuts of the graph, and then uses Boolean methods to make them mutually disjoint [10]. The probability of each disjoint path can then be summed to obtain the appropriate reliability. Unfortunately, the number of paths and/or cuts of a graph may be exponential in the number of fallible components (*e.g.*, the 2×100 grid network has 2^{99} paths under Model 'e'), although many SDP methods require only one or the other, not both. In addition, a network that has a very large number of paths may have a relatively small number of cuts, or vice versa. However, the process of making the paths and/or cuts mutually disjoint requires modifying each with respect to all others, which becomes impractical for large networks. If it is known that a network has a relatively small path or cut set, SDP is a good approach to use, but this is not generally the case.

Soh and Rai introduced the program CAREL [10], which used the COMpare, REDuce, CoMBine, and GENerate operators to convert paths or cuts into their SDP forms. Chaturverdi & Misra [14, 62] improved upon CAREL by modifying the operators used and combined it with the KDH88 method [63].

Soh *et al*. [64] proposed an efficient technique which relies on pre-sorting paths before applying SDP [10] to compute the EHC of a network. The sorting is necessary to meet the shortest path first routing scheme.

2.7.5. Factoring Methods

Factoring methods were introduced by Satyanarayana and Wood [65], and successively decide the state of an edge or vertex until the state of the network is fully determined [16]. The strength of factoring methods is that they perform reductions on network states in polynomial time, while preserving the desired properties of the network, such as reliability. The reliability of the network can be calculated directly from the final graph. This procedure corresponds to the construction of a binary computational tree [16]. AboElFotoh and Colbourn [3] used a factoring approach to find bounds for the reliability of radio broadcast networks with node failures and either perfect or fallible edges. Karbash and Wang [11] produced a tree-based method for computing 2-REL for a wireless ad-hoc network.

Factoring methods run in polynomial time on certain graphs, such as series-parallel ones. It is an exponential method, but is made efficient by performing simple and polygon-to-chain reductions [65], which decrease the size of a graph while

preserving its reliability. The factoring process of transforming a graph to a single edge or vertex occurs in linear time, but must be done an exponential number of times to calculate the network reliability.

Zang, Sun and Trivedi [31] have combined the factoring and SDP approaches, using a factoring method to obtain and store the minpaths/mincuts for SDP. This method was shown to be better than the SDP approaches available, although it has not been extended further.

Satyanarayana and Chang [66] and Wood [67] have shown that factoring algorithms with reductions are more efficient than path or cut enumeration methods for computing network reliability. This was confirmed by the experimental works of Theologou and Carlier [68]. A class of factoring algorithms that have become increasingly popular is the use of Decision Diagrams, as discussed in Section 2.8.

AboElFotoh [33] combined a breadth-first-search with applications of the factoring theorem to calculate EHC(*s*,*t*), however the solution does not scale well with large networks. AboElFotoh, Iyengar and Chakrabarty [32] later extended this approach to solve EHC(1-of-S,*t*) (Model 4.1).

Recently Li, *et al*. [34] proposed an approach using an acyclic directed graph to solve EHC for multi-state networks. This uses a factoring approach, but unlike the OBDD it does not take advantage of graph isomorphism to reduce redundant computations. Hence, this algorithm is only suitable for very small networks [34].

2.7.6. Approximation Methods

Ball [28] first showed that network reliability is NP-Hard. AboElFotoh and Colbourn [3] showed, further, that reliability is #P-Complete, which is the class containing problems that involve counting the accepting computations for problems in NP. Hence it is unlikely that an efficient (polynomial-time) general solution can be found for these problems.

Due to the difficulty of finding an exact answer, work has been done to find good approximations. Noltemeir and Wirth [69] suggested using such approximation algorithms to assist with the network design and improvement problems. Ramirez-Marquez, Coit, and Tortorella [70] use Monte-Carlo optimization to compute the reliability of a multi-state system. Marseguerra *et al.* [71] use the Monte-Carlo approach to provide an objective function for a Genetic Algorithm for a multi-objective network design problem. Elmallah and AboElFotoh [72] introduced a Circular Layout Cutsets approach to find reliability bounds in polynomial time.

Li and Silvester [73] use partial state space enumeration (see Section 2.7.1) to approximate solutions for 2-REL and EHC; the paper claims to address EMD, but the assumptions used only suffice for EHC. Brooks, *et al*. [74] use random graph models to approximate EHC(*s*,*t*) in mobile WSN, but assume fallible edges and perfect vertices. The Recursive Truncation Algorithm by Sharafat and Ma'rouzi [19] returns an approximation of ALL-REL from the list of all cutsets.

2.7.7. Section Summary

A number of solutions to REL, EHC and EMD have been discussed. None of these solutions apply to more general networks such as Models 3.3, 4.3 or 5, and all apply only to one (or at most two) component failure models.

While BS [15, 18] can compute REL for large (*e.g.*, the 4×40,000 grid network) undirected networks, under Model 2.2e, no approaches can compute REL for large general networks. No approaches can compute EHC [32, 33, 64] for large networks. Both SDP and factoring based techniques require generating (*s*,*t*)-minpaths, and thus, are not feasible for computing the EHC of networks with large number of paths (*e.g.*, the 2×100 grid network that contains 4.6×10^{20} paths for Model 'v').

The factoring algorithm proposed by AboElFotoh, Iyengar and Chakrabaty [32], is shown to be superior in generating all paths for all network states, but still requires the generation of a path of minimum length for each network state. For networks of even moderate size, especially those with large numbers of paths, this approach is not practical.

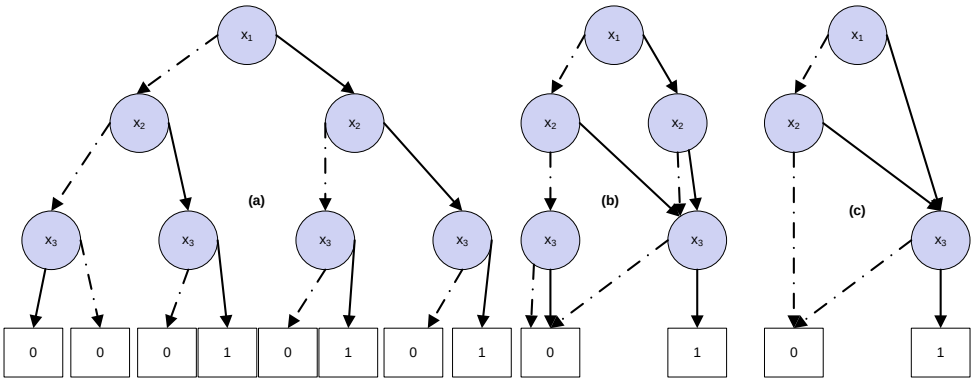
2.8. Ordered Decision Diagrams

2.8.1. Ordered Binary Decision Diagrams

The OBDD, developed by Bryant [75, 76], is an "ordered" version of the Binary Decision Diagram (BDD) introduced by Lee [77], which was reintroduced by Akers [78] in the form that is now commonly used. An OBDD is an efficient representation of a Boolean function, with each node representing the decision/evaluation of a Boolean variable. Figure 2.3(a) shows an OBDD which represents the Boolean function $(x_1, x_2, x_3) = x_1x_3 \cup x_2x_3$. OBDDs have been used for a number of applications, including network reliability, circuit simulation [77, 78] and design verification [79]. An efficient implementation of OBDD has been produced by Brace, Rudell and Bryant [80].

In this thesis, the term 'node' is used to refer to each OBDD node, and 'vertex' for the graph representation of the devices of a network. Similarly, 'link' is used for a connection between two OBDD nodes, and 'edge' for a connection between two vertices in a graph. When presenting an OBDD, round nodes are used to represent variables being evaluated, with the variables being shown either in each node, or on the left of that level of the diagram. Each round node of an OBDD has exactly two children, and is referred to as a *non-terminal node*. Square nodes represent the leaves of a diagram and are called *terminal nodes*; their content represents the output of the function being evaluated. For an OBDD, this will generally be 0 and 1. In this thesis, the terminal nodes contain the solutions to problems (*i.e.*, REL, EHC or EMD values).

For a non-terminal node evaluating variable x_i, we show a solid link to the child representing x_i=1, called the *positive child*, and a dashed or dotted link for the *negative child* that represents x_i=0. Where possible, we place the negative child to the left of the parent node and the positive child to the right.



(a) BDD for $x_1x_3 \cup x_2x_3$ (b) Merging Isomorphic nodes and (c) Removing redundant nodes

Figure 2.3: Sample OBDD from [76]

Each level of the OBDD represents the decision of one variable. OBDDs are based on the Shannon Expansion $f = x \times f_{x=1} + \overline{x} \times f_{x=0}$ and have a number of well-defined operations [76] that allow them to be modified or combined. These include AND, OR and NOT. In addition to the elementary operation of creating an OBDD representing a single variable, these can be used to efficiently encode any binary function. The nodes of the OBDD do not contain any information, other than what is inherent through their position in the diagram. To use an OBDD to calculate the value of the binary function it represents, the diagram is parsed upwards from the terminal nodes, which requires first creating the diagram, and then traversing it fully again.

The variables in OBDD are evaluated in the same order for each diagram path from the root node (*e.g.*, x_1 in Figure 2.3) to the terminal nodes 0 and 1. In contrast, a BDD (also called free BDD[9]), does not require variables to be evaluated in the same order for each path from the root node to the terminal nodes. Having a consistent ordering makes it more likely that identical sub-trees can be found, and decreases the number of nodes in the diagram. Section 2.8.2 discusses variable-ordering algorithms.

Reducing the number of OBDD nodes is important for reducing both time and space complexity. The limiting factor of several algorithms [15, 18] is the storage space. Hence, methods that reduce the number of nodes generated are especially important. Bryant [76] identifies three ways for doing so. Firstly, a parent node that has its right and left children leading to the same child node (*e.g.*, the right-hand x_2 to the right-hand x_3 node in Figure 2.3 (b)) can be omitted from the diagram since the evaluation on the variable, in this case x_2, has no effect on the sub-tree; Bryant [76] refers to this step as removing redundant diagram nodes. Figure 2.3 (c) shows an identical but smaller sized OBDD to that in Figure 2.3 (b), and hence, requires less storage space and less time to traverse. If nodes can be removed from the diagram before being generated, then the amount of time to generate the diagram is also reduced. Secondly, terminals that contain the same value are merged, since they are identical. For example, the five negative and three positive terminal nodes in Figure 2.3 (a) can be merged into only one positive and one negative node, as shown in Figure 2.3 (b).

Thirdly, Bryant [76] asserts that nodes with identical sub-diagrams are isomorphic and can be merged. This is the case with the rightmost three nodes deciding variable x_3 in Figure 2.3 (a), which are merged into one x_3 node in Figure 2.3 (b). This step provides a great reduction in the number of nodes, with the related reduction in storage space and time to traverse. Each sub-tree of such a node only needs to be generated once, likewise reducing the time taken to generate the diagram. Utilizing isomorphism between nodes is one of the strengths of OBDD algorithms, because it prevents sub-trees from being redundantly re-evaluated. Further, when the evaluation of a particular variable (node) does not affect the sub-tree, the redundant node can be removed. This third reduction rule is the main one used by the OBDD-A and OBDD-H algorithms, as described in Sections 3.2.6 and 4.2.4.

An OBDD that has been reduced is referred to as a Reduced OBDD (ROBDD). It is general practice in the literature to reduce an OBDD whenever possible, and

hence the ‘Reduced’ designation is generally omitted. For the remainder of this work, it is assumed that all OBDDs discussed are reduced where possible. It can be shown that reduced OBDDs are a canonical form of the Boolean function that they represent [81].

2.8.2. Variable Ordering

The ordering of the variables [76] affect the possibilities of reduction, and thus, is important in reducing the size of the OBDD. While it would be beneficial in many cases to find the optimal ordering of variables for an OBDD, the process for doing so is NP-Complete [82]. For this reason, a variety of heuristic methods are generally used when working with OBDDs. Drechsler, Becker, and Gockel [81] apply a genetic algorithm to find a variable ordering for minimizing the size of an OBDD. The authors report that their algorithm finds the optimal ordering for most of their benchmark networks, and can be applied to networks with more than 20 variables, due to its speed. This works by mutating the OBDD through exchanging pairs of adjacent variables in the ordering using roulette wheel selection. Yeh and Kuo [83] use a divide and conquer approach to order clusters of variables. This method is applied to circuit design. Unfortunately, these methods are designed to optimize existing OBDDs, and do not help in optimizing one that is undergoing creation.

For the application of OBDDs to Network Reliability (described in Section 2.8.5) and related fields, the OBDD needs to be reduced as much as possible to optimize the reliability computation. The ordering of variables should either be determined before execution begins or optimized during pre-processing with a minimum of incurred cost. A good general breadth-first ordering has emerged in the literature [30]. This thesis uses a comparable breadth-first ordering, as discussed in Section 2.10.4.

Another way to reduce the size of an OBDD is to use inverted edges [4], which means inverting the truth values of the negative child. OBDDs with inverted edges are around 7% smaller, in terms of number of nodes, than those with normal edges, and the use of OBDD manipulation functions is sped up by 100% [4]. This makes the technique useful for OBDDs that will be manipulated using the basic functions [76]. The OBDD-A method introduced in Chapter 3 does not manipulate the diagram using basic functions, since the diagram is not actually stored as a whole, and hence, the use of inverted edges is not applicable.

2.8.3. Ordered Multi-valued Decision Diagrams

A Decision Diagram with multiple-valued outputs was introduced by Cerny, Mange and Sanchez [84] to better model switching circuits with non-binary outputs, although initial findings indicated that OBDDs would generally be more efficient [85]. Srinivasan *et al*.[86] formalized these to produce an Ordered Multi-valued Decision Diagram (OMDD) following the work on OBDDs by Bryant [76], and similar work later carried out apparently independently by Kam [87]. Since the OMDD is essentially an OBDD without the restriction that each node has only two children, it is a more general form [88]. An OMDD can be homogeneous, when the variables being evaluated on each level have the same number of outcomes, or heterogeneous otherwise.

OMDDs are useful for problems where each variable has multiple values [84, 85], but can also be used for binary functions, since a number of binary variables can be grouped together into a single OMDD variable. While this decreases the number of variables needing to be processed, it greatly increases the number of child nodes per node processed, and hence will increase the number of nodes per level [88].

It was noted [88] that evaluating a logic function with OMDDs is theoretically orders of magnitude faster than with OBDDs but, in practice, the amount of memory required becomes the dominant consideration in large designs. A Quick Reduced MDD (QRMDD) was presented by McGeer *et al*. [88] to take advantage of the processing increase, while using reduction to minimize the increase in the number of child nodes processed. The speedup is largely due to the diagram being stored as a lookup table; this thesis uses a similar (hash table) technique to speed up diagram access.

As with finding the optimal ordering of variables for an OBDD, finding the optimal ordering for an OMDD is an NP-Complete problem. If the OMDD consists of groupings of binary variables, these groupings must likewise be optimized. With suitable optimization, the size of an OMDD has been shown to be between 86% and 67% of that of the comparative OBDD [89], when ordering both the variables in a grouping and the groupings themselves. However, this study is performed on DDs that have been fully created, and therefore, the results do not apply to the initial number of nodes created.

OMDDs have been applied to logic simulation [85, 88], system verification [90] as well as routing, resource scheduling and the graph colouring problem [86]. An efficient OMDD implementation [91] has been produced. Xing and Dugan [54] use

OMDDs for the analysis of network faults, and Zaitseva and Levashenko [92] apply OMDDs to the computation of dynamic reliability indices for estimating the reliability of a multi-state system. The only works applying OMDDs to network reliability and performability are the work of the author [21, 35, 93]. The OMDD-A is discussed in Section 6.2.3.

2.8.4. Other Decision Diagrams

A number of BDDs exist, other than the OBDD or OMDD. For example, the Ordered Functional DD (OFDD) [94] is similar to the OBDD but uses the AND/XOR operations instead of the AND/OR operations. The OFDD is shown to be similar to the OBDD in many ways, but to have an exponential worst-case representation size for some functions, while performing better for others [94]. The Ordered Kronecker Function DD (OKFDD) is a more general case of both the OBDD and the OFDD. In fact, it is shown [95] that the OKFDD represents the only two decompositions that help reduce the size of the diagram. Although some work has been undertaken to optimize OKFDDs [96, 97], they are not in common use. Other BDDs, such as Free BDDs and Repeated BDDs, are discussed in [98]. This thesis considers only the OBDD and OMDD.

2.8.5. OBDD for Network Reliability and Performability Evaluation

OBDD-based approaches have been used to compute 2-REL. In the application of the OBDD technique to reliability, each variable represents a vertex v_x (or an edge e_y), which is either available with probability $Pr(v_x)$ (or $Pr(e_y)$) or failed with probability $1-Pr(v_x)$ (or $1-Pr(e_y)$). The probability that the network is connected is then given by tracing paths upwards from the success terminal nodes, and multiplying the reaching path probabilities by the probability for a positive or negative sub-tree. Since each traversed path represents a disjoint event, the probability of each such path is summed to give the network reliability.

BDDs were first applied to (un)reliability by Doyle and Dugan [50], who used their DREDD algorithm for the computation of the coverage of a network. This work was extended [52, 54] to the use of multiple-valued DDs for fault coverage.

Yeh and Kuo [30] first applied OBDDs to 2-REL (Model 1e). The results were extended by Kuo, Lu and Yeh [4] with their EED and EED-ISO algorithms. These algorithms first decompose the network into an OBDD, and then process each node of the OBDD to generate 2-REL. The EED-ISO algorithm compares favourably with other methods [99] that were available at the time.

The EED-ISO algorithm was extended [17, 26] to the EED-BFS algorithm, which allows for vertex failure (Model 1ve) by using the incident edges method, including the probability of vertex failure in each edge failure probability. This requires extra processing, but is far more efficient than including the nodes as additional variables in the OBDD, however, the effect of including the probability of vertices multiple times in a supposedly disjoint equation is not addressed. The algorithms are also extended [17, 26] to solve the K-REL (Model 2.1) problem. For EED-BFS, K-REL is defined as the chance that a source vertex is connected to |K|-1 target vertices, which requires that the network be undirected. The algorithm is run for each source-target pair and the results are combined using Boolean AND expressions on the OBDDs that result. This is extremely computationally expensive, since it requires |K|-1 full executions of the 2-REL algorithm and then |K|-2 merge operations between (possibly large) binary diagrams. Despite these inefficiencies, EED-BFS is able to solve in reasonable time the 2×100 grid and other moderately sized networks that are difficult to solve using non-BDD methods, such as SDP.

EED-ISO was also expanded [6] to CUT-BDD, which generates cuts instead of paths for 2-REL. As discussed in Section 2.7.4, the number of cuts may be substantially smaller than the number of paths for some networks. CUT-BDD is shown to be competitive with EED-ISO on much of the test set, and considerably better on the rest. However, it is not clear whether any networks in the test set have considerably less cuts than paths; the method is expected to perform better on such networks, but worse on those with more cuts. CUT-BDD also calculates the Birnbaum importance measure [6] of the fallible edges of the network, which can be an aid to network improvement.

Lin, Kuo and Yeh [25] produced an alternative algorithm to calculate REL by generating the cutset of a network using recursive merging and OBDD. It was argued that the cutset is required for a number of applications. Good results were returned for some networks, but the method is not feasible for networks with large cutsets. Chang, Lin, Chen and Kuo [6] introduced an approach that uses an OBDD to represent the cut function of a network in order to compute the importance measures of network components.

Hardy, Lucet and Limnios [15, 18] introduced a combination of OBDD and BS to calculate K-REL and ALL-REL for undirected networks. The performance is good, even for extremely large networks, although only for Model 2.2e. This work is

discussed further in Section 2.9. Recently, Javanbarg *et al*. [5] applied this algorithm to computing the reliability of infrastructure networks with good results.

Before the work leading to this thesis [21, 35], OBDD algorithms had not been used to calculate EHC or EMD. They have, however, been used to compute fault-coverage [48, 50-54] and failure frequency [100, 101].

2.9. Boundary Sets

2.9.1. Model and Notation

Carlier and Lucet [16] introduced BS as a decomposition method for solving K-REL (discussed in Section 2.5.3) for undirected networks. The method identifies the *active vertices* of each network state; that is, those vertices for which the decomposition has at that stage gained some information, but has more information to gain. Note that the active vertices, and hence, the BS, will be identical for all nodes on one level of an Ordered Binary Decision Diagram.

Carlier and Lucet [16] define a BS as follows:

> "A sub-network of G = (V, E), is a network H = (V', E') such that $V \supseteq V'$, and $E' = E \cap (V \times V')$. Let H be any sub-network of G, and L = (V", E") the complement graph of H in G (*i.e.*, $V' \cup V" = V$); $E' \cup E" = E$ and $E' \cup E" = E$. Then, the set of vertices $F = V' \cap V"$, is called the *boundary set* of H." (p. 144)

While this definition is mathematically exact, it is not ideal for the application of BS to network reliability. In particular, its application to the use of OBDDs is not clear. A more applicable definition is given by Hardy, Lucet and Limnios [18]:

> "Consider that $E_k = \{e_1, e_2, \dots, e_k\}$ and $\overline{E_k} = \{e_{k+1}, \dots, e_n\}$. The graphs in the k^{th} level of the BDD are sub-graphs of G with the edge set $\overline{E}_k$. For each level *k*, we define the *boundary set* F_k as a vertex set, such that each vertex of F_k is incident to at least one edge in E_k and one edge in $\overline{E}_k$."(p. 1471)

This definition is relevant, but only accounts for edge failure, and does not suffice for defining the initial BS, F_0. This thesis presents a more general definition in Section 4.5.2.1 that applies to edge failure, vertex failure or both, for all levels of the diagram. BS must record any connections (via paths of active edges and vertices) between the active vertices. For a network that does not require full

connectivity, the algorithm must also record whether each active vertex is connected to any of the source vertices in |S|.

The interconnectivity of active vertices is encoded by a partition of F_k. Two vertices are in the same block of the partition if, and only if, they are connected to each other. For a network that does not require full connectivity, a block containing vertices connected with one or more of the target vertices is marked. Such a mark is denoted by an asterisk to the right of the block. Note that since BS applies only to undirected networks, connections between vertices are always undirected.

For example, the BS of level 2 of the OBDD for the network given in Figure 2.1 (p. 7) is $\{v_1,v_2\}$. If $s = v_0$ and $t = v_3$ the possible ways of partitioning F_2 are $[v_1\ v_2]^*$, $[v_1]^*[v_2]$ and $[v_1][v_2]^*$. Any partition that has no marked blocks (*e.g.*, $[v_1][v_2]$) has no connection with v_0 and therefore is failed. No partition can have two blocks marked, since that would imply both are connected to v_0, and hence, each other.[4]

In the work by Hardy, Lucet and Limnios [15, 18], each partition of F_k is implemented by a vector of size $|F_k|$, where the i^{th} position in the vector contains the number of the block containing $F_k[i]$. For a network that does not require full connectivity, an additional variable records which block is marked. For example, the partitions $[v_1\ v_2]^*$, $[v_1]^*[v_2]$ and $[v_1][v_2]^*$ are represented as ([1 1],0), ([1 2],0) and ([1 2],1), respectively. For ALL-REL, the marking of blocks is not required, and hence, the partitions $[v_1\ v_2]$ and $[v_1][v_2]$ would be implemented as ([1 1]) and ([1 2]), respectively.

Each partition of a BS can be associated with a distinct number; enumerating partitions allows a number to be stored instead of the full partition, and also simplifies tests of equality between partitions. The enumeration of partitions makes use of Stirling numbers of the second kind [15]. These are calculated using $A_{i,j} = j \times A_{i-1,j} + A_{i-1,j-1}$ for $1 \le j \le i$, with $A_{i,1}=1$ and $A_{i,j}=0$ if $i<j$. This ordering is described in [16] and [15, 18], and applies only to ALL-REL. It does not suffice for the general case of REL, because it does not consider marked partitions. An enumeration which applies to all REL (and related) metrics for component failure Model 'e' was introduced by Herrmann and Soh [102].

[4] If directed edges were permitted it would be possible to have more than one marked partition since the connections from v_0 would not necessarily imply connections to each other through v_0. Similarly, some problems have multiple sources or also mark blocks connected to a target vertex; such problems can have multiple blocks marked in a partition.

2.9.2. Boundary Sets for Network Reliability and Performability

In 1996, Carlier and Lucet [16] used a decomposition method utilizing BS to solve K-REL for undirected networks. The method was extended by Hardy, Lucet and Limnios [18] to compute ALL-REL and then K-REL [15] using a combination of OBDD and BS, with greatly improved results. The implementation was reported to be able to solve ALL-REL for the 4×40,000 grid network in around 5 minutes. Tittman's analysis [103] of the relationship between ALL-REL and K-REL uses a similar partitioning approach, but appears to be independent.

Carlier and Lucet [16] introduced an exact algorithm for computing ALL-REL and K-REL for undirected networks with edge failure and perfect vertices. The algorithm for K-REL is shown [16] to have space complexity $O(\mathrm{T}(\mathrm{F}_{max}))$, where F_{max} is the size of the maximal BS $T(i) = \sum_{j=1}^{i} |A(i,j)|$ and all $A(i,j)$ are Stirling numbers of the second kind [15], and a time complexity of $O\left(\sum_{p=1}^{N-1}\left(|\mathrm{F}_k|\sum_{j=1}^{|\mathrm{F}_k|}\left(|A(|\mathrm{F}_k|,j)| \times 2^{2j}\right)\right)\right)$. An application to the case of failed vertices is given with a time complexity of $O\left(|\mathrm{F}_k|\sum_{p=1}^{|\mathrm{F}_k|}\left(C_p^{|\mathrm{F}_k|}\sum_{j=1}^{p}\left(|A(p,j)| \times 2^{2j}\right)\right)\right)$. Note that this is a further exponential increase compared to the case of perfect vertices. The limiting factor of the algorithm is the space complexity [16] (p.153). The algorithm is shown to be faster than existing factoring algorithms.

Hardy, Lucet and Limnios [18] add the calculation of the Birnbaum importance measure [6] to the algorithm from Carlier and Lucet [16] to determine the degree of contributions to network reliability of each edge. This work is extended [15] using an edge contraction/deletion approach, and the complexity is evaluated as being bounded by $O\left(m \times \mathrm{F}_{\max}^3 \times \mathrm{B}_{\mathrm{F}_{\max}}\right)$ for K-REL and $O\left(m \times \mathrm{F}_{\max} \times \mathrm{B}_{\mathrm{F}_{\max}}\right)$ for ALL-REL, where $\mathrm{B}_{\mathrm{F}_{max}}$ is the *Bell number* defined by $\mathrm{B}_{\mathrm{F}_{\max}} = \sum_{j=1}^{|\mathrm{F}_k|} A(\mathrm{F}_k, j)$. The method is shown to be faster than the BDD approach used by Yeh, Lu and Kuo [17]. The method given computes K-REL by first computing 2-REL for every $\mathrm{v_s} \in \mathrm{K}$ separately, and then combining the results. Once again, the limiting factor is the amount of nodes that can be practically stored for the computation. Connected networks are the most difficult to compute for this approach [15].

Since it is a special case, methods for K-REL also apply to ALL-REL. The BS approach [15, 18] is more efficient for ALL-REL than K-REL, although this is not generally the case for other methods.

2.10. Network Topologies Used in this Thesis

This section discusses the specific networks used in this thesis; why they were chosen and how they were generated and sorted. Although these details are not required for an understanding of the algorithms that form the main body of this work, including the explanation makes it easier for this work to be reproduced.

Each network in this work was chosen for one or more of three reasons; networks from other scholarly work are used to compare the algorithms produced herein with those of the work, other networks were chosen to highlight particular features of the algorithms presented in this work, and still others were chosen specifically for use in detailed examples.

This section first discusses the networks used and the reasons for their use and then discusses the ordering used to label the network vertices and edges.

2.10.1. Networks Used

The networks from other works are primarily those from Hardy, Lucet and Limnios [15]. This is the main work used for comparison with our algorithms, and many of these networks are also found in other relevant works [4, 10]. In particular, we include networks path18 and path19, which are present in all three works. These are relatively small undirected networks which have been part of the reliability literature for some time. While they are not challenging for modern reliability algorithms, they serve as a useful comparison with some of the older methods.

Recent work on network reliability utilizing OBDDs [4, 15, 17, 18, 26, 30] has made extensive use of grid networks, since these networks are particularly well suited for this approach. While the main grid used is the 2x100 network, this thesis also includes a number of others found in these works, and several others that allow comparison of performance changes as network size increases.

Since OBDDs are best used on networks with relatively low inter-connectivity (*i.e.*, each network device is connected to as few other devices as possible) it is useful to test such algorithms on networks with maximal inter-connectivity. This thesis follows Hardy, Lucet and Limnios [15, 18] in using fully connected networks for this purpose. In order to gain information on how the algorithms in this thesis function for networks of constant inter-connectivity by varying size, use is made of the *w*-connected networks described in Section 2.10.3. Finally, some of the above works make use of two cyclic grid networks. These two networks (2×6 and 2x8) have been included in this work for the sake of completeness.

2.10.2. Grid Networks

Grid networks are those laid out in a grid; that is, where each device that is not at the edge of the network is connected to its nearest neighbour in each of the four cardinal directions. The devices on the edge of the network are connected to their nearest neighbours in those directions where other devices exist. Some examples of grid networks are shown in Figure 2.4. Streets may be arranged as a grid network [16], as may the power distribution, water and communication systems that follow those streets.

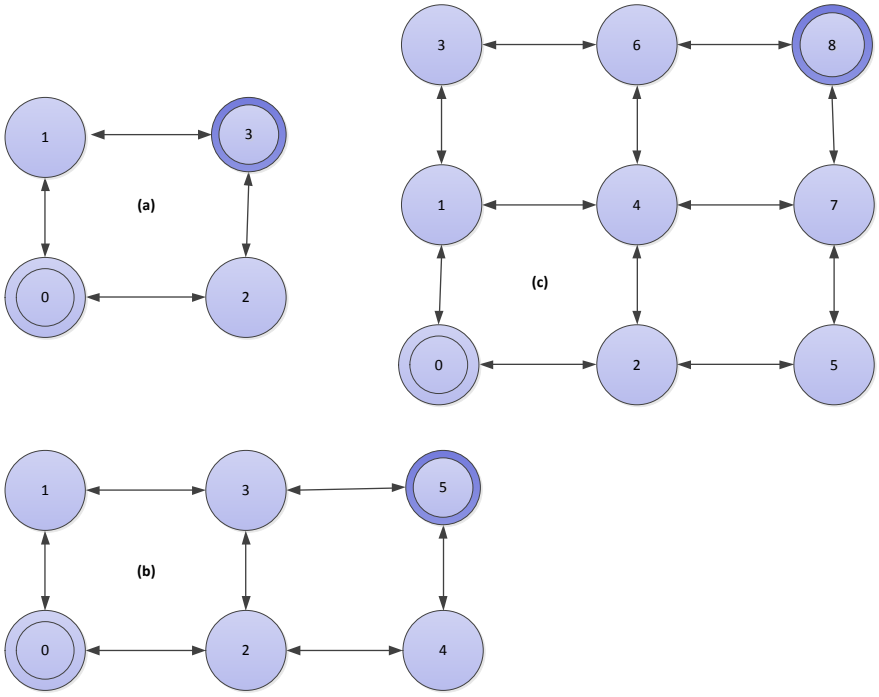


Figure 2.4: Graphs Representing (a) 2×2 (b) 2×3 and (c) 3×3 Grid Networks.

Grid networks are defined by their height and width, both measured in the number of devices. A grid network that is 5 devices high by 10 wide is referred to as a 5×10 grid, and written Grid5×10. Because grid networks have a large degree of redundancy, they are extremely robust when faced with component failure. This work uses the convention of placing the first source vertex in the bottom left-hand position, and the target vertex in the top right-hand position. Where more than one of each exists, their positions are specified.

Grid networks are commonly used for testing reliability algorithms based on Decision Diagrams [4, 6, 15-17, 21, 22, 25, 26, 30, 102], although they are also used for some non-Decision Diagram methods [6, 25], because their regular structure allows them to be easily extended in any dimension without changing the connectivity properties between the devices. This avoids the unpredictability

introduced when generating very large networks using random methods. In addition, narrow grid networks (*i.e.*, those with low height or width) are especially easy to analyse with Decision Diagram algorithms, making them attractive for performance measurement. This is discussed in Sections 3.5 and 3.6.

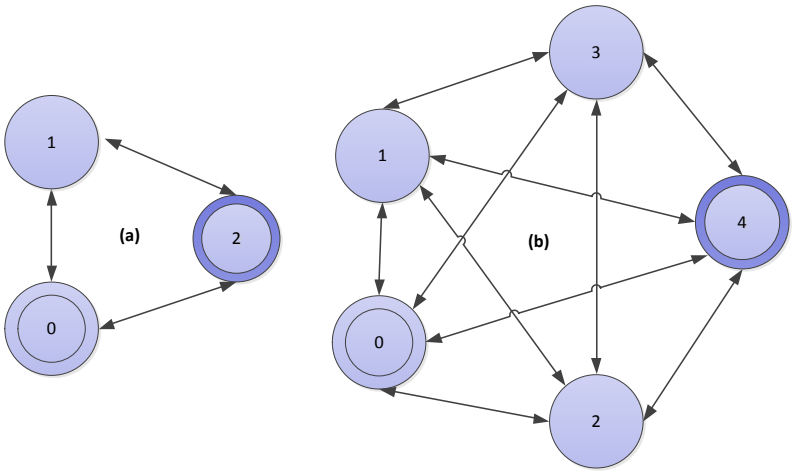


Figure 2.5: Networks (a) K3 and (b) K5

2.10.3. Fully Connected and w-Connected Networks

Another class of networks with known connectivity structure and easy extendibility are the connected networks. Fully connected networks are those where every device is directly connected with every other device in the network. This thesis uses a similar notation to that of [15], writing a fully connected network of *n* devices as K*n* (as compared to K_n in [15]). Fully connected devices have been used in reliability literature [4, 6, 15, 17, 25, 26, 102], although their high connectivity makes it challenging to solve networks of even moderate size. Networks K3 and K5 are shown in Figure 2.5.

W-connected networks are those where every device is connected to the *w*-1 next devices (and the *w*-1 previous devices in an undirected network). A *w*-connected network of *n* devices is written as K*w,n*. Note that network K*n,n* is equal to K*n* and hence fully connected networks are a special case of *w*-connected networks. *W*-connected networks were introduced by Herrmann and Soh [102] to study certain properties of algorithms on networks of increasing number of devices, but with constant connectivity structure. Figure 2.6 shows three different 3-connected networks.

Note that the K3,*n* networks are equivalent to the Brecht-Colbourn ladder network [9]. In addition, a K*w,n* network is equivalent to an interval graph [64] that has both the intervals and space between vertices constant.

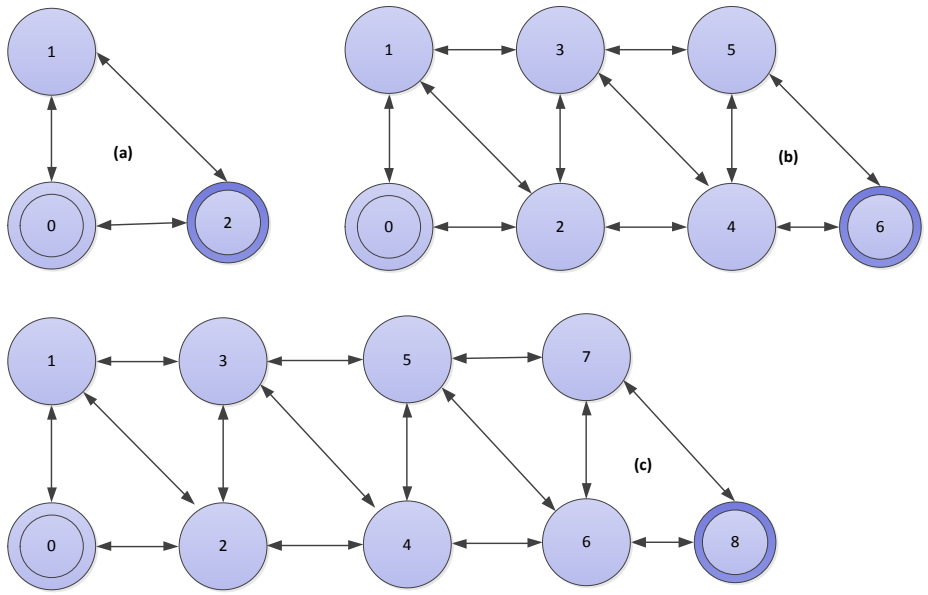


Figure 2.6: Networks (a) K3,3 (b) K3,7 and (c) K3,9

2.10.4. Variable Ordering

The ordering of the variables of the network affects the performance of algorithms used on that network. For example, the variable ordering affects the number of nodes generated by an OBDD, as discussed in Section 2.8. This thesis focuses on using OBDD-based approaches, so the ordering used is designed with this in mind.

Finding an optimal ordering is computationally expensive, and may not be possible until the full diagram has been generated. Hence, the OBDD reliability literature uses orderings that have been found to combine good performance with an efficient way to generate the ordering. The most common ordering is breadth-first [4, 15, 17, 18, 26, 30, 31]. The sorting algorithm is sometimes specified in detail [17, 26, 30, 31], and sometimes not [4, 15, 18].

The breadth-first ordering used in this work is achieved by labelling all sources from 0 to |S|-1, and then progressively labelling all other vertices based on their distance from the closest source. Edges (v_f,v_t) and $\{v_f,v_t\}$ are labelled in increasing order of MIN(f,t) and then MAX(f,t). Note that the ordering of vertices generally results in directed edges (v_f,v_t) with $f<t$. Since we can write undirected edges $\{v_f,v_t\}$ with the endpoints in any order, we always write them with $f<t$. Networks do not have a unique ordering, since two vertices that are equidistant from the source may be legitimately swapped in the ordering.

This work uses a Perl script to sort each network file, using the ordering given above. The script first places all sources onto a FIFO queue. It then recursively removes an element from the queue, labels it with the next available number, and

adds all unlabelled vertices connected to this vertex to the end of the queue. All edges (v_f, v_t) are then sorted according to v_f and v_t. Note that the ordering produced by this script relies on the ordering of the original network file.

The ordering described attempts to minimize the *width* of the network, defined as $\text{MIN}_{(x,y)\in E}|x-y|$ for the graph G=(V,E).

2.11. Implementation and System Details

Each algorithm proposed in this thesis has been implemented, and results based on this implementation are given following the algorithm descriptions in Chapter 3 to Chapter 5. These discuss the performance of the implementation on a range of benchmark networks, and compare and contrast its performance with the state of the art in existing solutions. This section gives the details of the implementation, testing and the methods used for the comparisons.

Each algorithm was implemented using Microsoft Visual Studio 2010 on a PC (i7 920 2.67GHz processors, 8MB L3 cache, 12GB RAM) running Windows 7. The implementation used for this thesis is the general version; previous implementations have existed and have since been consolidated. The details presented in a number of papers were made using a different server which is no longer available, and implementations that were optimized for the problem under consideration. Hence, some discrepancies exist between the results presented here and those in previous works such as [22, 104].

Networks were generated as discussed in Section 2.10. Once generated, all networks were pre-sorted using a Perl script that implements the ordering described in Section 2.10.4. The sorting of the network is not normally considered to be part of the computation time [15, 18], although sometimes its inclusion or exclusion is not made clear [4, 13, 17, 26, 31]. Indeed methods that require the generation of possibly exponentially large pathsets or cutsets may not even include the generation of those in their computation time [6, 30].

All networks were read from the text files into memory as part of the process; the generation and sorting of the networks is not included in the computation time given, but the reading-in of the files and output of the results to screen is. This is true for the implementations presented in this thesis; it is not necessarily true for results obtained from the papers of other authors. Hence, direct comparison with results published in other papers may not be fully accurate.

There are other reasons that direct comparison with published solutions can be misleading. While the vertex and edge ordering used in comparable works and the ordering used in this thesis are breadth first (described in Section 2.10.4), the exact ordering can still differ. For entirely accurate comparisons, both network orderings (not just ordering methods) should be identical. In addition, published results from other authors may utilize different coding methodologies (such as special techniques for comparison operations) that may affect the comparison of algorithms, as opposed to algorithm implementations. Finally, published results will invariably use a computer with different performance to the one used for this work.

For this reason it is beneficial to compare the results of our method with the results of other methods, using the same computer and similar programming methodologies on the same network data files. For this to occur, the source code from the authors of the algorithms, or these algorithms must be implemented independently.

The authors of the works used for comparison have been requested to provide the code for their solutions, but unfortunately, we have been unable to obtain the code for most of these. The exception to this is the code for the CAREL system [10], which this work uses to test the accuracy of the 2-REL answers produced by the presented implementation for smaller networks. Performance comparisons with CAREL are not given, since recent methods are more efficient.

The implementation used for this thesis has been designed for fair comparison between the different component failure and network connectivity models. In particular, code sharing between the solutions to different models is maximized, and heuristics that may improve performance have been avoided. Heuristics generally deal with specific cases, and thus, improve the performance of the algorithm for a limited selection of models. In sections where heuristics are used that differentiate from the main algorithm, the reasons and heuristics are discussed in detail. A number of heuristics that apply to the algorithms introduced in this thesis are discussed in Section 6.2.1.

2.12. Chapter Summary

The problems of network reliability and performability have been introduced, along with a number of measures to classify the problems. Models of both network connectivity and component failure have been described, as well as several performability metrics. Even for the simplest of these problems, computing the

two-terminal reliability for a network with perfect edges, is #P-Complete, meaning that general solutions are very likely to be exponential in both time and space complexity.

A range of existing solutions to the reliability and performability problems have been discussed. While a number of solutions exist for computing network reliability, the BS approach by Hardy, Lucet and Limnios [15, 18] has been shown to have a much better time complexity than other existing solutions, although the amount of memory required is problematic. This solution, however, is only applicable for undirected networks with infallible network devices (*i.e.*, Model 'e'); no solutions with matching performance exists that can compute the network reliability of directed networks whose devices (vertices) and communication links (edges) are both susceptible to failure.

A range of network topologies have been introduced; these will be used for the testing of the algorithms presented in this thesis. The details of this implementation and the system used for testing have also been listed, in order to clarify the experimental procedure. The first of these algorithms, the Augmented OBDD, is introduced in Chapter 3.

Chapter 3
Augmented Ordered Binary Decision Diagram

3.1. Chapter Overview

The Augmented Ordered Binary Decision Diagram (OBDD-A) is an OBDD that stores additional information in each diagram node. This information provides OBDD-A three main advantages as compared to the OBDD. Firstly, as described in Section 3.2.1, OBDD-A stores only two levels of diagram nodes at any one time. In contrast, other methods [4, 6, 17, 25, 26] need to keep all of the generated diagram nodes in memory so that they can be later traversed to calculate the network reliability (REL). Thus OBDD-A significantly reduces the required run-time memory for the algorithm as compared to the OBDD. Further, the simulations in Section 3.6 show that OBDD-A algorithm stores a constant number of nodes for families of networks with identical internal connectivity structure irrespective of the size of the network. This chapter focuses on introducing the OBDD-A algorithm and this benefit.

Secondly, OBDD-A is more flexible than the existing OBDD methods [21, 45, 104]. OBDD and other algorithms that compute REL and/or performability may be restricted in what networks (*e.g.*, undirected network, ladder network, *etc.*) they can be applied to. In addition, they can be used only for one or two connectivity models (*e.g.*, K-REL) and component failure models (*e.g.*, Model ‘v’). In contrast, OBDD-A can be applied to a wide variety of networks, with varying network connectivity requirements and component failure models.

The details of the OBDD-A algorithm for REL vary, depending on the component failure and network models being used. This chapter presents the OBDD-A for those with fallible communication links and perfect devices – Model ‘e’. Appendix A shows how to modify the OBDD-A for Model ‘e’ into the more general OBDD-A that can compute REL for Models ‘e’, ‘v’ and ‘ve’. However, this chapter includes the performance evaluation of the OBDD-A on all possible network models and component failure models. We refer to an OBDD-A for a network with fallible edges and perfect vertices as an Edge OBDD-A, written OBDD-Ae.

Similarly an OBDD-A for a network with perfect edges and fallible vertices can be written as OBDD-Av and one for networks whose edges and vertices are both fallible is written OBDD-Ave.

Finally, OBDD-A can store a variety of information in each node, which allows the tracking of more than just the network connectivity information available for OBDD. The stored information (*e.g.*, delay information) enables OBDD-A to compute performability metrics such as the Expected Hop Count (EHC) [21, 22, 45, 104] and the expected message delay (EMD) [35] of a network. The OBDD-A has been shown [21, 22, 45, 104] to compute exact solutions for REL and EHC but to be less useful for solving capacity-related metrics [45]. Further work has applied the OBDD-A to a number of REL models for WSN [22] and shown them to be extremely memory efficient [102][5]. The OBDD-A presented in this thesis improves the preliminary version in [101]. Chapter 5 focuses on calculating EHC and EMD.

This chapter is organized as follows. Section 3.2 introduces the mathematical model behind the OBDD-A for Model 1e, including the information contained in its nodes and node isomorphism. Section 3.3 presents the pseudo-code of the algorithm for Model 1e and Section 3.4 provides an example of computing REL using the OBDD-A. The complexity of the OBDD-A algorithm is analysed in Section 3.5 and Section 3.6 gives the results of performance testing on the implementation; both of these sections discuss the general OBDD-A, not just the one for Model 1e. The mathematical model and pseudo-code of the general OBDD-A is given in Appendix A. The chapter is summarized in Section 3.7.

3.2. Mathematical Model for OBDD-A

3.2.1. Nodes and Levels

Let $\Psi(N,G)$ denote the OBDD-A for graph G(V,E), where N is the set of OBDD-A nodes $\{N_0, N_1, \ldots, N_{2^{|E|}-1}\}$. An example OBDD-A, constructed from the example network in Figure 3.1 for communication Model 1e, is shown in Figure 3.2. Without loss of generality, let N_0 be the root node and for any parent node $N_i \in \Psi(N,G)$, let N_{2i+1} be the left (or negative) child node and let N_{2i+2} be the right (or positive) child node of N_i, for i=0, 1, 2, … . This notation clarifies the relationship between nodes. Figure 3.2 illustrates the node numbers for the

[5] The discussion given in the citation refers to the second (hybrid) version of the OBDD-A discussed in Chapter 4. However the arguments presented are equally applicable to the OBDD-A presented in this chapter. The performance of the OBDD-A is discussed throughout this chapter.

example Ψ(N,G). Once child nodes are generated the parent node is no longer required, making linking irrelevant. Thus, unlike existing OBDD methods [4, 15, 17, 18, 25, 26, 30], the nodes of Ψ(N,G) are not explicitly linked. Consequently, only part of the diagram will reside in memory at any one time.[6]

Each Ψ(N,G) is divided into a number of levels, with node N_0 on level 0 (the highest level), N_1 and N_2 on level 1, N_3 to N_6 on level 2, and so on. Thus any node N_i is on level k if and only if $2^k\text{-}1 \leq i \leq 2^{k+1}\text{-}2$. Each level k of Ψ(N,G) (except the last) represents an evaluation/decision of a variable and we say that a node on this level *decides* that variable, which is called the *decision variable* for level k.

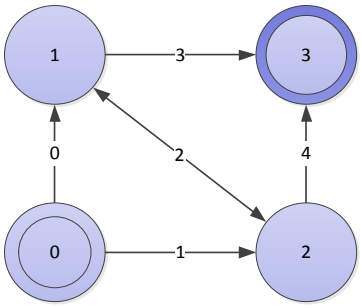


Figure 3.1: Simple Network

For Model 'e' addressed in this chapter, the decision variable is e_k, and hence the negative child of node N_i on level k represents the state of the network with e_k failed, and the positive child represents the state of the network with e_k available. Each level of the diagram in Figure 3.2 is identified by a value of k. For Model 'e', the depth of the diagram is |E|+1. For Models 'v' and 've', the depth is |V|+1 and |V|+|E|+1 respectively, as described in Appendix A. The last (lowest) level contains only the two terminal nodes[7], defined in Section 3.2.2. As an example, the Ψ(N,G) in Figure 3.2 has 5+1=6 levels since the network in Figure 3.1 contains five edges.

The OBDD-A stores information on the network state in each node on level k+1 based on the variable decisions for nodes on level k, which in turn are updated based on the variable decisions on nodes in level k-1, and so on. Therefore at any time OBDD-A algorithm needs to store nodes on any two consecutive levels; each *parent* node is removed from level k in turn and processed to give two *child* nodes

[6] Certain applications, such as network upgrade, may require the entire diagram to be kept in memory. This thesis only addresses the applications discussed, namely the computation of REL, EHC and EMD. Hence this work assumes that nodes do not need to be stored.

[7] Formally the last level of an OBDD-A contains the two terminal nodes, meaning there is one level more than the number of variables being decided. Practically, however, the terminal nodes do not need to be explicitly stored. Instead the information that would be contained in these nodes is harvested and stored in global variables, as discussed in Section 3.3.3. This means that the OBDD-A only generates nodes for a number of levels equal to the number of variables being decided (*e.g.*, |E| if vertices are perfect).

on level k+1. Each OBDD-A node, $N_i \in N$, contains an information pair [VI_i, CI_i]. This pair describes the *network state* that the node represents. The OBDD-A in Figure 3.2 uses a graph to illustrate the data structure that stores each node's network state information. Section 3.2.3 and 3.2.4 describe the details of vertex information VI_i and condition information CI_i respectively.

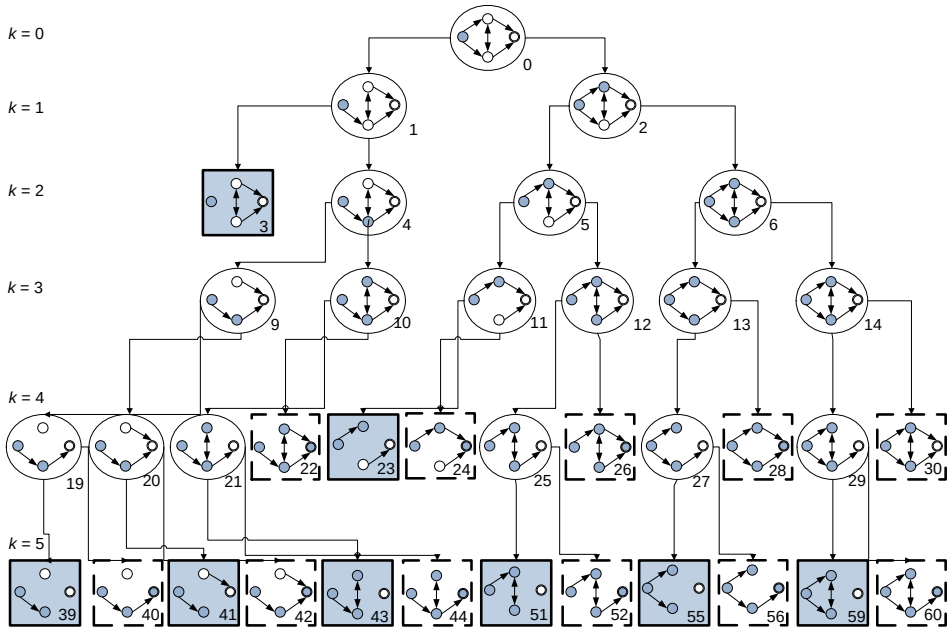


Figure 3.2: Unreduced OBDD-A for Sample Network in Figure 3.1

3.2.2. Node Type

OBDD-A nodes are divided into terminal, and non-terminal nodes. An OBDD-A node is *terminal* if it does not have children and is *non-terminal* otherwise. For network reliability, a node is terminal if the network state information it contains allows a complete determination of the network connectivity.

Each non-terminal node (shown in Figure 3.2 as a circle) has two child nodes. Terminal nodes are contained in squares and have no child nodes. A terminal node can be a *failure* node or a *success* node; in Figure 3.2, failure nodes are shaded and success nodes are un-shaded and have a dashed border. A node is successful if the source and target vertices are known to be connected and a failure node when they are known to be disconnected. In other words, when a node represents a state from which it is impossible to meet the requirements for the problem (*i.e.*, if it is no longer possible to send a message from the source to the sink vertex) it becomes a

failure node. When a node represents a state that meets the requirements then it is a success node.

For example consider N_3 in Figure 3.2, which is the failure node labelled 3 on level k=2. Because both edges leaving the source vertex have failed and been deleted, it is not possible to reach any other vertex, including the target. Hence N_3 is marked as a failure node. This detection of success and failure nodes reduces the diagram from a maximum of 63 (1+2+4+8+16+32) nodes to the displayed 37 nodes. A full diagram would show the sub-trees of the currently displayed terminal nodes whereas Figure 3.2 omits these. Each node in the sub-tree of a failure node is itself a failure node, and each node in the sub-tree of a success node is itself a success node. Terminal nodes are discussed further in Section 3.3.3.

3.2.3. Vertex Information

The *vertex information*, VI_i, is a pair (VS_i, P_i) that encodes network state information VS_i and the probability P_i of the network state represented by N_i, computed as discussed in Section 2.5.1. The (VS_i, P_i) pair stores all information required for the computation of REL, based on edges that have been decided for the diagram. The OBDD-A method computes REL from the (VS_i, P_i) pairs of all success nodes. When a node is found to be a success node, its probability is added to the network's reliability.

One way to encode VS_i is to record the decided edges as available or as failed. However, such an encoding is too inefficient for large networks, and contains much information that is not required. As an alternative, encoding only those reaching paths that have vertices in the *boundary set* F_k as end points is sufficient for computing REL. Specifically, the network state is stored in VS_i={ v_a, v_b … v_α }, where v_a, v_b … v_α are those elements of F_k (if N_i is on level k) that have *reaching paths* in the current network state. See Chapter 2 for the definitions of *boundary set* (BS) and *reaching paths*. This alternative requires far less information to be stored and manipulated.

The BS divides the edges in E of the graph into two groups for the k^{th} level of the diagram (level k-1); a group E_k that contains all edges e_0 to e_{k-1} that have already been decided and another group $\overline{E_k}$ containing the undecided edges e_{k+1} onwards. A vertex is in the BS F_k if it adjacent (incident) to at least one edge from each of the two groups. As each edge e_k is decided its endpoints are moved into F_k (if they are not already present), but when the last edge adjacent to a vertex has been decided

that vertex is removed from the BS. When a vertex is removed from the BS in this fashion we call it *redundant*. Section 3.3.2 describes a function to detect redundant vertex.

For example, consider the network in Figure 3.3 with s=v_0 and t=v_3. Assume that edges e_1 and e_3 have been decided as available (thick line) and edges e_0 and e_2 failed (dotted line) as shown in the figure; edge e_4 has not yet been decided. Since both v_2 and v_3 are adjacent to an edge that has been decided and also are adjacent to an edge that has not been decided, both vertices are in F_k, for the current level of the diagram. For this example, the only reaching path is to v_2, and hence the OBDD-A stores only VS_i = $\{v_2\}$ to fully describe the network state. All information about v_0 and v_1 is known, and hence these vertices are no longer important to the computation and have been removed from the BS; this occurred when the last edge adjacent to each of them was decided.

Note that when the OBDD-A was introduced in [104], the term *active vertices* was used for such vertices. Since active vertices are exactly those in the BS this thesis refers only to the BS to avoid confusion. The BS definition in Section 2.9.1 is sufficient for Model 'e' addressed in this chapter, although it is not complete. A complete definition of the BS is introduced in Section 4.2.2. While the description of the OBDD-A algorithm refers to F_k, the actual BS is never explicitly generated by the algorithm. In addition, while the BS algorithm by Hardy, Lucet and Limnios [15, 18] uses F_k, it tracks connections between vertices in the BS through the use of partitions, instead of the VI/CI notation used by the OBDD-A. The hybrid algorithm that combines both OBDD-A and BS is discussed in Chapter 4.

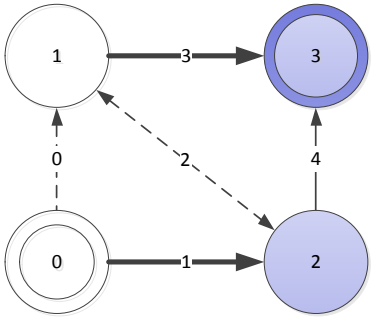


Figure 3.3: Sample Network Showing Network State

3.2.4. Condition Information

The *condition information* (CI_i) for a node N_i on level k is a set of *conditions* $\{C^0, C^1, \ldots C^{|CI|-1}\}$ of the form $C^x=(v_f,v_t)$, where v_f and v_t are both in F_k. We call v_f the

from (or *first*) endpoint of condition C^x and v_t the *to* (or *second*) endpoint of the condition.

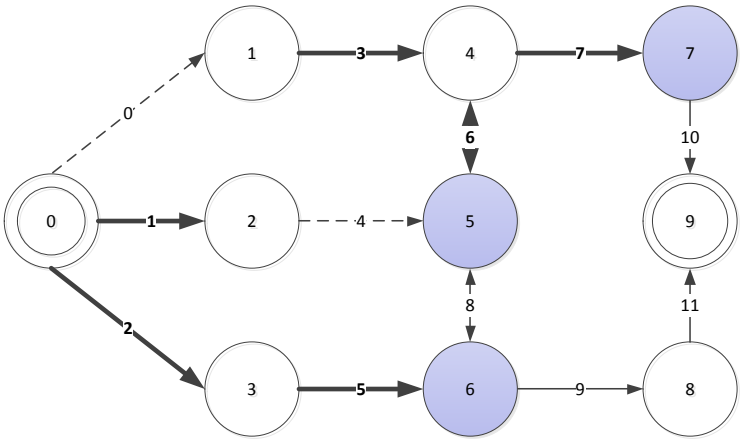


Figure 3.4: Sample Network to Illustrate Conditional Information

Consider the state of the network in Figure 3.4; the thick edges (e_1, e_2, e_3, e_5, e_6 and e_7) have been decided to be available and the dotted edges (e_0 and e_4) have been decided to be failed. In this case F_k = {v_5, v_6, v_7} as shown by the blue vertices. It can be seen that only one reaching path exists (to v_6) and hence VS_i = {v_6}. However it can be seen that a simple path (v_5,v_4,v_7) exists. This simple path is not a reaching path and hence is not stored in VS_i; such a path is called a *conditional path* and stored as a condition C^x=(v_5, v_7) in CI_i.

More formally, a *conditional path* is a simple path in the network that is not a reaching path. REL is computed based only on the minpaths of a network, and hence conditional paths do not contribute directly to REL. However if the beginning of a conditional path is later reached, a new minpath or reaching path may be formed. For example, if e_8 in Figure 3.4 is later decided to be available then the reaching path (v_0,v_3,v_6) is extended to (v_0,v_3,v_6,v_5). This reaching path is further extended via the conditional path (v_5,v_4,v_7) to become the new reaching path (v_0,v_3,v_6,v_5,v_4,v_7). Note that other conditional paths (*e.g.*, (v_4,v_5)) are not stored since at least one of the endpoints (in this case v_4) is not in F_k. Each such conditional path will not generate further paths since all edges adjacent to the redundant vertex have been decided.

When a node N_i is processed, the network state information it contains (*i.e.*, (VI_i, CI_i)) is updated to reflect that the edge being decided is either available (for the positive child) or unavailable (for the negative child). This update is achieved by first adding conditional information to CI_i representing an available edge, and then

combining all CI_i and VI_i to generate more conditions and/or additional elements of VI_i.

3.2.5. Information Redundancy

A vertex is added to F_k the first time an adjacent edge is decided, indicating that some information regarding this vertex may be known. The vertex remains in F_k until all adjacent edges have been decided. When this happens all information regarding this vertex is known, and it becomes redundant. On the diagram level at which a vertex is redundant, all information containing that vertex (presence in VS_i and conditions with this vertex as an endpoint) are removed from all nodes.

The OBDD-A algorithm generates child nodes that contain the redundant information, since removing information before node generation is complete can create difficulties. Only when the child nodes are complete is any redundant information removed. Reducing the amount of information in a node increases the occurrence of isomorphism, and hence improves the performance of the algorithm. Section 3.3.2 describes two methods to detect and remove a redundant vertex.

3.2.6. Node Isomorphism

In OBDD-A, variables are decided in a fixed order for all nodes from the root to the terminal node, and thus any two nodes on the same level decide the same variables. However, despite representing different combinations of inactive and failed variables, nodes on one level may represent identical network states.

For example, consider the four nodes shown in Figure 3.5; these nodes can all be found on level k=3 of Figure 3.2. For each of these nodes, both v_1 and v_2 have been reached, there are two edges still to be decided and v_3 is not yet in the boundary set. The difference between these nodes is that the edges between v_0, v_1 and v_2 are available for some and failed for others. However, each of the network states represented by the nodes has reaching paths to both v_1 and v_2. Since network reliability only considers the connectedness of vertices with the source, the current connectedness of the vertices in each node is identical. For this case, deciding the two currently undecided edges will affect each node equally, since the current vertices reached are identical; hence the sub-trees below each of these nodes will be identical. Consequently, computing more than one of these sub-trees is unnecessary and should be avoided. This can be done by testing for node *isomorphism*.

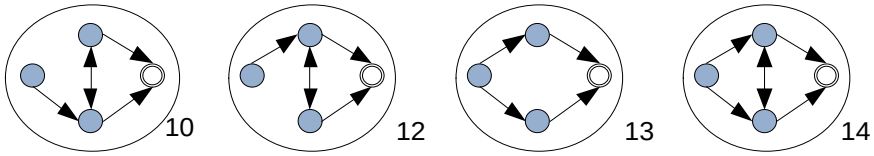


Figure 3.5: Isomorphic Nodes

Two nodes are *isomorphic* if their sub-diagrams are identical and hence all of the four nodes above are isomorphic. It is desirable to test for node isomorphism with a minimum of computational expense. One may detect isomorphic nodes by comparing their identical sub-diagrams, however this approach is not advisable since it requires generating all four sub-diagrams. Since the number of nodes on one level may be large, detecting isomorphism using such approach would be impractical.

For the OBDD-A computing REL, we use the following definition to test for node isomorphism.

Definition 3.1. Two nodes N_i and N_j at the same level of an OBDD-A are *isomorphic* if $VS_i = VS_j$ and $CI_i = CI_j$.

This definition is specific to the OBDD-A. For a general OBDD, two nodes are isomorphic if they have identical sub-trees [76]. Consider two nodes N_i and N_j on level *k* of an OBDD-A diagram. As shown in Section 3.3, the child nodes of N_i are fully determined by the (VI_i, CI_i) pair and the state of e_k ('available' for N_{2i+2} and 'failed' for N_{2i+1}) and the same holds for N_j.

If VI_i=VI_j and CI_i=CI_j, it follows that the only factor differentiating the (VI_x, CI_x) pairs of the child nodes N_x is the state of e_k. Since e_k is available for both positive child nodes, the (VI_x, CI_x) pairs of those nodes will be identical; similarly the (VI_x, CI_x) pairs of both negative child nodes will be identical. The respective children of these child nodes (*e.g.*, the positive child nodes of the positive child nodes) will similarly have identical (VI_x, CI_x) pairs, and hence the sub-trees of N_i and N_j are identical. Hence two OBDD-A nodes that are isomorphic according to definition 3.1 will also be isomorphic according to the more general OBDD definition of isomorphism.

Unlike standard OBDD isomorphism [76], the property above does not allow for the possibility of nodes at different levels being isomorphic. Since OBDD-A constantly removes nodes on one level while generating nodes in the next level, any form of isomorphism designed to remove nodes after child nodes are created

gives no benefit. In other words, the OBDD-A algorithm must detect whether the child nodes will be isomorphic to the parent before the child nodes are generated. Model ‘ve’ allows some use of isomorphism between levels (see Appendix A) but this thesis does not consider this possibility for Model ‘e’. Heuristics exist to check for certain types of such isomorphism, including for ‘e’, but the additional testing requirement and relative rareness of such cases makes using the heuristics impractical. This is discussed further in Chapter 6.

Two isomorphic nodes N_i and N_j are merged into a single node N_m that contains $VS_m=VS_i=VS_j$, $CI_m=CI_i=CI_j$ and $P_m=P_i+P_j$ since both nodes represent disjoint states. Without loss of generality, we set m=MIN(i, j).

3.3. The OBDD-A Algorithm

Figure 3.6 shows the OBDD-A algorithm for Model ‘e’; Appendix A describes the OBDD-A algorithm for Models ‘v’ and ‘ve’. Line 1 creates the root node N_0 and its information that represents the network state where information is ready to be sent from the sources, but no components have yet been decided as available or failed. The step sets N_0 = [VI_0= (S, 1.0), CI_0 ={ }]; CI_0 = { } because no decisions have been made at level 0, thus there are no conditional paths. It sets VS_0 = S, the set of source vertices, since information starts at source vertices. Notice that for Model 1e with only a single source v_0, VS_0={v_0}. The probability of the network state at N_0 is 1.0; the initial network state is always able to occur.

Line 2 initializes the level k=0, indicating that vertices have last been generated on level 0 of the diagram and, for Model ‘e’, that e_0 is the variable being decided. The reliability accumulator, RELIABILITY, is initialized to 0. The step also initializes the two queues used by the algorithm’s main body, Q_C and Q_N. Q_C contains the current nodes at level k, while Q_N stores nodes generated at level k+1. Since k=0, Step 2 sets Q_C={N_0} and Q_N={}. The final part of initialization involves setting the variables REDF and REDT, which are set to **TRUE** if the *from* and *to* vertices of the edge being decided are redundant, respectively. This is done by calling CHECK-REDUNDANT (line 3), which is described in Section 3.3.2. Note that most networks have both variables REDF and REDT set to **FALSE** which could be handled through a simple initialization; the call to CHECK-REDUNDANT in line 3 is left in for the rare network that does require it.

For each level k, the first node is removed from Q_C (line 4) and functions CREATE-NEG-CHILD and CREATE-POS-CHILD are called (lines 5,6), using

the node's information to generate a negative child and a positive child respectively; Section 3.3.1.2 describes the implementation details of the two functions. The positive child represents edge e_k being decided as active, while the negative child represents e_k being failed. If either endpoint of e_k is redundant this endpoint is removed from both VS_i and CI_i of both child nodes (lines 8-11) using the DEL-REDUNDANT function, described in Section 3.3.2.

Once both child nodes are generated, they are tested for termination (line 12); Section 3.3.3 describes the details of function NON-TERMINAL-NODE. Terminal nodes are discarded while non-terminal nodes are stored on level *k*+1 of the diagram. Storing a child node first involves comparing it to nodes on Q_N to check for isomorphism (line 13). The implementation in this thesis uses hashing to speed up the comparison of nodes, meaning that a newly created child node does not necessarily have to be compared to every node already on Q_N.

```
OBDD-A (G):                          // Algorithm for Edge Failure
1)   Initialize the root node N0=[( S, 1.0 ), { }];
2)   Initialize k=0, QC = { N0 }, QN = { }, reliability = 0;
3)   CHECK-REDUNDANT(k, redf, redt);
4)   Remove the first node, Ni, from QC;
5)   N2i+2 = CREATE-POS-CHILD(Ni, k );
6)   N2i+1 = CREATE-NEG-CHILD(Ni, k );
7)   for each newly created child node Nfi do
8)        if redf then
9)             DEL-REDUNDANT( Nfi, vf );
10)       if redt then
11)            DEL-REDUNDANT( Nfi, vt );
12)       if (NON-TERMINAL-NODE(Nfi, reliability)) then
13)            Check each node on QN for isomorphism with Nfi;
14)            if ( an isomorphic node Nx was found ) then
15)                 Px = Px + Pfi;          // Merge nodes
16)            else
17)                 Add Nfi onto QN;
18)  if ( QC == { } ) then
19)       if ( QN == { } ) then
20)            return reliability;
21)       else
22)            k++;       // ek = (vf, vt) or {vf, vt}
23)            Swap QC and QN;
24)            CHECK-REDUNDANT( k, redf, redt);
25)  goto 4)
```

Figure 3.6: OBDD-A Algorithm for Edge Failure

If an existing node, N_x, is found to be isomorphic to the child node, N_{fi}, then the child node is merged into the existing node (lines 14-15). For REL, this involves merging node probabilities ($P_x = P_x + P_{fi}$) as discussed in Section 3.2.5. If no isomorphic node is found the newly created child node is added to the end of Q_N (lines 16-17).

Once both child nodes have been discarded or stored, the loop conditions (lines 18-19) are checked. If both queues are empty, there are no more nodes to process and the algorithm terminates with variable RELIABILITY containing the value of REL (line 20). If only Q_C is empty, the algorithm has finished processing a level and must move onto creating the next level of the diagram (lines 21-24). This involves incrementing the level, *k*, moving the nodes on Q_N to Q_C and checking for redundancy. The move of queues can be implemented as a swap operation (line 23) since the contents of Q_C are to be moved to Q_N, it is known that Q_C is currently empty (due to the condition in line 18) and Q_N is to be set as empty in order to be ready to receive the nodes on the next diagram level.

3.3.1. Conditions and Child Nodes

3.3.1.1 Conditional Information

A condition is added to the positive child of a node, representing the directed edge decided as being available at the deciding level (and another condition for the reverse direction of an undirected edge). For edge $e_k=(v_f,v_t)$ the condition added represents the edge; (v_f,v_t). If the edge is undirected the reverse condition added is (v_f,v_t). Two conditions $C_x = (v_f^x,v_t^x)$ and $C_y = (v_f^y,v_t^y)$ are equal (written $C_x = C_y$) when every part of the conditions are equal; that is $v_f^x = v_f^y$ and $v_t^x = v_t^y$.

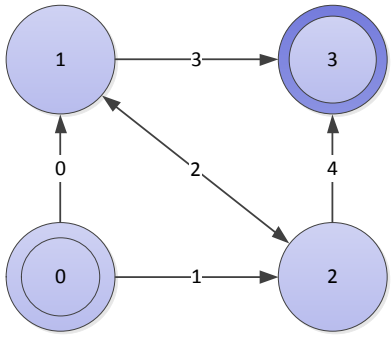


Figure 3.7: Simple Network

Consider the network shown in Figure 3.7 with $s = v_0$ and $t = v_3$; the first edge to be decided is $e_0=(v_0,v_1)$. Hence the root node N_0=[({v_0}, 1.0), { }] is modified by the addition of condition (v_0,v_1) to give the (incomplete) positive child N_2=[({v_0}, 1.0), { (v_0,v_1) }]. This represents the edge between v_0 and v_1 being decided as available and hence a new path being created from v_0 to v_1. This is done using ADD-COND function shown in Figure 3.8. This method is used in CREATE-NEG-CHILD and CREATE-POS-CHILD methods, described in the following section.

Recall that one of the requirements for the condition (v_0,v_1) to be in N_2 is that neither endpoint is in VS_2; hence the newly added condition should be deleted immediately after being transformed into a reaching path. Similarly, when a vertex is added to VS_i any conditions with this vertex as the second endpoint are deleted for the same reason – once an endpoint is in VS_i the condition is no longer required.

To prevent the unnecessary double modification of N_2, the OBDD-A algorithm does not actually add the condition. Instead, when adding the condition it first checks to see if the condition already exists in CI_i or if the second vertex already exists in VS_i; in either case the condition is discarded as shown in Figure 3.8 (lines 1-4).

If neither of the two conditions above hold, but the first endpoint is in VS_i, then the second endpoint, v_t, is added to VS_i since this vertex has now been reached along the newly decided edge using the TRIGGER method (lines 5-6). The TRIGGER method is shown in Figure 3.9. In this case the condition isn't actually added to avoid having to delete it again immediately. Hence N_2=[({v_0,v_1}, 0.9), { }] with CI_2 empty.

```
ADD-COND (N_i, (v_f, v_t)):
1)  if ( (v_f, v_t)∈CI_i ) then     // If exists do nothing
2)      return N_i;
3)  if ( v_t ∈VS_i ) then           // If reached do nothing
4)      return N_i;
5)  if ( v_f ∈VS_i ) then
6)      TRIGGER(N_i, v_t );
7)  else
8)      Add (v_f, v_t) to CI_i;
9)      for each C = ( v_t, v_x ) ∈ CI_i and x≠f do
10)         ADD-COND (N_i, ( v_f, v_x ) );
11)     for each C = ( v_x, v_f ) ∈ CI_i and x≠t do
12)         ADD-COND (N_i, ( v_x, v_t ) );
13) return N_i;
```

Figure 3.8: ADD-COND

If neither endpoint of the new condition is in VS_i and the condition doesn't already exist, then the new condition is added to CI_i (lines 7-12). Note that both endpoints will be in F_k or F_{k+1} since they are included in other conditions. This triggers recursive calls to ADD-COND as conditions adjacent to the new condition's endpoints are extended. If we add the condition (v_f, v_t) and the condition (v_t, v_x) already exists, then we also add the composite condition (v_f, v_x). This is important because, for example if we are finished with v_t before v_f we will delete all information regarding v_t, including the conditions (v_f, v_t) and (v_t, v_x). Since the

composite condition (v_f, v_x) does not have v_t as an endpoint, this would not be deleted however.

The TRIGGER method is used to add a new vertex, v_t, to VS_i. It is only called when v_t is not already in VS_i; conditions leading to v_t are immediately deleted and ADD-COND does not add further conditions to vertices in VS_i. Conditions leading from v_t lead to recursive calls to TRIGGER on the other endpoint; when we have reached v_t, any condition (v_t, v_x) indicates that v_x is also reachable. Note that the recursive call to TRIGGER will delete the condition that caused it to be called. The recursive call maintains the pre-condition that TRIGGER only be called on vertices in VS_i since the endpoint v_x is the endpoint of a condition and therefore is not in VS_i.

```
TRIGGER (N_i, v_t):
1)  Add v_t to VS_i;
2)  for each C1 = ( v_t, v_x ) ∈ CI_i do
3)      Add x to list L;      // Remember new vertices
4)      Delete C1 from CI_i;
5)      for each C2 = ( v_x, v_t ) ∈ CI_i do
6)          Delete C2 from CI_i;
7)  for each x in List L do
8)      TRIGGER (N_i, v_x );
9)  return N_i;
```

Figure 3.9: TRIGGER

3.3.1.2 Creating Child Nodes

The positive and negative child nodes are created as copies of the parent node and then updated using the CREATE-POS-CHILD and CREATE-NEG-CHILD methods, shown in Figure 3.10 and Figure 3.11 respectively. Each positive child is updated by adding the edge being decided as a condition. If the edge is undirected, a second condition is added for the reverse direction of the edge. The node probabilities of both the positive and negative children are updated to reflect the changed state probability.

```
CREATE-POS-CHILD (N_i, k):        // e_k = (v_f, v_t) or {v_f, v_t}
1)  Create N_2i+2 as a copy of N_i;
2)  P_2i+2 = P_i × Pr(e_k);
3)  ADD-COND(N_2i+2, (v_f, v_t) );
4)      if ( e_k is undirected ) then
5)          ADD-COND(N_2i+2, (v_t, v_f) );
6)  return N_2i+2;
```

Figure 3.10: CREATE-POS-CHILD

Consider the network shown in Figure 3.7 from Section 3.3.1.1 and apply CREATE-POS-CHILD to the root node N_0=[({v_0}, 1.0), { }] to create the

positive child. Assume that probability that each edge is active is 0.9, meaning that the state probability becomes 1.0×0.9 = 0.9. Hence, after line 2, N_2=[({v_0}, 0.9), { }]. The condition (v_0, v_1) is then added, as discussed above, meaning N_2 becomes [({v_0,v_1},0.9),{}]. Because e_1 is directed, the reverse (v_1,v_0) is not added.

Creating the negative child, shown in CREATE-NEG-CHILD in Figure 3.11, is simpler than creating the positive child since it represents the edge being down. Hence no new information is added, other than the state probability being updated to represent edge e_k having failed (line 2). Hence the negative child of N_0=[({v_0}, 1.0), { }] is N_1=[({v_0}, 0.1), {}].

Note that since the negative child is created last, there is no need to create a new copy of the parent node. Since the parent node will be discarded after creating the negative child it is more efficient to simply relabel it to become the negative child.

CREATE-NEG-CHILD (N_i, *k*):
1) Relabel N_i as N_{2i+1}; // N_i no longer needed
2) $P_{2i+1} = P_i \times (1 - Pr(e_k))$;
3) **return** N_{2i+1};

Figure 3.11: CREATE-NEG-CHILD

Once both child nodes have been created any unnecessary information needs to be removed from them. This occurs when a vertex changes from being active to being inactive. At this point, all information on the vertex is removed from the child nodes; this includes being removed from VS_i (and hence VI_i) and all conditions with that endpoint being removed from CI_i.

3.3.2. Detecting and Removing Redundant Vertices

The OBDD-A algorithm must check for vertex redundancy when starting a new level of the diagram. Since a vertex can only become redundant when the last edge adjacent to it has been decided, only the two endpoints of the edge being decided must be tested. The naïve way to check is to scan all undecided edges, looking for either endpoint, but this method is inefficient for large networks. The OBDD-A takes advantage of the variable ordering, described in Section 2.10.4, to improve testing efficiency, as shown in Figure 3.13.

```
CHECK-REDUNDANT ( k, redf, redt): // e_k=(v_f,v_t) and e_{k+1}=(v_a,v_b)
1)  low_k = MIN(f,t); high_k = MAX(f,t);
2)  redf = false; redt = false;
3)  if ( low_k != MIN(a,b)) then
4)      if ( f < t )
5)          redf = true;                        // v_f is redundant
6)      else                                    // Reverse edge
7)          redt = true;
8)  for x = k+1 to x < |E| do                   // e_x = (v_y, v_z)
9)      if ( (y == high_k) or (z == high_k) ) then
10)         break;
11)     if ( high_k < MIN(y,z) ) then
12)         if ( f < t )
13)             redt = true;
14)         else
15)             redf = true;
16)         break;
17) if ( x == |E| )             // Checked all edges and didn't find t
18)     redt = true;
19) return;
```

Figure 3.12: CHECK-REDUNDANT

Firstly (line 1) the lower and upper endpoints of the edge being decided, $e_k=(v_f,v_t)$, is determined. This test is not needed for undirected edges, but networks with directed edges may have edges where $f > t$ or $a>b$. The two variables that store the redundancy state of the endpoints, REDF for v_f and REDT for v_t, are then initialized to false (line 2).

Next we check whether the lower of the endpoints of e_k is redundant. Due to the variable ordering we only need to check the next edge, e_{k+1}; if the endpoint isn't present in that edge it will not be present in any later edges and one of REDF or REDT is true (lines 3-7).

Unfortunately such a shortcut does not exist for the higher of the endpoints of e_k, hence the for loop (line 8) checks all edges from e_{k+1} onwards. If the higher endpoint is found in a later edge, then it is not redundant (lines 9-10). The benefit of the ordering in this case is twofold; firstly the vertex being sought is likely to be in an edge relatively close to e_k – if it exists in any other edge – and that if we come across an edge whose smaller endpoint is greater than the endpoint being sought we know that it is not contained in any other edge and is thus redundant (lines 11-16). Finally, if we searched all remaining edges and didn't find the endpoint sought, it is redundant (lines 17-18).

Once one or both endpoints of an edge have been found to be redundant, they must be removed from all child nodes created on this level, as shown in Figure 3.13. The information is deleted from VS_i (and hence VI_i) of each node (lines 1-2). In

addition, any condition that has the redundant vertex as an endpoint is deleted from CI_i (lines 3-5).

```
DEL-REDUNDANT ( N_i, v_x ):
1)  if (v_x ∈ VS_i ) then
2)        Delete v_x from VS_i;
3)  foreach ( C ∈ CI_i ) do      // C = (v_a, v_b)
4)        if (x == a) or (x == b)   then
5)              Delete C from CI_i;
6)  return N_i;
```

Figure 3.13: DEL-REDUNDANT

Note that redundancy can also be implementation dependent for network models such as Model 2.1 that require multiple target vertices to be reached. For such models the redundancy test above can lead to the deletion of target vertices, possibly causing errors. One possibility is to record target vertices reached separately in a structure similar to VS_i and information is never removed from this structure. This requires additional information to be stored and a number of code modifications. Unfortunately it complicates the computation of performability metrics.

An alternative is to modify the pseudo-code shown in Figure 3.13 such that target vertices are never found to be redundant. This requires changes in CHECK-REDUNDANT as well as possibly requiring extra processing for the creation of child nodes. The second approach was chosen for the implementation in this thesis to allow for the performability computations described in Chapter 5, but since changes are implementation dependent they are not reflected in the pseudo-code given in this thesis.

3.3.3. Terminal Node Detection

As described in Section 3.2.2, a node is terminal only when the desired connectivity criteria have been met. The connectivity criteria is determined by both the network model (*e.g.*, Model 1 or Model 2) and component failure model (*e.g.*, ‘e’ or ‘v’) used; thus testing for terminal node is necessarily affected by both models being used. This section introduces the termination test for the Model ‘e’ for all communication models, as shown in Figure 3.14. The termination tests for Models ‘v’ and ‘ve’ are discussed in Appendix A.

```
NON-TERMINAL-NODE (N_i, reliability):
1)  if ( |T_j ∩ VS_i| ≥ c_j ∀j=0…m ) then
2)      reliability += P_i;
3)      return false;                          //a success node
4)  else if ( VS_i == { } ) then
5)      return false;                          // a failure node
6)  else
7)      return true;                           //a non-terminal node
```

Figure 3.14: NON-TERMINAL-NODE

The algorithm first searches for success nodes (line 1) by traversing each target grouping T_i to check if sufficient target vertices have been reached. Note that target vertices are only explicitly grouped for Model 3.3; for all other models, the target vertices are considered to be in a single target grouping, and hence only one set of target vertices is compared to VS_i. The number of target vertices to be reached is $|T|$ for Model 2, c_j for Model 3.3, c for Model 3.2, and 1 for Models 1 and 3.1. As discussed in Section 2.3.2, Model 4 is the reverse of Model 3, and hence all references to Model 3 above also apply to the reverse of Model 4.

If the success conditions of the network model are satisfied (*i.e.*, line 1 is **TRUE**) then N_i is a success node; the probability is added to the current contents of the variable RELIABILITY, and a value of **FALSE** is returned to indicate a terminal node was found. For example if RELIABILITY is currently 0.0009 and P_i for a success node is 0.8748 then RELIABILITY becomes 0.0009+0.8748=0.8757. When execution completes, RELIABILITY contains the network's REL.

If N_i is not a success node, line 4 tests if it is a failure node. A failure node has no sub-trees that contain a success node, and thus it should be detected before being processed to avoid generating nodes unnecessarily. We detect node N_i as failed if VS_i is empty (line 4); if N_i has no information on any vertices, then no new vertices (including the target) will be reached. When a failure node is detected a value of **FALSE** is returned.

Note that this is a sufficient but not a necessary condition; N_i can be a failure node through not having any success nodes in sub-trees but not have empty VS_i. The VS_i test is an efficient one that has no false positives and detects failure nodes in most cases. A test that detects all failure nodes would be too computationally intensive to be practical since it would require generating the sub-trees in some way. The VS_i test is a good trade-off between efficiency and detecting all failure nodes.

If N_i is a terminal node, whether success or failure, it will be discarded instead of stored. The only difference between the two types of terminal nodes is that the

success nodes have their probability stored before being discarded. If a node is found to be non-terminal the method returns **TRUE**. When this occurs the OBDD-A algorithm must process the non-terminal node to create child nodes for the next level of the diagram.

3.4. Example

This section illustrates how the OBDD-A algorithm in Figure 3.6 computes the REL of the simple network in Figure 3.15 (a repeat of Figure 3.7) for Model 1e, with $s = v_0$, $t = v_3$, $Pr(e_i)=0.9$ for all edges e_i. The example focuses on the important aspects of each part of the algorithm.

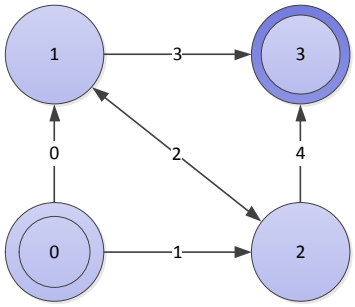


Figure 3.15: Simple Network

The resulting OBDD-A is shown in Figure 3.16. It can be seen that it has five levels that include non-terminal nodes, which is expected since the simple network has five fallible variables (*i.e.*, the edges of the graph). It has a sixth level which has only terminal nodes. There are at most three nodes on any level and 10 non-terminal nodes in total. The number of nodes actually created (equal to the number of arrows in the diagram) is 20; six of these are terminal nodes and four were found to be isomorphic and merged. Note that the unreduced diagram shown in Figure 3.2 has 37 nodes for the same problem out of a possible 63; pruning sub-trees with isomorphism eliminates 27 out of these 37 nodes.

Line 1 sets N_0=[({v_0}, 1.0), { }], as given in Section 3.3 with S = {v_0}; there is only one source vertex in this example. The next line sets Q_C = { N_0 }, Q_N = { } and k = 0; this means that we are at the top level (k=0) of the diagram which has only node N_0 and there are no nodes created on the next level (k=1) yet. Because k=0 we are deciding edge e_0 = {v_0,v_1}.[8] Line 3 uses CHECK-REDUNDANT to see whether any of the endpoints of e_0 will become redundant for the next level (k=1). This method compares e_0 and e_1, setting f=v_0, t=v_1, a=v_0 and b=v_2. Because $f = a$

[8] Note that because of the ordering used, the first edge to be decided will always be (v_0,v_1) or {v_0,v_1}.

we know that v_0 is not redundant (lines 7-8) and hence REDF remains set to **FALSE**. We next check for other occurrences of v_1 in edges of the network (lines 9-15) and find that $e_2 = \{v_1,v_2\}$ matches (line 10) and hence we exit the loop, leaving REDT also set to **FALSE**. CHECK-REDUNDANT completes and returns, having set the values of the global variables REDF and REDT; both set to **FALSE** to indicate that the 'from' and 'to' vertices of the edge being decided are not becoming redundant.

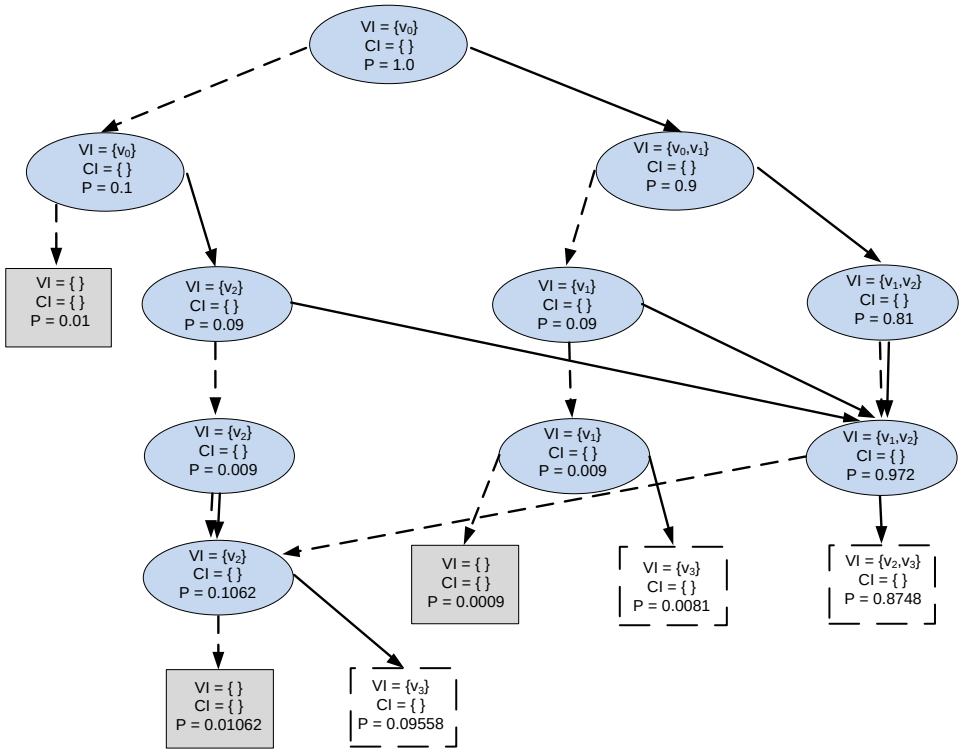


Figure 3.16: OBDD-Ae for Simple Network

Having completed the initialization for level *k*=0, the algorithm enters the main loop (lines 4 to 17). After removing the first node from Q_C in line 4, two child nodes, first the positive and then the negative, are created in lines 5 and 6 respectively.

The positive child, N_2, is created as a copy of N_0 and the probability is multiplied by $Pr(e_0)$=0.9, giving N_2=[({v_0}, 0.9), { }]. Next we add (v_0,v_1) via ADD-COND(N_2, (v_0,v_1)) and its reverse via ADD-COND(N_2, (v_1,v_0)).

For ADD-COND(N_2, (v_0,v_1)), the third **IF** statement (line 5) is true; v_0 is in VS_2 so v_1 is added to VS_2 giving N_2=[({v_0,v_1}, 0.9), { }]. Then all conditions with v_1 as an endpoint are removed from CI_2; there are currently no conditions so nothing is done. When ADD-COND(N_2, (v_1,v_0)) is called the second **IF** statement (line 3) is true and the method returns without making any modifications. The negative child,

N_1, is created by CREATE-NEG-CHILD by renaming N_0 to N_1 and multiplying the probability by 0.1 (= 1 - 0.9). Hence N_1 = [({v_0}, 0.1), { }].

Since there are no redundant vertices DEL-REDUNDANT is not called. The child nodes are tested for termination, but neither contains v_3 in VS and neither have empty VS; hence both child nodes are non-terminal and are added to Q_N. N_1 is added first to give Q_N = { N_1 } and then N_2 is compared to N_1 to check for isomorphism. Since $VS_1 \neq VS_2$ the nodes aren't isomorphic and Q_N = { N_1, N_2 }. These two nodes are now the only nodes in memory since N_0 was over-written to create N_1.

The main loop is now complete and the loop conditions (lines 18-24) are checked. Q_C is empty, but since Q_N is non-empty the loop continues on a new level. The queues are swapped and *k* is incremented to 1; the diagram is now deciding e_1={v_0,v_2} to build level *k*=2. CHECK-REDUNDANT is called with argument *k*=1.

The method compares edges e_1=(v_0,v_2) and e_2={v_1,v_2}. In this case the **IF** condition in line 7 is true since the 'from' vertex of e_1 does not appear in e_2; hence v_0 is redundant and REDF is set to **TRUE**. Since v_2 appears again in e_2 it is not redundant, and REDT is set to **FALSE**. The conditions and re-initialization over the program returns to the start of the main loop (line 4).

The first node, N_1 = { ({v_0}, 0.1), { } }, is removed from Q_C and processed to give child vertices N_3 and N_4. In a similar manner to above, N_3 is found to be [({v_0}, 0.01), { }] and N_4 = [({v_0,v_2}, 0.09), { }]. Since v_0 is redundant DEL-REDUNDANT is called on both vertices, giving N_3 = [({ }, 0.01), { }] and N_4 = [({v_2}, 0.09), { }]. NON-TERMINAL-NODE finds that N_3 is a failure node since VS_3 is empty, meaning that it is discarded. Node N_4 is non-terminal and is added to Q_N.

When the loop conditions are checked again no changes are called for, so execution returns to the start of the loop. The second node, N_2=[({v_0,v_1}, 0.9), { }], is similarly processed to give N_5=[({v_1}, 0.09), { }] and N_6=[({v_1,v_2}, 0.81), { }]. Both are non-terminal and are added to Q_N, completing level two of the diagram; Q_N = { N_4, N_5, N_6 }.

Q_C is empty, so *k* is incremented to 2 (deciding e_2 = {v_1,v_2}) and the queues are swapped. No vertex becomes redundant since both v_1 and v_2 are endpoints of undecided edges. N_4 = [({v_2}, 0.09), { }] is removed from Q_C and processed to

give N_9 = [({v_2}, 0.009), { }] and N_{10} = [({v_1,v_2}, 0.081), { }]. Both nodes are non-terminal and are added to Q_N. There are currently two nodes in Q_C and two nodes in Q_N, giving a total of four nodes in memory; this is the greatest number of nodes in memory at any one time during this computation.

Similarly N_5=[({v_1}, 0.09), { }] is processed to give N_{11}=[({v_1}, 0.009), { }] and N_{12}=[({v_1,v_2}, 0.081), { }]. Again both nodes are non-terminal, and N_{11} is added to Q_N as usual. However when N_{12} is checked against the nodes of Q_N it is found to be isomorphic with N_{10} since both VS and CI are identical for both nodes. Instead of being added to Q_N, N_{12} is merged with N_{10}, giving N_{10} = [({v_1,v_2}, 0.162), { }]. The VS and CI remain the same but the node probability becomes the sum of the probabilities of both nodes. Now Q_N = [N_9, N_{10}, N_{11}].

Finally N_6=[({v_1,v_2}, 0.81), { }] is removed from Q_C and processed to give N_{13}=[({v_1,v_2}, 0.081), { }] and N_{14}=[({v_1,v_2}, 0.729), { }]. In this case both child nodes are non-terminal and isomorphic with N_{10}, hence both are merged into it to give N_{10} = [({v_1,v_2}, 0.972), { }]. In Figure 3.16 the merging of these four nodes can be seen as four arrows entering the right-hand node on level two.

Since Q_C is empty once more, *k* is incremented to 3 (deciding e_3={v_1,v_3}) and the queues are swapped. Vertex v_1 is not an endpoint of e_4, the only edge that has not yet been decided, meaning it becomes redundant. The main loop begins with Q_C = [N_9, N_{10}, N_{11}].

Node N_9 = [({v_2}, 0.009), { }] is removed from Q_C and processed to give N_{19} = [({v_2}, 0.0009), { }] and N_{20} = [({v_2}, 0.0081), { }]. Both nodes are non-terminal, and are isomorphic with each other. Hence N_{19} is added to Q_N and N_{20} is merged with it to give N_{19} = [({v_2}, 0.009), { }].

The next node, N_{10} = [({v_1,v_2}, 0.972), { }], is removed from Q_C and processed to give N_{21} = [({v_2}, 0.0972), { }] and N_{22} = [({v_2,v_3}, 0.8748), { }]. The negative child, N_{21}, is non-terminal and isomorphic with N_{19}; hence N_{19} = [({v_2}, 0.1062), { }]. The positive child, N_{22}, has $v_3 \in VS_{22}$ and hence the intersection between T_0={v_3} and VS_{22} has size 1. Since K_0=1, N_{22} is a success node. Thus RELIABILITY is increased to 0.8748 and N_{22} is discarded.

Finally N_{11}=[({v_1}, 0.009), { }] is removed from Q_C and processed to give N_{23}=[({ }, 0.0009), { }] and N_{24}=[({v_3}, 0.0081), { }]. Both of these are terminal nodes; N_{23} is a failure node since VS_{23} is empty, and N_{24} is a success node since

$v_3 \in VS_{23}$. RELIABILITY is incremented by 0.0081 to 0.8829 and both nodes are discarded.

The last level of the diagram is begun by incrementing k to 4 (deciding $e_4=\{v_2,v_3\}$) and swapping the queues. Vertex v_2 becomes redundant; we are on the last level so all vertices other than v_t are redundant. The target vertex never becomes redundant since information regarding this is always relevant to our computation.

Q_C has only one node, N_{19} = [($\{v_2\}$, 0.1062), { }], which is removed and processed to give N_{39} = {[({ }, 0.01062), { }] and N_{40} = [($\{v_3\}$, 0.09558), { }]. Both are terminal nodes, as always; since this is the last level of the diagram neither node can have children. The negative child, N_{39}, is a failure node since VS_{39} is empty. However, node N_{40} is a success node since $v_3 \in VS_{40}$. The probability of the latter, 0.09558, is added to RELIABILITY to give 0.97848 and both nodes are discarded.

Neither queue contains any nodes, so the computation terminates (lines 18-20). The network reliability is equal to RELIABILITY = 0.97848. Of the 20 nodes created, only 10 were stored in memory and at most four at one time.

While the reliability can be computed from this diagram, other metrics such as the hop count cannot. Consider node N_{10} which represents the network state of a path from v_0 directly to v_1 as well as a different state where the path to v_1 travels through v_2; hop counts of 1 and 2 respectively. Both network states are subsumed in the same node, because path length information is not recorded. Appendix A demonstrates the working of the OBDD-A algorithm on Model ‘v’ and ‘ve’ and the computation of metrics other than reliability is discussed in Chapter 5.

3.5. The Complexity of the OBDD-A Algorithm

This section analyses the time and space complexity of OBDD-A for computing the REL of a graph G=(V,E). As the size of the network increases, so will both |V| and |E|. Let n=|V|, then the number of edges, |E|, is at most $\binom{n!}{(n-2)!} = n \times (n-1)$ edges; hence as n becomes large there are $O(n^2)$ edges. An OBDD-A diagram contains no more than $2^{|E|}$ non-terminal nodes since each node on the |E| non-terminal levels of the diagram has exactly two child nodes.

Practically speaking, the use of isomorphism reduces the maximum number of nodes considerably, however the number of nodes created (and hence both the space and time complexity) are still expected to be exponential in size. In this

section, we describe the details of OBDD-A complexity analysis when isomorphism is considered. In particular, Section 3.5.1 analyses OBDD-A space complexity. Then, sections 3.5.2 and 3.5.3 describe the time complexity of each function used in OBDD-A and its overall time complexity, respectively.

3.5.1. Analysis of Algorithm Space Complexity

Hardy, Lucet and Limnios [15] analysed the impact of F_{max}, the maximum size of the boundary set F_k, on the complexity of their BS algorithm. They found that the maximum number of diagram nodes on one level of the diagram to be $O(F_{max}^2 \times B_{F_{max}})$ where $B_{F_{max}}$ is the Bell number for F_{max} [15]. Note that $B_{F_{max}}$ is factorial in n [105]. The OBDD-A is shown in Section 3.6.2.1 to have the same number of nodes as the equivalent OBDD generated by BS, and hence the maximum number of nodes in one level of OBDD-A is also bounded by $O(F_{max}^2 \times B_{F_{max}})$. BS must keep all of its nodes and thus require at least $O(n^2 \times F_{max}^2 \times B_{F_{max}})$ storage since the number of levels has $O(n^2)$. In contrast, OBDD-A stores at most two diagram levels of nodes at any time; thus the space complexity of OBDD-A is $O(2 \times F_{max}^2 \times B_{F_{max}}) = O(F_{max}^2 \times B_{F_{max}})$.

The actual amount of storage used by the OBDD-A algorithm is dependent on the exact implementation. In order to obtain a better estimate of memory usage, the structure of the nodes is considered. Recall that each OBDD-A node, N_i, contains VS_i and CI_i structures. The maximum number of vertices in VS_i is at most the number of vertices in F_k, and hence the maximum size of VS_i at any time is equal to F_{max}. Since probability P_i stored in each VS_i is represented by a number, each VS_i requires $O(F_{max})$ space. For CI_i, since each condition must have endpoints that are in F_k, the maximum number of conditions in it is bounded as for $|E|$ above, giving $O(F_{max}^2)$ space complexity. Hence each node takes $O(F_{max}^2)$ space and the algorithm has a space complexity of $O(F_{max}^2 \times F_{max}^2 \times B_{F_{max}}) = O(F_{max}^4 \times B_{F_{max}})$. Note that any vertex that appears in VS_i cannot be the endpoint of a condition, which reduces the actual number of conditions in a node.

In practice, the number of conditions in a node is generally relatively small. Further, for families of nodes with constant F_{max}, the number of nodes becomes $O(n^2)$ and the space complexity becomes $O(1)$. These results are borne out by the results in Section 3.6.

3.5.2. Analysis of Function Time Complexity

Ignoring the recursive call, the complexity of the TRIGGER method is $O(F_{max}^2)$ for traversing $O(F_{max}^2)$ conditions in CI_i. It is assumed that the deletion of an element from CI_i can be done in constant time through the use of a data structure such as a linked list. The TRIGGER method is only called on vertices that are not yet in VS_i, so the number of calls to this function for any one node is bounded by F_{max}, giving a total complexity of $O(F_{max} \times F_{max}^2) = O(F_{max}^3)$ for all calls to TRIGGER for any particular node.

The second recursive function in the algorithm is ADD-COND, which calls both itself and TRIGGER. The initial part of the function (lines 1-6 in Figure 3.8) is not recursive, although it does contain the call to TRIGGER. The complexity of lines 1-4 is $O(F_{max}^2 + F_{max}) = O(F_{max}^2)$ since both CI_i and VS_i are tested. The call to TRIGGER is $O(F_{max}^3)$, but note that this includes all calls to TRIGGER.

The recursive part of the function (lines 8-12) includes a traversal of CI_i and two recursive calls. Consider the recursive calls divided into those that successfully add a new condition, those that add a new vertex, and those that do neither. A node can only have $O(F_{max}^2)$ conditions, hence this is the limit to the amount of calls to ADD-COND that can add a new condition. Since only calls to ADD-COND that add a new condition make recursive calls, it can be concluded that there are at most $O(3 \times F_{max}^2) = O(F_{max}^2)$ recursive calls made (one for each successful adding of a condition, and two unsuccessful calls for each of these).

Each call to ADD-COND tests both CI_i and VS_i (lines 1-4), with a cost of $O(F_{max}^2 + F_{max}) = O(F_{max}^2)$, giving a cost of $O(F_{max}^2 \times F_{max}^2) = O(F_{max}^4)$ for all calls to these lines. The total cost of TRIGGER is $O(F_{max}^3)$, which is not modified by multiple calls to ADD-COND since it already counts the maximum number of times that TRIGGER can be called. Hence the complexity of ADD-COND is $O(F_{max}^4 + F_{max}^2) = O(F_{max}^4)$.

The CREATE-POS-CHILD method first creates a copy of the parent node, which includes copying all conditions and thus has complexity $O(F_{max}^2)$. It then updates the probability (a constant operation since the probability variable is finite) and makes either one or two calls to ADD-COND, with complexity $O(F_{max}^4)$. Hence the complexity of CREATE-POS-CHILD is $O(F_{max}^2 + 1 + 2 \times F_{max}^4) = O(F_{max}^4)$. The CREATE-NEG-CHILD method does not create or copy a node, but

only changes a label (constant time) before modifying the probability (also constant time). Hence this method has O(1) complexity.

The CHECK-REDUNDANT function is mostly constant in complexity, apart from the traversal of E to check for the second endpoint. While, in practice, only a small part of E is likely to be traversed, this gives us a worst-case complexity of $O(n^2)$. DEL-REDUNDANT traverses both VS_i and CI_i, giving an identical complexity $O(F_{max}^2)$ on the assumption that deletions are constant-time operations. NON-TERMINAL-NODE traverses only VS_i, giving a complexity of $O(F_{max})$.

3.5.3. Analysis of Algorithm Time Complexity

The algorithm itself has an initialization (lines 1-3 of Figure 3.6) that is constant apart from the call to CHECK-REDUNDANT ($O(F_{max}^2)$). This is followed by a loop (lines 4-17) that is repeated for every level that includes non-terminal nodes, and every non-terminal node on those levels. Recall that the total number of nodes in the diagram is bounded by $O(n^2 \times F_{max}^2 \times B_{F_{max}})$.

For each non-terminal node, CREATE-POS-CHILD and CREATE-NEG-CHILD are each called once, DEL-REDUNDANT may be called up to two times. In addition, NON-TERMINAL-NODE is called twice, and an isomorphism check is performed with each node on the same level for each child. The isomorphism check has complexity $O(F_{max}^2)$ since both VS_i and CI_i may be traversed, and there are potentially up to $O(F_{max}^2 \times B_{F_{max}})$ comparisons made. The use of a good hashing function will reduce this in practice, but a perfect hash may require an extremely large amount of space. Ignoring the effects of the hash in order to compute the worst-case, results in a complexity of $O(F_{max}^2 \times F_{max}^2 \times B_{F_{max}}) = O(F_{max}^4 \times B_{F_{max}})$. Using a hash to check for isomorphism means that if no existing node is determined to be isomorphic, a place has been found to store the child node. Hence the result of finding an isomorphic node or not (lines 15 and 17 of Figure 3.6) requires constant time.

Thus processing a node to give two child nodes has a complexity of $O(F_{max}^4 + 1 + 2 \times (F_{max}^2 + F_{max}^2 + F_{max}^4 \times B_{F_{max}}))$. It can be seen that this complexity is dominated by the isomorphism check, and hence the complexity for processing a node to create two child nodes is $O(F_{max}^4 \times B_{F_{max}})$. Thus processing all of the nodes on one level of the OBDD-A has complexity $O(F_{max}^4 \times B_{F_{max}} \times F_{max}^2 \times B_{F_{max}}) = O(F_{max}^6 \times B_{F_{max}}^2)$.

Once the level has been completed, the loop checks for termination and prepares the next level, including a call to CHECK-REDUNDANT. The complexity of these steps (lines 18-24 of Figure 3.6) is dominated by the node generation for the level. Hence generating each level of the OBDD-A has complexity $O(F_{max}^6 \times B_{F_{max}}^2)$, and generating all $O(n^2)$ levels has complexity $O(F_{max}^6 \times B_{F_{max}}^2 \times n^2)$. This is combined with the initialization, $O(F_{max}^2)$, which is not significant. Thus the worst-case time complexity of the OBDD-A algorithm for computing the REL of Model 1e is $O(F_{max}^6 \times B_{F_{max}}^2 \times n^2)$.

Note that for families of networks that have constant inter-connectivity but an increasing number of nodes (*e.g.*, the grid $4 \times L$ networks) the maximum boundary set size, F_{max}, is constant. This indicates that $B_{F_{max}}$ is constant as well. For such a family of networks the time complexity of the OBDD-A is $O(n^2)$, which is based on the number of edges of the diagram. In the special case of a grid or w-connected network, the number of edges is linear in the number of vertices; hence families of such networks actually have time complexity $O(n)$. This is demonstrated in the results obtained in Section 3.6.

3.6. Performance Evaluation

3.6.1. Overview

Implementations of computing algorithms are traditionally measured in terms of execution time and the space required. For OBDD-based algorithms, space is traditionally measured in terms of diagram nodes [4, 6, 15, 17, 26, 30, 31]. While OBDD methods store all diagram nodes in memory, OBDD-A only stores two levels of the nodes at any one time. For this reason it is useful to measure both the total number of nodes generated and the maximum number of nodes per level of the diagram.

OBDD methods perform better on narrow networks, which are those with low F_{max}. The number of vertices and edges in the network under consideration are also relevant; one or both of these determine the depth of the diagram, depending on the component failure model used. This was demonstrated by the complexity analysis in Section 3.5. The worst-case for OBDD methods are networks with F_{max} as high as possible. For Models ‘e’ and ‘ve’ the worst case are the fully connected K*w* networks with *w* vertices and $F_{max}=w$, however these are trivial for Model ‘v’.

This section tests the implementation on a number of the networks discussed in Section 2.10. Firstly the performance of the OBDD-A is compared to existing methods in Section 3.6.2. Section 3.6.3 analyses the performance of the OBDD-A for Model 1e in order to demonstrate the effects of F_{max} on algorithm performance. In particular, the results from the complexity analysis in Section 3.5 are verified through testing. Section 3.6.4 tests the performance of the OBDD-A on other communication models (see Table 2.1). Section 3.6.5 gives an example of an application of the OBDD-A and discusses its performance, while Section 3.6.6 summarises the results.

3.6.2. Performance Comparison with Existing Methods

A number of methods have been proposed to solve network reliability. These include the boundary set (BS) method [15, 18], the EED_BFS method [17, 26], EF [13]. The BS method has the best time performance of these algorithms, but is only applicable to Model 'e' for undirected networks. While EED_BFS is not as fast, it can be extended to Model 've'. It can theoretically be used to compute solutions for Model 'v' by setting the probabilities of edges to 1[9] and applying Model 've', but results for doing this are not presented in [17, 26] and this approach is not mentioned. Hence Model 'v' does not list EDD_BFS as a relevant solution. Finally EF is not directly compared to either BS or EED_BFS; it applies only to 'v' networks and assumes a structure involving clusters.

Note that while EED_BFS is able to compute Model 2.1e, it does so by solving |T| versions of Model 1 and then combining them. By contrast, BS is able to solve Model 2.1e very efficiently, and solves Model 2.2e even more efficiently. While EF is able to solve Model 2.1v (and hence Model 2.2v) no results for doing so are given [13].

Table 3.1: Network Models Applicable to Various Existing Methods

	Component Failure Model		
Communication Model	**'e'**	**'v'**	**'ve'**
Model 1 – REL(s,t)	BS*, EED_BFS	EF*	EED_BFS
Model 2.1 – REL(s,T)	BS*, EED_BFS	EF*	EED_BFS
Model 2.2 – REL(s,T),T=V	BS*, EED_BFS	EF*	EED_BFS
Model 3.1 – REL(*s*,1-of-T)	None	None	None
Model 3.2 – REL(*s*, *c*-of-T)	None	None	None
Model 3.3 – REL(*s*, c_i-of-T_i)	None	None	None

[9] Performance for EED_BFS on 'v' would be identical to the performance for Model 've', which places it at a great disadvantage compared to methods designed specifically for Model 'v'.

The component failure and communication model that each approach applies to is shown in Table 3.1, with an asterisk indicating special network requirements. The OBDD-A applies to each combination listed, and hence is not shown in the table. Further, OBDD-A is not restricted to networks that meet special requirements.

The state enumeration method is not mentioned in this table, since although it can solve each method it is not feasible for networks of even moderate size. In addition, methods which solve only one particular sub-type of network (*e.g.*, directed ladder networks [7]) are not included in the table. Finally, methods which have been shown to be less efficient than one or more listed methods without being applicable to more models (*e.g.*, CAREL [10]) are not shown.

OBDD-A has been designed to be a flexible algorithm that is applicable to as many network models as possible. In addition, the implementation used in this thesis focuses on code sharing for better comparison between different models, instead of applying heuristics and programming tricks that may optimize the performance for any particular model. By contrast, it can be seen that the other successful methods are more limited. In particular, the method whose time performance is the best (BS) is extremely limited, while the method that is the most flexible (EED_BFS) has relatively poor performance on large networks.

The BS algorithm [15, 18] has an effective way to extend its normal test for node failure to this special case, resulting in BS being more efficient for Model 2 than for Model 1. The test comes about as a result of the partition notation used by BS; when a partition becomes empty as a result of a redundant vertex being deleted, the node is a failure node. Unfortunately OBDD-A notation does not allow a simple test; when a vertex is removed from a node without having been reached it may still reside in a condition. Complex tests can be performed, but these add a processing overhead to each node. Worse, because OBDD-A must track targets reached even when those vertices have become redundant, computing Model 2 greatly reduces node isomorphism.

A number of factors impact performance comparisons between different methods. Results shown in this section for the BS, EED_BFS and EF algorithms are taken from the papers presenting each of these algorithms. The implementations for these papers were executed on different computers from the one used to generate the OBDD-A results for this thesis, the network input may be different and the implementation style is unknown. These factors were discussed in Section 2.11, and combine to make processing time comparisons inaccurate. Comparisons in this

section are intended as a general guideline rather than a definitive ranking of performance.

The following sub-sections compare the performance of the general OBDD-A implementation with the algorithms applicable to Models ‘e’, ‘v’ and ‘ve’ respectively. Comparisons are only made for Model 1 and 2.2 since results given for EED_BFS [17, 26], EF [13] address only Model 1 and 2. For Model 2.1 the exact vertices chosen as the target vertices needing to be connected are not specified. This means that a comparison between the OBDD-A and these methods gives no real information for Model 2.1.

3.6.2.1 Model ‘e’

As discussed in Section 2.7 the computation of reliability for Model 1e and related metrics has been the focus of much work. Excluding the OBDD-A, the state of the art solutions are the BS method [15, 18] and the EED_BFS method [17]. The former exhibits better performance but is restricted to undirected networks while the latter can be used for any network. Since the OBDD-A is able to compute REL for both directed and undirected networks its performance is compared to both of these methods. However comparisons are only made using undirected networks, in order to allow BS to be used.

Table 3.2: Comparison between BS, EED_BFS and OBDD-A for Model 1e

	BS		EED_BFS		OBDD-A		
Network	**Time**	**Nodes**	**Time**	**Nodes**	**Time**	**Nodes**	**Max N**
K6	0.03	154	0.02	148	0.04	190	27
path 18	0.06	1810	0.02	292	0.03	284	27
path 19	0.09	5370	0.33	3591	0.06	4912	422
ring 2x6	0.35	65665	0.08	423	0.03	264	28
ring 2x8	0.11	8452	0.02	302	0.03	432	28
3x10	0.06	356	0.55	257	0.03	384	9
3x12	0.05	440	2.88	317	0.03	474	9
5x5	0.05	2425	0.43	1149	0.04	1694	90
2x20	0.08	152	0.02	115	0.03	169	3
2x100	0.02	792	2.52	595	0.03	889	3

Table 3.2 compares EED_BFS, BS and OBDD-A for Model 1e. The EED_BFS performance is taken from [17] and BS from [15]. Note that the number of nodes generated for each method is different, with OBDD-A often generating more nodes than EED_BFS but less than BS. Although the same networks are used and all are sorted using BFS, the actual order of the variables can still vary. OBDD-A shows

relatively good time performance on these relatively small[10] networks; note however that part of the improvement over EED_BFS is due to the testing of the latter having been carried out on a slower computer. Furthermore, all nodes for both BS and EED_BFS must be stored while OBDD-A only stores at most twice the number of nodes shown in the (Max N) column, which shows the maximum number of nodes on any one level of the OBDD-A. For example, for the 2×6 ring network, EED_BFS stores 423 nodes and BS stores 65,665 nodes while OBDD-A stores at most 56. Hence OBDD-A has a far better space performance than the other two methods.

The only network for which EED_BFS has been shown able to compute 2-REL for moderately large (100+ vertices) networks is for the 2×*W* grid which has width 2. This is true of a number of OBDD-based solutions [4, 6, 17, 26]. The 2×100 grid has 2^{99} paths and is hence intractable for path-based approaches such as SDP. OBDD-based approaches, by contrast, are very suited to this network due to the extremely small width of the network.

Also note the increase in processing time for EED_BFS between the 3×10 and 3×12 networks; the abrupt increase in processing time indicates very poor time performance scalability. Although the length is only slightly higher (12 compared to 10), the performance of EED_BFS worsens rapidly. By contrast OBDD-A shows only a gradual decrease in performance.

Note that we cannot make such comparisons for BS; the reported processing time for the 3×10 network is actually greater than that of the 3×12 network meaning that no accurate conclusions can be drawn. The same occurs for other networks; the 2×8 vertex ring takes longer than the 2×6 vertex ring and has a greatly increased number of nodes and the 2×100 grid network requires less time than the 2×20 grid.

Both BS and OBDD-A perform better than EED_BFS for most of the networks. The performance of OBDD-A is comparable to that of BS for some networks, and better for the rest. Unfortunately there are no tests given for BS on large networks for Model 1.

Comparison with Model 2.1 does not yield significant results, but comparison on small networks for Model 2.2 is possible. While the implementation in this thesis is a general one to allow better comparison between OBDD-A performance on

[10] It should be noted that at the time of OBDD-A being designed, the 2×100 grid was considered to be large for the purposes of network reliability. The BS method raised the bar to far larger networks firstly for ALL-REL in 2005 and then for the more general K-REL in 2007.

different models, computing Model 2.2 without the use of additional heuristics does not allow for a good comparison. It is assumed that the implementations of the BS, EED_BFS and EF algorithms are programmed to take advantage of their particular network models, so using a heuristic specific to Model 2.2 does not invalidate the comparison. The heuristic is described in Section 3.6.4.1, which analyses the performance of OBDD-A on Model 2.2.

Table 3.3: Comparison between BS, EED_BFS and OBDD-A for Model 2.2e

	BS		EED_BFS		OBDD-A		
Network	**Time**	**Nodes**	**Time**	**Nodes**	**Time**	**Nodes**	**Max N**
K6	0.04	221	0.82	173	0.03	351	61
path 18	0.07	160	0.83	232	0.03	288	26
path 19	0.12	2062	2.5	1903	0.04	1784	154
ring 2x6	0.2	174	0.95	221	0.03	246	25
ring 2x8	0.06	258	1.08	295	0.03	357	25
3x10	0.08	219	2.4	184	0.03	369	9
3x12	0.04	269	7.73	226	0.04	455	9
5x5	0.07	822	2.72	697	0.04	1241	69
2x20	0.05	116	1.55	97	0.04	170	3
2x100	0.09	596	1677	497	0.06	890	3

Table 3.3 shows that BS has better time performance than EED_BFS for Model 2.2e, and that OBDD-A (with heuristic) is better than BS. Note that the number of nodes generated is generally least for EED_BFS and greatest for OBDD-A, although the number of nodes stored (and hence memory used) by OBDD-A is at most twice that in the Max N column. Hence OBDD-A is far more space efficient than both other algorithms, in addition to its time performance advantage.

3.6.2.2 Model 'v'

To assess the performance of the implementation for Model '1v', it is compared to the results presented by Xiao *et al*. [13]. While most works on network reliability consider only Model 'e', Xiao *et al*. [13] address Model 'v'. The authors correctly point out that this model is far more appropriate for WSN.

As with previous comparisons, the comparison between the implementation in this thesis and the implementation of the EF algorithm in Xiao *et al*. [13] is not entirely accurate since different computers and different coding standards are used. Hence the comparison with Xiao *et al*. [13] is used as an initial indication that the implementation presented here is competitive in terms of time performance. Since

EF is a factoring algorithm there is no measure of the number of nodes generated; nor is there any other measure of space complexity.

It can be seen that the overheads involved in processing each network are a large part of the OBDD-Av process, as shown by the fact that all OBDD-A processing times in Table 3.4 are relatively similar. The OBDD-Av implementation performs better than EF on the networks tested, except for the 2×10 grid network.

While the difference in computers and network orderings makes the comparison uncertain, it is important to note the increase in processing time as network size increases. For example, the processing time for EF on the 3×10 network is ten times larger than for the 3×5 network, while the difference is negligible for OBDD-Av. A similar difference is seen between the 2×50 and 2×100 networks. This rapid increase in processing time indicates that the OBDD-Av would further improve in comparisons with EF if comparisons were possible on larger networks.

Table 3.4: Comparison between OBDD-Av and EF

	EF	OBDD-Av
Network	Time (s)	Time (s)
2x10	0.0232	0.0356
2x50	0.0792	0.0357
2x100	0.2031	0.0448
3x5	0.0541	0.0343
3x10	0.5603	0.0356
ring 2x10	1.2015	0.0387
path18	0.1135	0.0356
path19	1.3702	0.0372

It is not possible to compare OBDD-Av with EF for Model 2.2 since no results for this model are given [13]. It is similarly impossible to compare space performance.

3.6.2.3 Model 've'

The EED_BFS algorithm by Yeh, Lin and Kuo [26] is designed to be extended to allow both vertices and edges to fail. Direct comparisons between this and the OBDD-A are difficult since the former was run on a SPARC 20 and the latter on a modern PC, however such a comparison can at least indicate some differences. The results are shown in Table 3.5.

Both EED_BFS and the OBDD-A show considerably poorer performance for Model 've' as compared to 'e'. In both cases the number of nodes and processing time (in seconds) in the diagram increases markedly. The OBDD-A also shows the

maximum number of nodes per level of the diagram; the maximum number of nodes in memory at any one time is less than twice this number.

Table 3.5: Performance Comparison for Model 've'

	EED_BFS		OBDD-A		
Network	**Time**	**Nodes**	**Time**	**Nodes**	**Max N**
2x100	2.6	1,286	0.04	2,671	7
2x20	0.03	245	0.03	511	7
3x10	0.73	707	0.04	1,478	31
3x12	3.13	879	0.04	1,834	31
5x5	0.75	5,613	0.08	11,913	656
K6	0.03	305	0.04	1,652	313
Path 18	0.05	601	0.03	1,888	174
Path 19	1.18	8,681	0.13	25,947	1814
ring 2x6	0.12	1,332	0.03	1,092	98
ring 2x8	0.1	888	0.03	1,742	103

Note that the processing times in the table do not vary greatly, especially for the OBDD-Ave. This indicates a large overhead in the OBDD-A implementation which overshadows the small difference between the networks. In general it can be seen that the times for OBDD-A are smaller than for EED_BFS, and the number of nodes stored are far fewer. Hence OBDD-A has better time and space performance on these networks than EED_BFS.

Yeh, Lin and Kuo [26] propose using the incident edge method by Aggarwal, Misra and Gupta [29]. However the resulting OBDD grows in size to a depth of $|V|+|E|+1$, with a corresponding exponential increase in the number of nodes generated. Furthermore it is not clear how the algorithm avoids repeating vertex decisions when two edges have the same vertex as an endpoint.

Other algorithms may be able to be modified directly to generate SDP terms or an OBDD in order to include both vertex and edge failure. However in each case the corresponding increase in complexity is exponential. The same has been shown to occur for the OBDD-A. The use of these methods to compute REL for networks with both vertex and edge failure is hence impractical even for relatively small networks. Hence, for networks with both fallible edges and vertices another method must be developed.

There are no results given for the performance of EED_BFS [26] for Model 2.2ve, and hence no comparison is possible.

3.6.2.4 Performance Comparison Summary

The performance comparisons above show that OBDD-A has far better time performance than EED_BFS and EF. The time performance of BS is closer to that of OBDD-A, especially when the difference in power of the test computers is considered. This holds true for each of the network models addressed by the respective algorithms, despite the OBDD-A implementation being a general one. Note also that OBDD-A is far more flexible that the other algorithms; each of the others addresses only a few network and connectivity models.

The space performance of OBDD-A is superior to that of the other algorithms, including BS. Since OBDD-A keeps less than two levels of nodes in memory at any one time, memory performance improves compared to the other methods as the number of levels of the diagram increases. Memory performance has been referred to as the limiting factor of network reliability [15, 16, 18, 88], which makes the excellent performance of OBDD-A in this area especially useful.

3.6.3. The Effects of F_{max} on OBDD-A Performance

Complexity analysis (see Section 3.5) indicates that the OBDD-A has time and space performance closely linked to the maximum size of F_k. In particular, time complexity is polynomial in the number of vertices, but factorial in the size of F_{max} (since $B_{F_{max}}$ is $O(F_{max}!)$). Further, the OBDD-A was shown to have linear time complexity and constant space complexity on networks that have a constant F_{max} as the size of the network increases.

This section confirms the complexity analysis by experimental means. The effect of keeping F_{max} constant is shown in Section 3.6.3.1, resulting in linear time performance and constant space performance in the size of the network. Section 3.6.3.2 tests the OBDD-A implementation for Model 1 on general networks to show that time performance is worse than exponential, as predicted by the complexity analysis. Space complexity is shown to be somewhat better.

3.6.3.1 Networks with Fixed F_{max}

The complexity analysis in Section 3.5 indicates that OBDD-A has linear time performance and constant space performance for families of networks with constant inter-connectivity; that is F_{max} has a constant bound as $|V|$ increases. Such networks include grid networks and *w*-connected networks.

Table 3.6: OBDD-A Space Performance on Grid Networks

	OBDD-Ae		OBDD-Av		OBDD-Ave	
Network	**Nodes**	**Max N**	**Nodes**	**Max N**	**Nodes**	**Max N**
4x1000	195,587	28	97,720	30	1,149,095	154
4x2000	391,587	28	195,720	30	2,301,095	154
4x3000	587,587	28	293,720	30	3,453,095	154
4x4000	783,587	28	391,720	30	4,605,095	154
4x5000	909,607	28	489,720	30	5,757,095	154

Consider the space performance, as demonstrated in Table 3.6. This shows that for the 4×*L* networks the maximum number of nodes per level (Max N) is constant for each of the component failure models. Because the number of nodes per level has a constant bound, the number of nodes stored in memory at any one time likewise has a constant bound.

The space performance shown in Table 3.6 is linear for each of the component failure models. While this is not apparent from the table, it is clearly shown in Figure 3.17.

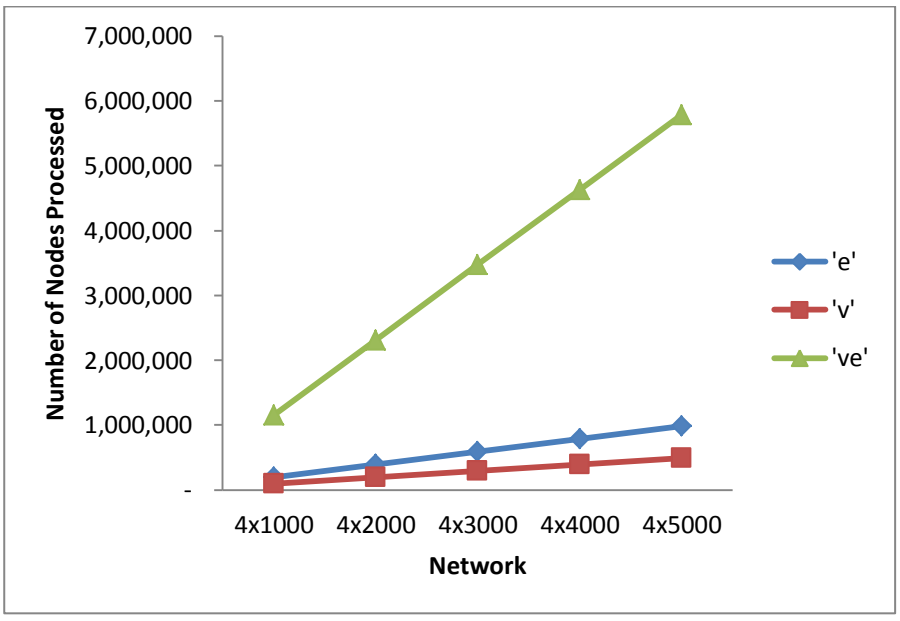


Figure 3.17: Space Performance of 4×*L* Grids

The time performance is clearly shown in Figure 3.18. The processing time increases linearly in the length, *L*, of the network. Recall that grid networks have a linear relationship between |E| and |V|, and hence the processing time performance is linear with respect to the size of the network.

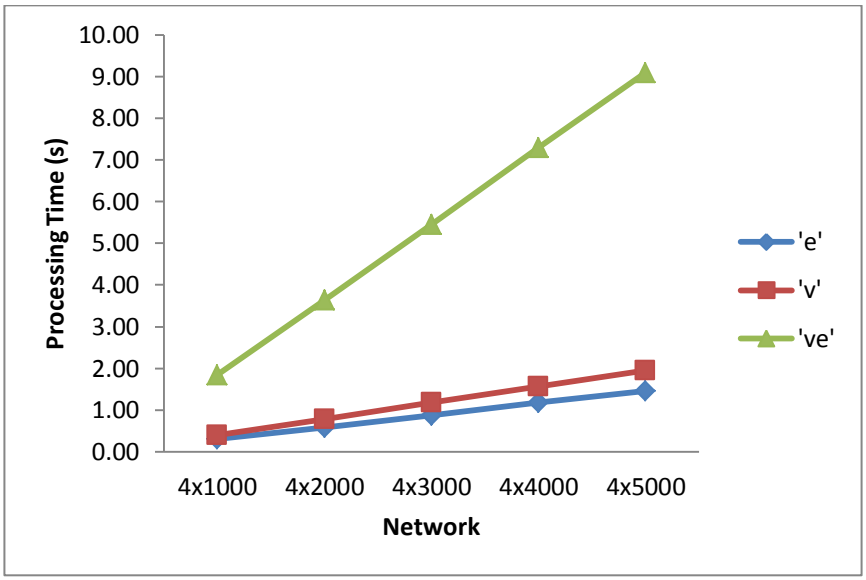


Figure 3.18: Time Performance for 4×L Grids

In each case above, networks have been chosen to best demonstrate the topic under discussion, but the complexity arguments are true in general. The 4×*L* grid networks used above could have been replaced with any network meeting the criteria of constant F_{max}. Similar tests have been carried out on K*w*,*L* networks, other grid networks and ring networks with similar results observed.

3.6.3.2 Networks with Varying F_{max}

Consider the performance of OBDD-A on networks when both |V| and F_{max} are increasing. This is demonstrated by the fully connected networks described in Section 2.10.3. Figure 3.19 shows the logarithm of processing time for the K*w* family of networks, where *w* goes from 5 to 12. The logarithm (base 10) was used in order to make the progression clearer. It can be seen that the progression for Models 'e' and 've' is more than exponential; the fully connected networks form a worst case for these models. The performance of 'v' is far better since fully connected networks are a best case. Note that while the graph connects the discrete data points with lines, this is only to clarify the pattern.

By contrast, the $\log_{10}$ of the number of nodes shown in Figure 3.20 form straight lines for 'e' and 've', indicating that the progression is exponential in either the number of vertices or F_{max}. Recall that for fully connected networks F_{max} = |V|. The progression for 'v' curves downward, showing less than exponential progression. OBDD-Av processes *w* nodes for each K*w* network, making the relationship linear.

The same behaviour holds for the maximum number of nodes per level, and hence the space performance of the OBDD-A as seen in Figure 3.21.

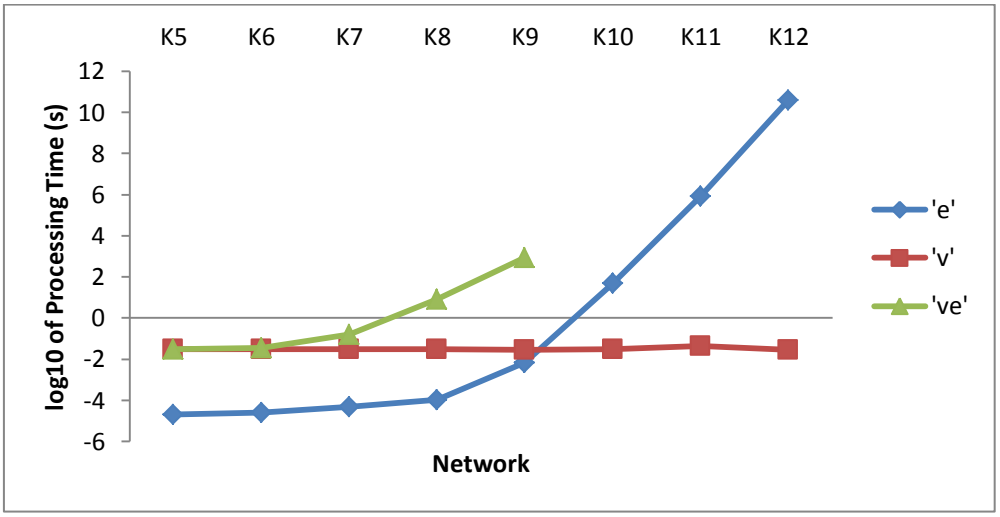


Figure 3.19: Time Performance of OBDD-A on Fully Connected Networks

From the graphs, it can be seen that OBDD-Ae has better time and space performance than OBDD-Ave on the same network. Not only does the OBDD-Ave diagram have more nodes per level than the comparable OBDD-Ae, it also has more levels. However the graphs do not make the general performance of OBDD-Av clear. A better comparison of the three models under OBDD-A is needed.

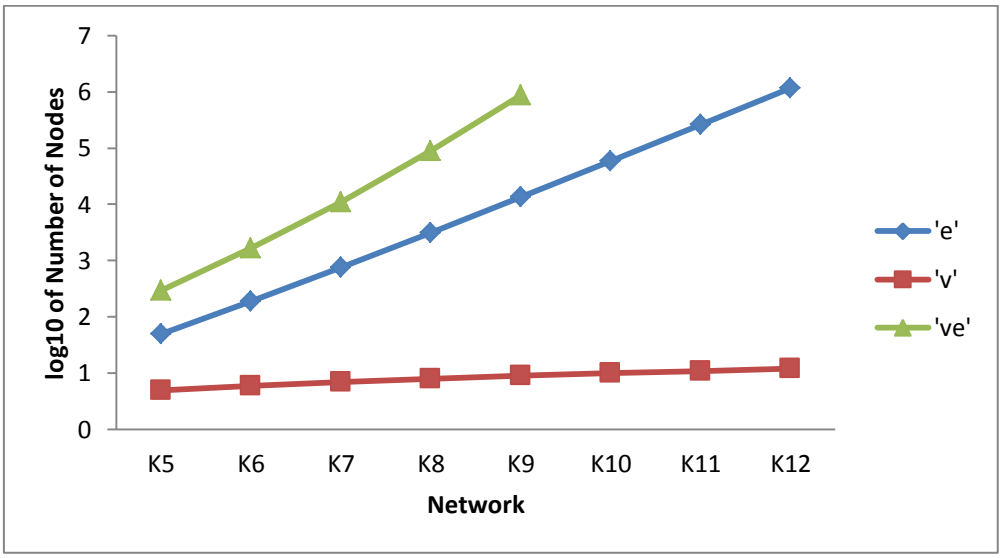


Figure 3.20: Number of Nodes for OBDD-A on Fully Connected Networks

Consider the three large networks shown in Table 3.7, with the maximum number of nodes per level shown as ‘M N’. It can be seen that the OBDD-Av diagram has fewer nodes than the corresponding OBDD-Ae diagram; although it has slightly

more nodes per level it has fewer levels. The OBDD-Ave diagram has both more nodes per level and greater depth, leading to poor performance.

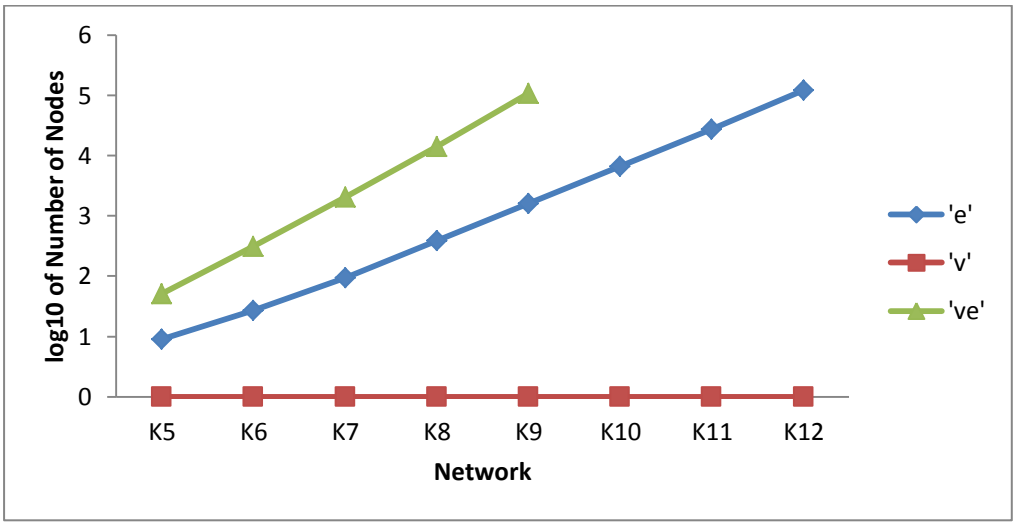


Figure 3.21: Space Performance on Fully Connected Networks

For very large networks, such as those shown in Table 3.7, the computation of Model 've' becomes problematic. The depth of the OBDD-Ave for this network is 439,997 as compared to 160,001 for Model 'v' and 279,997 for Model 'e'. Both the time and space performance for the computation of REL for the 4×40,000 grid are reasonable, because F_{max} is low.

Table 3.7: Performance for Different Models on Large Networks

	OBDD-Ae			OBDD-Av			OBDD-Ave		
Network	**Time**	**Nodes**	**MaxN**	**Time**	**Nodes**	**MaxN**	**Time**	**Nodes**	**MaxN**
4x40,000	21.8	78,939,587	28	0.4	97,720	30	2.3	1,149,095	154
5x15,000	112.1	12,147,644	90	28.4	5,953,539	103	507.0	114,873,613	843
7x1,000	103.5	12,951,730	1,001	64.6	6,431,280	1,293	48,806.1	360,533,359	30,102

The performance for the 5×15,000 grid is notably worse, despite F_{max} increasing by only one. The space performance is still good (a maximum of 843 nodes per level) but it takes more than 8 minutes to compute. The 7×1000 grid is even worse, despite the drastically reduced number of vertices. The increased F_{max} leads to a greatly increased maximum number of nodes per level and hence an extremely large number of diagram nodes overall. The time performance worsens dramatically for the 7×1000 grid. While the 4×40,000 grid takes 2×10^{-6}s (= $^{2.26}/_{1,149,095}$) per node of processing time and the 5×15,000 grid takes 4.4×10^{-6}s, the 7×1000 grid requires 1.4×10^{-4}s. Recall that the dominant time complexity term in the analysis in Section 3.5.3 relates to the isomorphism comparison of each new

node against existing nodes on that level. The increased per-node processing time reflects this.

This indicates that, in order to solve Model 've', a method is required to reduce both the number of diagram levels as well as the maximum number of nodes per level

3.6.4. The Effects of Communication Models on OBDD-A Performance

Section 3.6.3 deals specifically with the performance of OBDD-A on networks under Model 1. The behaviour of OBDD-A as |V| and F_{max} changes is similar for other network communication models, and hence is not repeated. However the performance of OBDD-A as network communication models vary is yet to be addressed. This section addresses the remaining network models in turn.

The performance of OBDD-A is demonstrated using the 5×5 grid network. This network has sufficient F_{max} to avoid being trivial, and enough vertices on each opposing edge of the network to allow for some variety in |T|. The network is shown in Figure 3.22, with the source (v_0) and target (v_{24}) marked. When additional targets were needed, vertices on the right-hand side of the grid were chosen. Hence for two targets, the vertices v_{23} and v_{24} are used, for three targets v_{21} is also used, and so on.

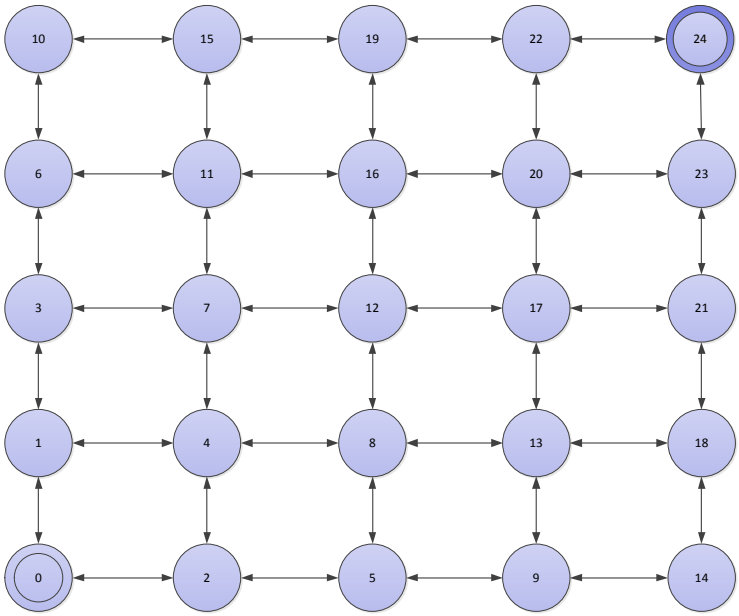


Figure 3.22: 5x5 Grid Network

While the performance of the OBDD-A as the network models change is the main focus of this section, Model 3 has not been addressed in reliability research previous to the work making up this thesis. For this reason, the reliability of a network under these models is also of some interest, and is addressed along with the OBDD-A efficiency.

3.6.4.1 Model 2 – REL(s,T)

The broadcast model, Model 2.1 from Table 2.1, is common in computer network communication and often referred to as K-REL. The general formulation of the K-REL problem is that a set K of network vertices must be mutually connected. This problem is identical to Model 2.1 for undirected networks; for directed networks Model 2.1 is generally used even when K-REL is being referred to [17]. A second common problem is ALL-REL, the probability that all vertices in the network are connected. This is Model 2.2, and is a special case of Model 2.1 with V = T+{*s*}. This section addresses both K-REL and ALL-REL.

For K-REL, the communication is successful if a message can reach each target vertex in T from the source vertex. There is a greater chance that the communication is not successful since more target vertices must be reached. Without the use of heuristics, processing of nodes will reach deeper down the tree until nodes are found to be terminal. In addition, the tracking of those targets that have been reached reduces isomorphism on lower levels, which also increases the number of nodes. These trends can be observed in Figure 3.23 and Figure 3.24, which show the increase of the number of nodes generated and the decrease in reliability, respectively, as |T| increases.

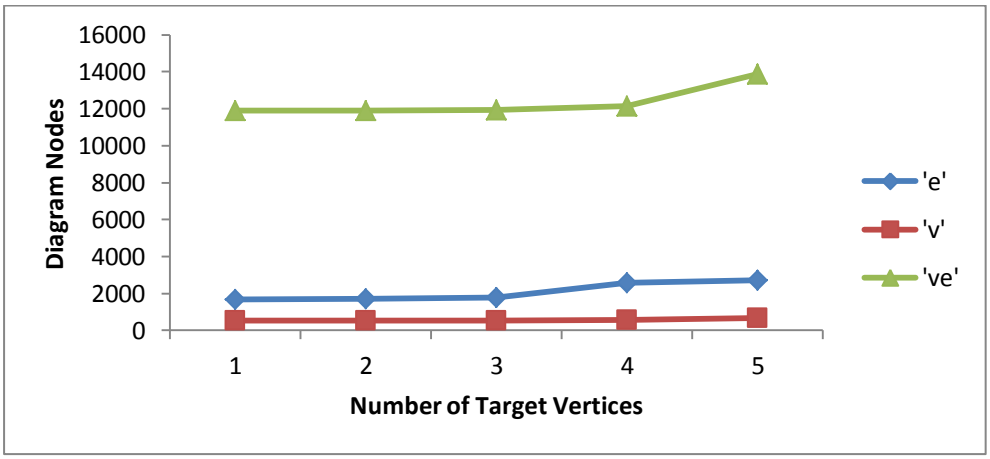


Figure 3.23: Diagram Nodes for 5×5 Grid with Varying |T| - Model 2

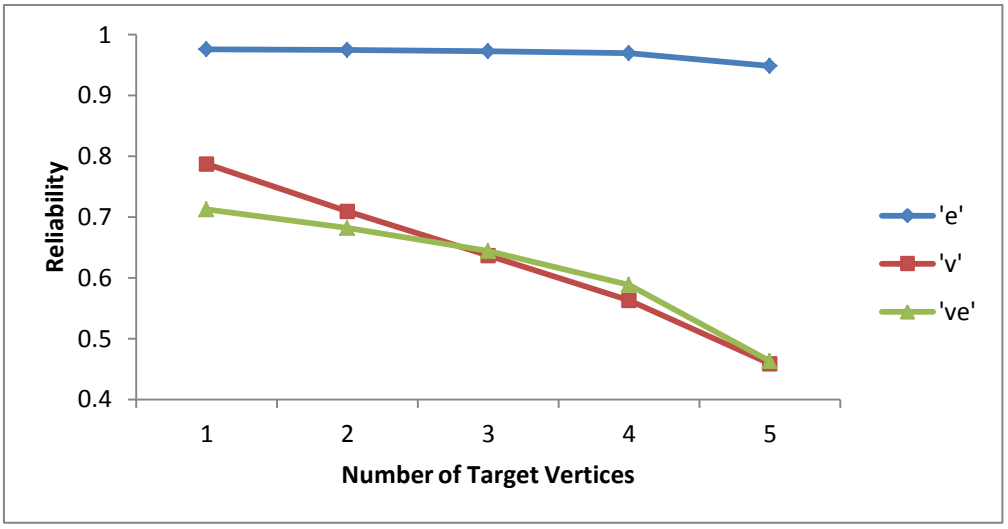


Figure 3.24: Reliability for 5×5 Grid with Varying |T| - Model 2

Because the OBDD-A tracks all target vertices reached, the performance of OBDD-A for ALL-REL is not optimal. For example the 5×5 grid network, which processes 1,694 nodes for Model 1e, requires 1,799,933 nodes for ALL-REL. The OBDD-A has no easy way to track vertices in conditions – other than the endpoints of the conditions of course. It can be achieved by comparing the targets in a condition being deleted to the reached targets of the node, and then searching all other conditions for this target. Doing so is inefficient compared to BS.

While creating a heuristic identical to the BS treatment of ALL-REL is complex, a simple heuristic is able to greatly improve performance. In this case, a check is inserted into DEL-REDUNDANT that tests whether a redundant vertex has been reached (*i.e.*, is present in VS_i). If not, all conditions are tested to see if the redundant vertex is in the condition but not an endpoint (since conditions with a redundant endpoint are deleted). If both tests are false, the node is set to be failed and all vertices are deleted from VS_i.

This test correctly identifies cases when a newly redundant vertex has not been reached and is no longer reachable via conditions, meaning the vertex is unable to be reached, but it misses some negative nodes; a vertex interior to a condition (in the condition but not an endpoint) may still become unreachable if the condition is later deleted. Catching these additional nodes would involve checking all conditions on all nodes however, which is too computationally expensive.

Using the heuristic, the number of nodes processed is reduced from 1,799,933 to 1,241, which is less than for Model 1e (*i.e.*, 1,694). However even with the heuristic OBDD-A cannot compute ALL-REL for large networks due to the expense of keeping track of every reached target vertex, either by not deleting them or by storing them separately. For this reason, we recommend that OBDD-H (as described in Chapter 4) be used for computing ALL-REL.

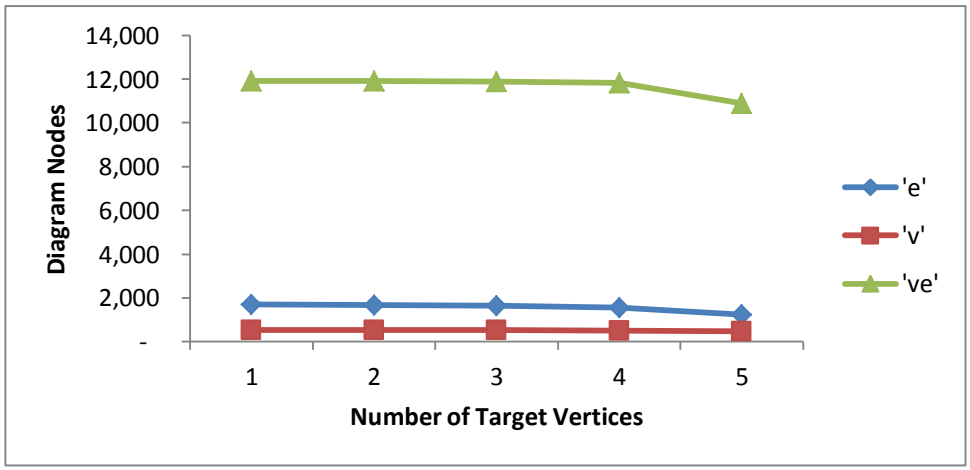


Figure 3.25: Diagram Nodes for 5×5 Grid with Varying |T| - Model 3.1

3.6.4.2 Model 3.1 – REL(s, 1-of-T)

Model 3.1 has multiple target vertices but only a single target needs to be reached in order for the communication to be successful. This results in a successful communication being more likely and it is also more likely to find such a communication at higher levels of the diagram. Hence it is expected that as |T| increases, the number of nodes in the network will decrease.

The experimental results of varying the size of |T| for the 5×5 grid network are shown in Figure 3.25. As can be seen, the number of nodes does decrease slightly as |T| increases. The processing time decreases in a similar manner.

The reliability of the network increases as |T| increases, as shown in Figure 3.26. It can be seen that the probability for ‘e’ is much larger than for the other two network models. A large part of this is the probability that the single source vertex fails for Models ‘v’ and ‘ve’, which immediately causes the communication to fail.

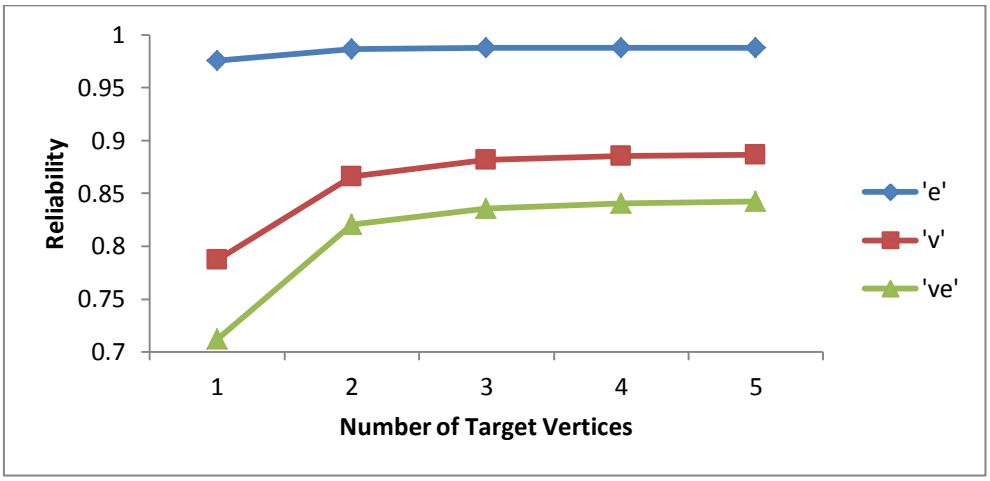


Figure 3.26: Reliability for 5×5 Grid with Varying |T| - Model 3.1

3.6.4.3 Model 3.2 – REL(s, c-of-T)

Communication Model 3.2 not only has varying size of |T| but also varying number *c* of targets needing to be reached. Instead of a network state being successful when a target vertex is reached (and decided as active), at least *c* target vertices need to be reached. This means that, as *c* increases, nodes will be found successful at lower levels of the diagram. This behaviour is shown in Figure 3.27.

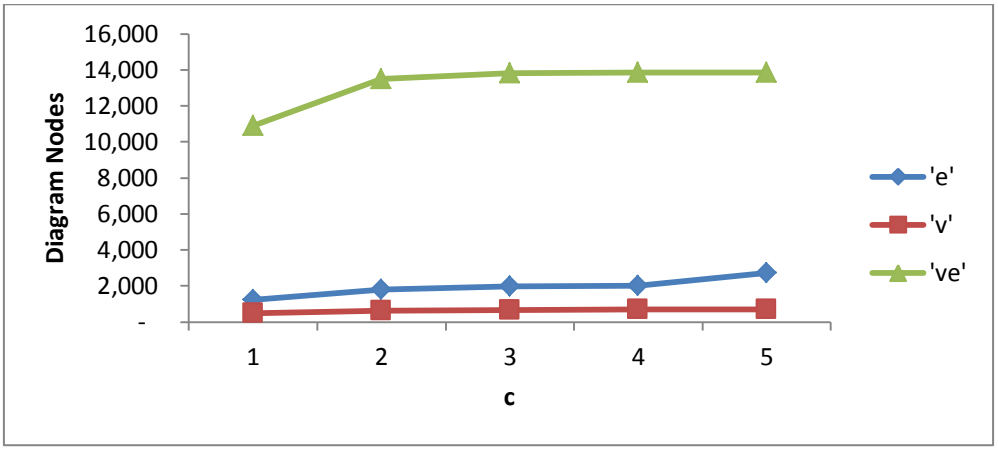


Figure 3.27: Diagram Nodes for 5×5 Grid with Varying *c* – Model 3.2

For the same reason, the chance of successful communication decreases as the number of targets requiring to be reached increases. Hence the reliability of the network decreases as *c* increases, as shown in Figure 3.28.

Again, the reliability for Model ‘e’ is noticeably higher than for the other two models.

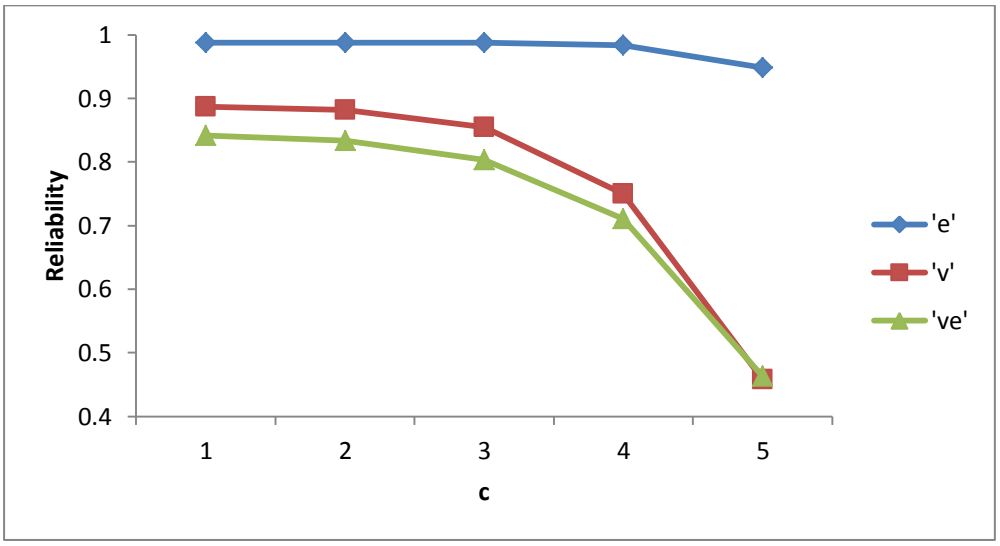


Figure 3.28: Reliability for 5×5 Grid with Varying *c* – Model 3.2

3.6.4.4 Model 3.3 – REL(s, c_i-of-T_i)

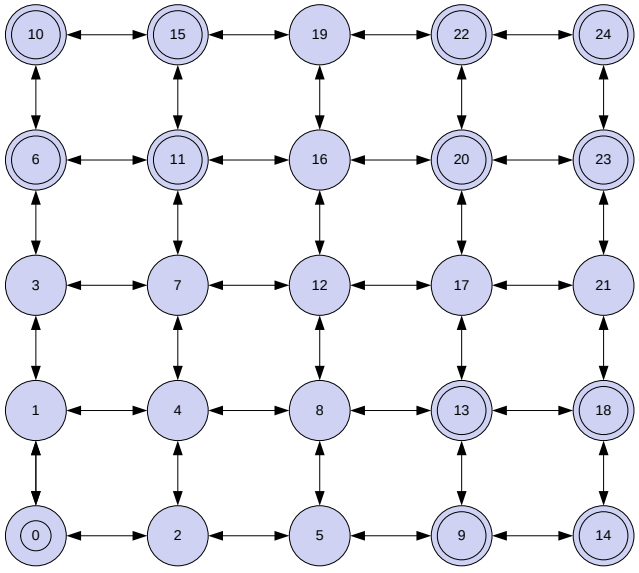


Figure 3.29: 5×5 Grid with Grouping – Model 3.3

Finally, communication Model 3.3 follows the general pattern established above. Three groups of four vertices were chosen on the 5×5 grid as shown in Figure 3.29, representing clusters of sensor nodes around three separate events or redundant device arrays. This means that the number of target vertices increases, but there are restrictions on which target vertices must be reached. For example, if c_i=2 for all groupings, at least two target vertices from each grouping must be reached. If all vertices in one grouping are reached but no vertices in another, the communication is considered failed.

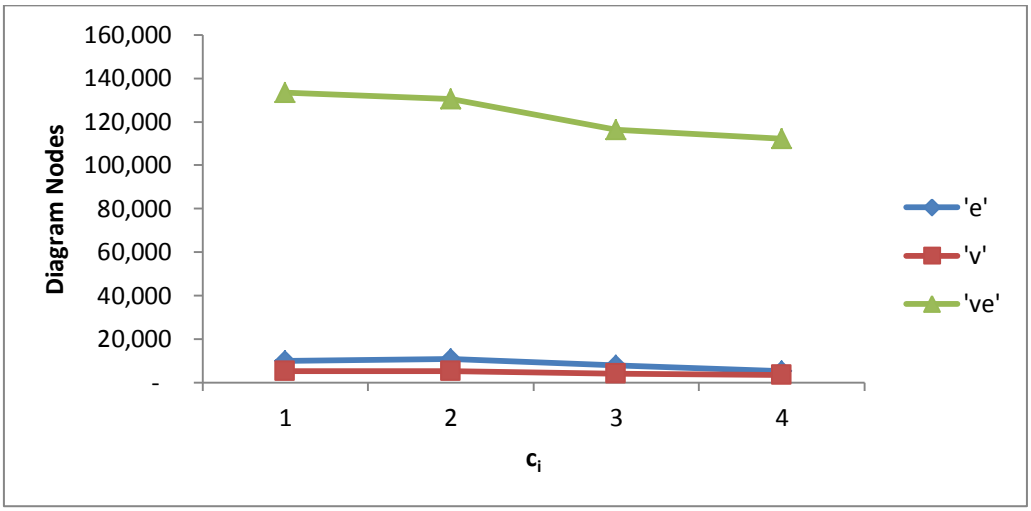


Figure 3.30: Diagram Nodes for 5×5 Grid with Varying c_i – Model 3.3

Because of the multiple groupings, the effect of increasing each c_i is even more dramatic than for Model 3.2. The graph in Figure 3.30 shows that the number of diagram nodes generally decreases as c_i increases; c_i is set to be equal for each grouping. The scale of the graph hides that the number of nodes increases from c_i=1 to c_i=2 for Models 'e' and 'v' before beginning the decrease. The two factors affecting the number of nodes are nodes being more easily found successful for low c_i and more easily found unsuccessful for higher c_i. The latter comes about because, as discussed in Appendix A, a node can be declared failed when it is no longer possible to reach enough target vertices.

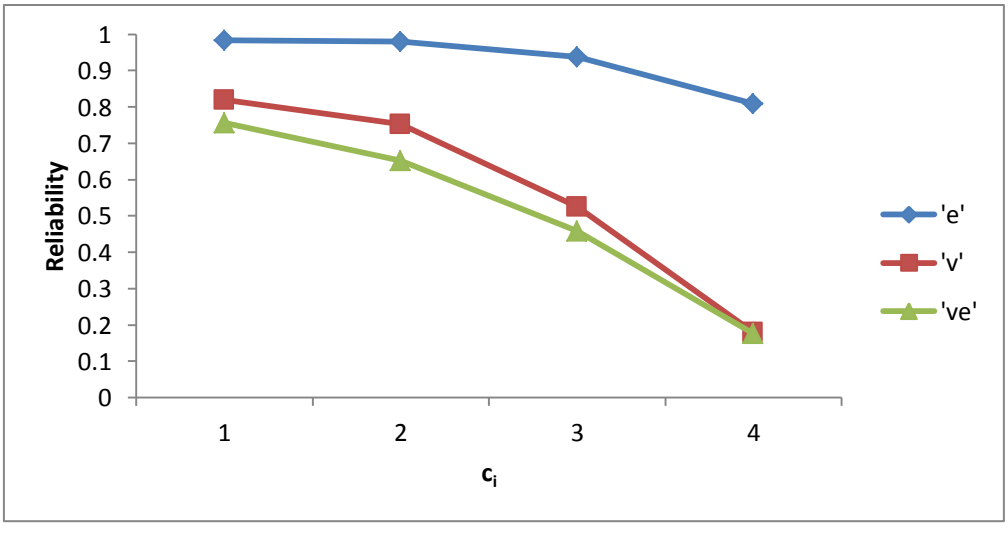


Figure 3.31: Reliability for 5×5 Grid with Varying c_i – Model 3.3

As expected, the reliability of the network decreases sharply as c_i increases, as shown in Figure 3.31. Once again the probability for Model 'e' is much higher than that for 'v' and 've'.

3.6.5. Performance Example

Consider a large number of wireless sensor nodes that have been deployed in a grid pattern with 8 nodes per side as shown in Figure 3.32, with the node marked *s* being the base station which allows the user to interact with the network. Three events are sensed, close to the nodes marked E1, E2 and E3. Each of the nodes adjacent to the event nodes can also sense the event. These nodes are shown inside box shapes and referred to as the *event nodes* for this example. Every WSN node can communicate with its closest neighbours as shown by the lines linking nodes.

Assume that this WSN is a security network with motion sensors and/or cameras designed to catch intrusion, and that each event is an intrusion (possibly by the same intruder). This example considers three connectivity cases; a simple alarm, a redundant alarm and an intrusion position system. Nodes in a WSN are considered to be relatively prone to failure [2] and hence a vertex failure model is appropriate in each case, assuming that the intruder isn't deliberately attempting to jam wireless communications in some way. In each case assume that each device is 99% likely to function (*i.e.*, that WSN nodes fail with a probability of 0.01).

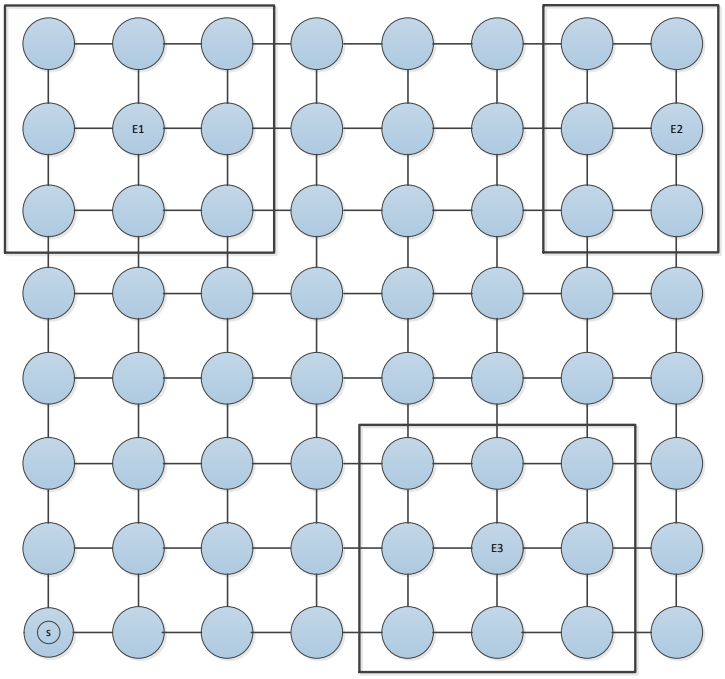


Figure 3.32: Grid Sensor Network

First consider the case of a simple alarm system; when any sensor detects an intruder an alarm is immediately raised. This is an instance of Model 4.1v, and will be modelled by a reverse network under Model 3.1v. In this case the set of target

vertices are exactly the event vertices (all vertices inside a box), giving |T| = 24. The source vertex is the base station, marked *s*. OBDD-A can be used to compute the reliability (chance of successfully raising the alarm) as 0.8876 in 0.31 seconds. This requires processing 10,765 nodes, with at most 509 on any level (and hence less than 1018 stored in memory at any one time).

Secondly, consider the case where at least two alerts need to be received before an alarm is raised, in order to reduce the chance of false alerts. In this case Model 4.2v is appropriate, with c=2, and again the reverse method is used and the OBDD-A is applied to Model 3.2v. In this case the reliability (chance of successfully raising the alarm) was computed as 0.8874. This requires processing 60,007 nodes, with a maximum of 2,717 nodes per level and took 1.40 seconds.

The second case above allows the two alerts to be detected in separate areas. It would be more realistic to require that both alerts should be in the same area, which would require a correct connection to at least two sensors in each of the three sensory groupings. This is an instance of Model 4.3v with c_i=2 for each group g_i. Each of the three groups contains the vertices inside one of the 'event' boxes. In effect, this is equivalent to solving three instances of Model 4.2v simultaneously. The OBDD-A computes the reliability in this case to be 0.8442. This requires processing 11,819,041 nodes, with a maximum of 888,625 nodes per level and took around two weeks[11]. Solving three NP-Complete problems simultaneously has the expected effect of an exponential increase in the time required, largely due to isomorphism comparisons. The problems cannot be solved separately and then combined, however, since they are not disjoint.

It is important to realise that, to the best of our knowledge, network Models 3 and 4 are not solvable using any algorithm other than those presented in this thesis. Existing algorithms are only able to solve the case where one or all of the targets must be reachable with the source, and grouping of sources and targets is not possible. Other examples of when these models could be valuable include environmental data collection, sensor networks for monitoring aged or disabled persons, networks requiring triangulation or other location (such as mobile phone networks and the use of GPS) and hazard detection. Power transmission networks that have critical areas (such as hospitals or the homes of people requiring life support machinery) can also be modelled in this way.

[11] The algorithm was run at standard priority on a Windows system that was used for other tasks as well during this time, so an exact time would be misleading.

3.6.6. Performance Summary

The formal complexity measure of many algorithms takes into account only the size of the input (in this case the edges and/or vertices of the input network). This is not sufficient for the OBDD-A and other BDD-based algorithms whose performance is greatly affected by F_{max}. For example, the OBDD-A computes the REL of a 160,000 vertex network (4×40,000 grid) in less than 20 seconds while requiring more than 24 hours for the 13 vertex K13 network. The former has F_{max}=4, while the latter has F_{max}=12.

Complexity analysis has shown that, while the OBDD-A is factorial in F_{max}, its performance for large networks that have low F_{max} is impressive. In particular, if F_{max} is kept constant then increasing the number of vertices and edges has been shown to have only a linear effect on processing time and not to affect space performance at all. This property was shown in the complexity analysis in Section 3.5 and verified through testing.

The OBDD-A has been shown to compute REL for communication Models 1, 2 and 3 from Section 2.3, as well as reversed Model 4 networks. Further, all three component failure Models ('e', 'v', and 've') are solvable. The time performance of the OBDD-A is at worst comparable with existing algorithms and the space performance is far superior.

The details of the implementation have a marked effect on the time performance of the algorithm. As an example, switching from a simple queue-based model for Q_N to a queue with hashed indices reduced the time performance by roughly a factor of 10. Both methods are 'correct' implementations of the algorithm, but their time performance is very different. A similar effect was gained by sorting VS_i and CI_i.

Neither choice affects the space performance, however, since that is measured by the number of nodes. Because the algorithm specifies when to merge nodes and how to create child nodes, all correct implementations of the algorithm will yield the same number of nodes for any particular ordered network[12].

Hence we assert that the number of nodes generated by the algorithm is a better measure of performance than the processing time. When the maximum number of

[12] If the network files have different orderings, this does not apply. Indeed one implementation is likely to produce different results for two network files representing the same network but with different orderings. It is assumed that the mechanism used by the implementation to load the network files into memory does not change the ordering. Note also that heuristics can be used to reduce the number of nodes processed, as discussed in Chapter 6.

nodes on any one level of the diagram is also considered, the measure becomes more accurate.

3.7. Chapter Summary

This chapter has introduced the OBDD-A, and shown that it is a useful tool for computing the reliability of a wide range of network models. The OBDD-A can be used to compute REL for networks with multiple source or target vertices, including those where target vertices are divided into groups. Furthermore it has been shown that while the processing time of the OBDD-A is competitive for such networks when compared to existing solutions, the memory required is extremely small.

The OBDD-A has been shown to have the useful property for groups of networks whose size increases while F_{max} remains constant; linear time performance and constant space performance in terms of the size of the network. This allows for extremely large networks to be computed with relative ease. For example the 4×40,000 grid network takes around 20 CPU seconds to compute and stores less than 60 nodes at any one time.

While the Boundary Set algorithm (BS) proposed by Hardy, Lucet and Limnios [15, 18] is as efficient in terms of processing time, the OBDD-A is superior in terms of memory required. In addition, BS is only applicable to undirected networks and failure Models 1e and 2e. However BS is particularly useful for Model 2e, and uses efficient notation for undirected networks. Further, the time performance of the OBDD-A is taken on a faster computer, indicating that BS may be more than competitive. Hence it would be useful to combine the efficient BS notation with the space efficiency of the OBDD-A. Chapter 4 discusses BS and introduces a hybrid OBDD-A which makes use of BS notation.

Chapter 4
Hybrid Ordered Binary Decision Diagram

4.1. Chapter Overview

This chapter presents the Hybrid Ordered Binary Decision Diagram (OBDD-H) that combines the benefits of the Ordered Binary Decision Diagram (OBDD-A), described in Chapter 3, and the Boundary Set (BS) [15, 18].

OBDD-H improves BS in three ways. Firstly, OBDD-H can be used to solve all communication models and component failure models, described in Table 2.1. On the other hand, BS is only applicable to Models 1 and 2 with edge failures and perfect vertices (*i.e.*, Model 'e'). This chapter describes how OBDD-H extends the BS notation to enable the computation of REL for Models 1, 2 and 3 under any of the component failure models. Secondly, OBDD-H stores the diagram nodes of no more than two levels at any time, as with OBDD-A. This feature significantly reduces the run-time memory requirement of BS. Thirdly, like OBDD-A, OBDD-H can be used to calculate the network performability that includes delay measure, described in Chapter 5. On the other hand, BS is restricted to computing only the reliability metric.

OBDD-H uses the more efficient BS notation to improve the time performance of the OBDD-A for communication Models 1 and 2 under Model 'e'. However, like BS, OBDD-H is only applicable to undirected networks.

The OBDD-H presented in this chapter is an extended version of preliminary work in [3,4], which addressed only Model 'e'. The initial version of the OBDD-H [102] stored the partition numbers of each node's partition of the BS, F_k, however this was shown [106] to be less efficient than storing the partitions directly. This thesis describes only the latter version of the OBDD-H.

Several key methods and concepts of OBDD-H are similar to those in OBDD-A. For convenience, this chapter repeats relevant information presented in Chapter 3 where it is deemed to aid in the understanding of the OBDD-H.

The remainder of this chapter is organized as follows. Section 4.2 discusses the theoretical aspects of the OBDD-H for Models 1e and 2e, including the computation of F_k. The pseudo-code for this OBDD-H is given in Section 4.3, with

an example of its construction in Section 4.4. Section 4.5 introduces an extension of BS notation to allow computation of Models 'v' and 've' as well as connectivity Model 3, with an example of such an OBDD-H in Section 4.6; we call this generalized algorithm as General OBDD-H, or OBDD-H. Section 4.7 discusses the differences and similarities between the OBDD-A and OBDD-H algorithms. The performance of the OBDD-H on the various component failure and connectivity models is discussed in Section 4.8 and Section 4.9 concludes the chapter.

4.2. Mathematical Model for OBDD-H

4.2.1. Nodes and Levels

Let $\Theta(N,G)$ denote the OBDD-H for graph G(V,E), where N is the set of OBDD-H nodes $\{N_0, N_1, \ldots, N_{2^{|E|}-1}\}$. Without loss of generality, let N_0 be the root node and for any parent node $N_i \in N$, let N_{2i+1} be the left (or negative) child node and let N_{2i+2} be the right (or positive) child node. Once child nodes are generated the parent node is no longer required. This makes linking irrelevant and hence the nodes of $\Theta(N,G)$ are only implicitly linked. Consequently, less than two levels of the diagram will reside in memory at any one time.

Each $\Theta(N,G)$ is divided into a number of levels, with node N_0 on level 0 (the highest level), N_1 and N_2 on level 1, N_3 to N_6 on level 2, and so on. Thus any node N_i is on level k if and only if $2^k-1 \leq i \leq 2^{k+1}-2$. Each level k of $\Theta(N,G)$ (except the last) represents an evaluation/decision of a variable.

The OBDD-H stores information on the network state in each node on level k+1 based on the variable decisions for nodes on level k, which in turn are updated based on the variable decisions on nodes in level k-1, and so on. Therefore at any time OBDD-H algorithm stores nodes on only two consecutive levels; each parent node is removed from level k in turn and processed to give two child nodes on level k+1. Each OBDD-H node, $N_i \in N$, contains an information pair $[Part_i, P_i]$. This pair describes the network state that the node represents, with P_i being the probability of being in the network state represented by N_i. Section 4.2.2 describes the details of partition information $Part_i$.

4.2.2. Partitions

Following the notation in BS [15, 18], the BS for level k of the OBDD-H is defined as:

$$F_k = \{v_x | \exists \text{ edges } e_y = \{v_a, v_x\} \text{ and } e_z = \{v_x, v_b\} \text{ with } a \leq k,\ b > k\} \quad (4.1)$$

Eq. (2.2) is a formalization of the definition from [15] as discussed in Section 2.9.1. Note that the definition in (2.2) does not apply for F_0; define $F_0=\{v_0\}$. Further, this definition is sufficient for Model 1, 2 and 3 under Model 'e', but not for Models 'v' or 've'. A definition for the latter two Models is discussed in Section 4.5.2.

The OBDD-H uses BS notation (see Section 2.9) to track the connectivity between the vertices in F_k instead of the paths leading to them. While this notation is more elegant than that of the OBDD-A, it requires that all edges in the graph be undirected.

Each OBDD-H node stores a partition in the form of an array or other structure. This partition information is used to generate child nodes. Formally, a node N_i for an OBDD-H contains a pair $[Part_i, P_i]$, where $Part_i$ is the partition information, and P_i is the probability of being in the state represented by $Part_i$.

The format for the implementation of $Part_i$ by Hardy *et al*. [15] (as discussed in Section 2.9) is not necessarily optimal for the OBDD-H. We define $Part_i$ as a set $\{BL_0^i, BL_1^i, \ldots BL_{|Part_i|}^i\}$ where each *block*, $BL_x^i = \left(\{v_0^i, \ldots, v_{|BL_x|-1}^i\}, m_x\right)$, is a set of vertices from F_k, and the Boolean m_x denotes whether the block is marked. The formal definition is shown in Definition 4.2. While the last property is required to make $Part_i$ a partition of F_k it does not hold for component failure models other than 'e'; a modified definition of $Part_i$ for Models 'v' and 've' is discussed in Section 4.5.2.

$$\forall x.\, v_x^i \in F_k,\ \bigcup_x v_x^i = F_k \text{ and } \bigcap_{x=0}^{x=|Part|} BL_x^i = \{\} \quad (4.2)$$

The blocks of $Part_i$ are sorted in increasing order of least vertices; so the vertex in F_k with the lowest subscript is always in BL_0^i, BL_1^i contains the next lowest vertex in F_k that is not in BL_0^i, and so on. This ordering makes comparing and manipulating partitions more efficient. The vertices in each block are sorted in order of increasing subscripts.

4.2.3. Node Type

OBDD-H nodes are divided into terminal and non-terminal nodes. A node is terminal if it does not have child nodes and each non-terminal node has exactly two child nodes. A terminal node can be a failure node or a success node. A node is

successful if the source and all target vertices are known to be connected and a failure node when they are known to be disconnected. In other words, when a node represents a state from which it is impossible to meet the requirements for the problem (*i.e.*, if it is no longer possible to send a message from the source to the sink vertex) it becomes a failure node. When a node represents a state that meets the requirements then it is a success node.

The type of node N_i depends entirely on $Part_i$. A marked block contains those vertices that are connected to a source or target vertex. Hence if one or more target vertices are in a single marked block, a minpath exists. For example, in Model 2.1e, a node is successful if all non-redundant target vertices are in a single marked block.

The OBDD-H does not need to track redundant target vertices. The failure conditions of the OBDD-H recognize a node as failed if a block that had contained a source or target is found to be disconnected from the rest of the network. In particular, if a marked block becomes empty due to the last remaining vertices in it becoming redundant, then the node is failed.

For the special case of ALL-REL (Model 2.2e) all blocks are considered marked and all vertices are in T; hence to be successful for ALL-REL a node must have all non-redundant vertices[13] in a single block.

4.2.4. Node Isomorphism

Two nodes N_i and N_j at the same level of an OBDD-He are isomorphic if $Part_i$ = $Part_j$.

Partitions $Part_i = \{BL_0^i, \ldots, BL_x^i\}$ and $Part_j = \{BL_0^j, \ldots, BL_y^j\}$ are equal if $x = y$ and ($BL_z^i = BL_z^j$ for $z = 0,\ldots, x$). Finally, two blocks $BL_a^i = (\{v_0^a, \ldots, v_\alpha^a\}, m_a)$ and $BL_b^j = (\{v_0^b, \ldots, v_\beta^b\}, m_b)$ are equal if $\alpha = \beta$, $v_u^a = v_u^b$ for $u = 0 \,..\, \alpha$, and m_a=m_b. The definition requires that the blocks of each partition are ordered as discussed in Section 4.2.2. This isomorphism definition is equivalent to that used for BS[15, 18]; if two BS nodes are isomorphic then the equivalent OBDD-H nodes will also be isomorphic.

Each partition of F_k can be translated into a unique partition number [15, 18]. One advantage in the use of partition numbers is that they can be compared easily; if the

[13] Note that 'all non-redundant vertices' includes vertices that have not been in F_k yet; hence such a node is only successful if all non-redundant vertices are in F_k.

partition numbers of two nodes are equal then the nodes are isomorphic. The OBDD-H stores partitions as structures instead of as numbers, and hence it compares the partitions directly. While comparing two numbers is generally more efficient than comparing two structures, the numbers in question quickly grow large enough to require special libraries to compare. Using partitions directly has been shown to be more efficient for the OBDD-H [106].

The isomorphism test for the OBDD-H is more efficient than that for the OBDD-A because it will compare at most $|F_{k+1}|$ vertices instead of a potentially larger number of conditions as discussed in Section 4.7. The exact performance of the OBDD-H partition comparison depends on its implementation; one efficient method is to use a dynamic array for each block with another dynamic array containing all of the blocks. Using this approach rather than the one proposed by Hardy, Lucet and Limnios [15, 18] is slightly less efficient but allows the computation of other models as discussed in Section 4.5.

Merging two isomorphic nodes is straight-forward; since nodes are identical in everything except probability, the existing node's probability is incremented by the probability of the newly created node.

Given that the child nodes created by the BS and OBDD-H methods are equal, that the isomorphism definitions between both methods are equivalent, and that testing for success and failure is also equivalent, the shape of the BS and OBDD-H diagrams will be identical. In other words any two equivalent nodes in BS and OBDD-H will generate equivalent children. Hence the number of nodes generated is the same for both the BS and the OBDD-H methods when applied to the same network with identical ordering[14].

4.3. The OBDD-H Algorithm

The OBDD-H algorithm for Model 2.1e (K-REL) is shown in Figure 4.1. This algorithm also functions for Models 1e and 2.2e; a more general OBDD-H algorithm is described in Section 4.5.

The algorithm starts by initializing the diagram (line 1) and assorted variables (line 2). F_0 is computed (line 3) as discussed in Section 4.3.1 before the main loop (lines 4-17) starts. Note that the initialization of F_0 and F_1 also initializes REDF and REDT

[14] Identical ordering refers to the exact ordering of each vertex and edge, not just the ordering strategy. Both the published BS [15,18] and OBDD-H use a breadth-first ordering strategy, however both strategies can produce files with slightly different orderings. Indeed the strategy used for this thesis is dependent on the original ordering, and hence two different instances of the same network could be assigned different orderings.

in a similar way to that in UPDATE-LEVELH; for example if v_f is not in F_1 then v_f is redundant and REDF is **TRUE**.

```
OBDD-H (G):
1)  Initialize the root node N0=[( S, 1.0 ), { }];
2)  Initialize (k=0, QC = { N0 }, QN = { }, reliability = 0 };
3)  Initialize redf, redt, F0 and F1;
4)  Remove the first node, Ni, from QC;
5)  N2i+2 = CREATE-POS-CHILDh(Ni, k );
6)  N2i+1 = CREATE-NEG-CHILDh(Ni, k );
7)  for each child node Nfi do
8)      if redf then
9)          DEL-REDUNDANTh( Nfi, vf );
10)     if redt then
11)         DEL-REDUNDANTh( Nfi, vt );
12)     if (NON-TERMINAL-NODEh(Nfi, reliability)) then
13)         Check each node on QN for isomorphism with Nfi;
14)         if ( an isomorphic node Nx was found ) then
15)             Px = Px + Pfi;          // Merge nodes
16)         else
17)             Add Nfi onto QN;
18) if ( QC == { } ) then
19)     if ( QN == { } ) then
20)         return reliability;
21)     else
22)         UPDATE-LEVELh( k );
23) goto 4)
```

Figure 4.1: OBDD-H Algorithm for Model 2.1e

The main loop removes a node from Q_C (line 4) and processes it to create two child nodes (lines 5-6). These child nodes then have redundant information removed (lines 8-11), are tested for termination (line 12) and either merged with an isomorphic node (lines 13-15) or stored (line 17). The loop condition then checks the queues (lines 18-19); if both are empty the algorithm terminates (line 20) and if only Q_C is empty the level of the diagram is incremented (line 22).

The computation of F_k and incrementing of diagram levels are discussed in Section 4.3.1 and the methods of creating child nodes using BS notation are introduced in Section 4.3.2. Finally, the detection of success and failure nodes is covered in Section 4.3.3.

4.3.1. Computing the Boundary Set

Every level k of the OBDD-H requires the computation of F_k, as well as the updating of various variables. This update is gathered into the UPDATE-LEVELH method, as shown in Figure 4.2.

```
UPDATE-LEVELh ( k ):
1)   k++;
2)   Swap Q_C and Q_N;
3)   UPDATE-FKh( k );
4)   return;
```

Figure 4.2: UPDATE-LEVELh for Model 2.1e

For every level *k*, OBDD-He requires the BS for two levels of the diagram; the levels represented by Q_N and Q_C. The BS for level *k* (Q_C) is F_k, and for *k*+1 (Q_N) is F_{k+1}. These are computed by the UPDATE-FKH method show in Figure 4.3. F_0 and F_1 are initialized in the main algorithm; F_0 is always $\{v_0\}$ and F_1 is either $\{v_0,v_1\}$ or $\{v_1\}$ depending on whether or not v_0 becomes redundant after the first level.

When computing the BS at level *k*, F_k is already known because this was F_{k+1} for the previous level. Hence UPDATE-FKH starts by copying the old F_{k+1} to the new F_k (line 2)[15]. If vertices are deleted in the transition from F_k to F_{k+1}, the REDF and REDT flags are set for vertex v_f or v_t respectively (lines 6 and 12). These flags are both set to **FALSE** by default (line 1).

The first test (line 3) checks to see if v_f is still present in e_{k+1}; if it isn't then v_f is redundant. In this case v_f is removed from F_{k+1} (line 4) and flagged as redundant by setting REDF to **TRUE** (line 5).

```
UPDATE-FKh ( k ): // e_k={v_f,v_t} and e_{k+1}={v_a,v_b}
1)   redf = false; redt = false;
2)   Create F_{k+1} as a copy of F_k;
3)   if ( f ≠ a ) then              // Edges are undirected
4)        Remove v_f from F_{k+1};
5)        redf = true;
6)   if ( k+1 < |E| ) // Don't delete last vertex
7)        for each ( x : k < x < |E| ) do // e_x = {v_y,v_z}
8)             if ( y ≥ t ) then
9)                  break;
10)       if ( ( y > t ) or ( x == |E| ) ) then
11)            Remove v_t from F_{k+1};
12)            redt = true;
13)       else if ( v_t ∉ F_{k+1} ) then
14)            Add v_t to F_{k+1};
15)  return;
```

Figure 4.3: UPDATE-FKh for Model 2.1e

Checking for the redundancy of v_t is more involved. If this is the last vertex, we avoid deleting it in order to simplify termination testing. If not, the only sure way to do this is to traverse all edges until either v_t is found or until the ordering makes

[15] The implementation in this thesis uses linked lists to represent boundary sets, hence the copy operation reduces to a pointer assignment.

it clear that v_t isn't present (lines 7-9). In the latter case, v_t is redundant (line 10), and thus removed from F_{k+1} (line 11) and flagged by setting REDT to **TRUE** (line 12). Finally, if v_t is not redundant then add it to F_{k+1} if it isn't there already.

4.3.2. Child Creation

To create the children for an OBDD-H node, the positive child has information on the available vertices and/or edges added and both children have redundant information removed. Note that even the negative child of the OBDD-H may have information added; unconnected vertices are also included in the partition.

CREATE-NEG-CHILDh (N_i, k):
1) Relabel N_i as N_j;
2) EDGE-DELETEh(N_j, e_k);
3) $P_j = P_j \times (1\text{-}Pr(e_k))$;
4) **return** N_i;

Figure 4.4: CREATE-NEG-CHILDh for Model 2.1e

CREATE-POS-CHILDh (N_i, k):
1) Create N_j as a copy of N_i;
2) EDGE-CONTRACTh(N_j, e_k);
3) $P_j = P_j \times Pr(e_k)$;
4) **return** N_i;

Figure 4.5: CREATE-POS-CHILDh for Model 2.1e

The CREATE-POS-CHILDH and CREATE-NEG-CHILDH methods are shown in Figure 4.5 and Figure 4.4 respectively. It can be seen that both are almost identical; both now call a method that modifies the child node based on the edge being either contracted or deleted, respectively. The calls to EDGE-CONTRACTH (Figure 4.6) and EDGE-DELETEH (Figure 4.7) mirror the contraction and deletion of edges discussed in [15]. Both methods adjust the partition information to reflect the edge being available or failed.

The EDGE-CONTRACTH method first locates the block containing v_t in F_k. Note that, due to the ordering of blocks and vertices in the blocks, the first block b_0 always contains the lowest vertex. Due to the ordering of the undirected edges, the lowest vertex is v_f. Hence it is always true that v_f is the first vertex of b_0.

If v_t is not found in any block of F_k it is added to b_0, the block containing v_f (lines 2-5). Because edges are undirected v_f is guaranteed to already be in a block unless the network being considered is disconnected even when all edges are available, and the reliability of such a network is always zero. If vertex v_t is a target vertex then b_0 becomes marked (lines 4-5).[16]

[16] If b_0 is already marked, setting it to be marked has no effect.

```
EDGE-CONTRACTh (N_i, k):                    // e_k = {v_f, v_t}
1)  Locate block b_y containing v_t in Part_i;   // Set y=B if no partition contains v_t
2)  if ( y == B ) then                      // No partition contains v_t
3)      Add v_t to b_0;                     // b_0 always contains v_f
4)      if ( v_t ∈ T )
5)          Mark b_0;
6)  else if ( y > 0 ) then                  // v_f and v_t in different partitions – merge
7)      for each v_a∈b_y do
8)          Delete v_a from b_y;
9)          Add v_a to b_0;                 // b_0 always contains v_f
10)     if ( b_y is marked )
11)         Mark b_0;
12)     Delete b_y;
13) return N_j;                             // Do nothing if in same partition
```

Figure 4.6: EDGE-CONTRACTh for Model 2.1e

Alternatively, if v_t is found in a block, EDGE-CONTRACTH merges the blocks containing the endpoints of e_k (lines 6-11). All vertices in one block are copied to the other, and if one block was marked the merged block is also marked. Note that if both endpoints are already in the same block, nothing needs to be done.

```
EDGE-DELETEh (N_i, k):                      // e_k = {v_f, v_t}
1)  Locate block b_y containing v_t in VI_i;     // Set y=B if no block contains v_t
2)  if ( y == B ) then                      // No block contains v_t
3)      Add new block b_B;
4)      Add v_t to b_B;
5)      if (v_t ∈ T) then
6)          Mark b_B;
7)  return N_j;
```

Figure 4.7: EDGE-DELETEh for Model 2.1e

The EDGE-DELETEH method simulates edge failure, hence if both endpoints already exist nothing is done, but if one doesn't then it is added as a new block. As with EDGE-CONTRACTH, it is known that v_f is already present in b_0 and hence only the location of v_t is sought. Note that if the new vertex is a target, the new block is marked.

Given that the edge contraction and deletion methods mirror the graph manipulation for BS [15], the OBDD-H algorithm creates child nodes which are equivalent to the child nodes created by BS.

4.3.3. Terminal Node Detection

OBDD-H nodes are divided into success, failure and non-terminal nodes. With the OBDD-H for Models 1e and 2e, the type of node N_i depends entirely on $Part_i$. Testing for node type is shown in the NON-TERMINAL-NODEH method in Figure 4.8.

```
NON-TERMINAL-NODEh (Ni):
1)  if (all non-redundant targets are in the single marked block) then
2)      reliability += Pi *Prob(T);
3)      return false;           //a success node
4)  else if ( Parti == { } ) then
5)      return false;           // a failure node
6)  else
7)      return true;            //a non-terminal node
```

Figure 4.8: NON-TERMINAL-NODEh for Model 2.1e

To determine the success of a node, NON-TERMINAL-NODEH looks in the marked block(s) of $Part_i$. Essentially the computation must have reached all of the target vertices (*i.e.*, each target vertex has been in F_{k+1} for some level) and that all of the target vertices are connected. Whenever a block contains a target vertex it becomes marked, so if there is only one marked block that contains any target vertices that are not yet redundant then all target vertices are connected. Note that if a target is not redundant and is not in F_{k+1} then it cannot be in the marked block, and hence the node cannot be successful.

The failure condition for the OBDD-H is that a marked block is empty[17]. Because empty blocks are removed from the partition stored in each node, NON-TERMINAL-NODEH cannot detect this. Hence if a block becomes empty during the creation of a child node N_i, the entire partition $Part_i$ is set to be empty. Hence NON-TERMINAL-NODEH checks for the partition as a whole being empty.

```
DEL-REDUNDANTh ( Ni, vx ):
1)  for each (bj∈Parti ) do
2)      if ( vx∈bj ) then
3)          Delete vx from bj;
4)          if ( bj is empty ) then
5)              if ( bj is marked ) then
6)                  Parti = { };
7)              else
8)                  Delete bj from Parti;
9)          break;    // Only present once for e
10) return Ni;
```

Figure 4.9: DEL-REDUNDANTh for Model 2.1e

The deletion of $Part_i$ can be seen in the DEL-REDUNDANTH method shown in Figure 4.9; when a marked block becomes empty the node is failed. Note that if blocks are ordered in the implementation, then the deletion of a vertex from a block means re-sorting the partition.[18]

[17] For the special case of ALL-REL all blocks are considered marked and hence any block becoming empty indicates failure.

[18] Since only one block is changed, insertion sort is a good choice for this; the first block is removed from its position and inserted into the appropriate place in the block structure.

Note that if v_f is being deleted, it will be found in the first block. However on the rare occasions that v_t is deleted, this may be in a different block. Hence the method keeps looking through blocks until the redundant vertex is found.

4.4. Example

Consider once again the network shown in Figure 4.10, with s=v_0 and t=v_3, each edge having a probability of 0.9 of being active, and all vertices perfect. The resulting OBDD-H is shown in Figure 4.11 with the level, edge being decided and BS information on the left-hand side. Note that the OBDD-H shown in Figure 4.11 is identical to the OBDD-A for the same problem, shown in Figure 3.16.

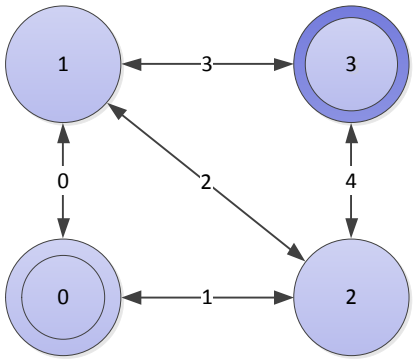


Figure 4.10: Sample Undirected Network

The BS for the last level of the diagram is F_5={v_3} since, as discussed in Section 4.3.1, the last vertex is not deleted from the F_5. Using the definition of the BS given in Eq. (4.1) F_5 would be empty, meaning v_3 would be redundant and deleted from both nodes on the last level. If this were the case, it would be necessary for the algorithm to recognize that an empty marked block does not always indicate failure, which requires additional processing or an additional flag stored in each diagram node.

The root node is N_0=[({[v_0]*},1.0)]; the same information as previously but now using the partition representation instead of VS_i/CI_i. The other constants are unchanged, except that the active vertices are formally listed as F_0 = {v_0} and F_1={v_0,v_1}. The variables REDF and REDT are initialized as **FALSE** when the two BS are created; in this case both are set to **FALSE** since neither endpoint is being deleted.

Once initialization completes, the root node is removed from Q_C and processed. The positive child is created as a copy of N_0 and EDGE-CONTRACTH is called. The edge being decided is e_0={v_0,v_1}, and no block currently contains v_t = v_1 (Y =

B). This means that the test in line 2 is **TRUE** and hence v_1 is added to the first block (b_0). The new vertex, v_1, is not a target vertex and hence line 5 is not executed. Even if it was, b_0 is already marked so nothing would change. Execution returns to line 3 of CREATE-POS-CHILDH and the node probability is multiplied by $Pr(e_0) = 0.9$, making it $1 \times 0.9 = 0.9$. As a result, N_2=[({[v_0,v_1]*},0.9)].

The creation of the negative child through CREATE-NEG-CHILDH follows a similar process except that EDGE-DELETEH adds v_1 in a new block instead of into b_0. This gives N_1=[({[v_0]*[v_1] },0.1)].

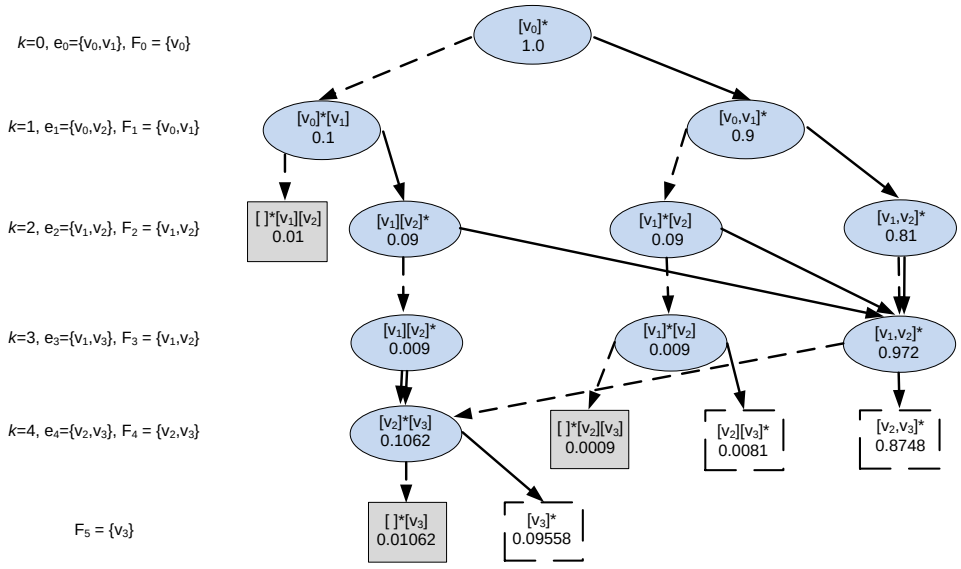


Figure 4.11: OBDD-He for Sample Network

When the nodes are added to Q_N, N_1 is added without checking for isomorphism because Q_N is empty, however N_2 is first compared to N_1. The two nodes are not isomorphic since the partitions are not equal.

The level check in lines 18-19 finds that Q_C is empty but Q_N is not, and thus UPDATE-LEVELH is called. This increments *k* from 0 to 1, swaps the queues, and then calls UPDATE-FKH in turn. This considers both the edge about to be decided, e_1={v_0,v_2}, and the next edge in the ordering, e_2={v_1,v_3}. Since v_0 is not the endpoint of e_2 it is removed from F_2 and marked as redundant by setting REDF to **TRUE**. Since v_2 is also an endpoint of e_4 it is added to F_2 (lines 13-14), making F_2={v_1,v_2}. The loop returns to the top, and N_1=[({[v_0]*[v_1] },0.1)] is removed from Q_C and processed. This process continues until no nodes remain in either queue.

The negative child of N_1=[({[v_0]*[v_1] },0.1)] is created using CREATE-NEG-CHILDH, to give N_3=[({[v_0]*[v_1] [v_2] },0.1)]. In this case, REDF is **TRUE** and hence DEL-REDUNDANTH is called (lines 8-9 of the main algorithm in Figure 4.1) to give N_3=[({[ ]*[v_1] [v_2] },0.1)], which detects as a terminal (failure) node and is discarded.

In contrast, consider N_{11}=[({ [v_1, v_2]* },0.009)], whose positive child is created using CREATE-POS-CHILDH, to give N_{24}=[({[v_1, v_2, v_3]* },0.0081)]. In this case REDF is **TRUE** again and DEL-REDUNDANTH results in N_{24}=[({[v_2, v_3]* },0.0081)]. It can be seen that all targets (v_3 only in this case) are in the marked block, and this is the only marked block in the partition. Hence line 1 of NON-TERMINAL-NODEH is **TRUE**, and the node is a success node.

The algorithm ends with reliability = 0.97848 just as with the equivalent OBDD-A. Note that there is a direct correlation between the nodes of this OBDD-H and its comparable OBDD-A shown in Figure 3.16. For Model 1e the nodes of both diagrams are always identical, but this isn't necessarily the case for other models. This correlation between the OBDD-A and OBDD-H is discussed in Section 4.7.

4.5. The General OBDD-H

4.5.1. Introduction

The OBDD-H is based on BS [15, 18], which was proposed only for Models 1e and 2e. It is possible to modify the OBDD-He algorithm presented in Section 4.3 to compute the same metrics for Models 'v' and 've'. These modifications are relatively straightforward, except for the definition of the F_k and the partitioning thereof. This section presents the general OBDD-H (OBDD-H) that is usable for all three component failure Models (*i.e.*, 'e', 'v', and 've'). This thesis refers to the OBDD-H that is specifically used for Model 'e' as OBDD-He, for Model 'v' as OBDD-Hv, and for Model 've' as OBDD-Hve.

The BS and partition model discussed for the OBDD-He is not applicable for Models 'v' and 've' because it assumes that vertices are perfect. For example, consider the undirected version of the network shown in Figure 4.12 for Model 'v' with a network state where v_0, v_3, v_4 and v_5 are active, v_1 and v_2 are inactive, and v_6 is next to be decided. At this stage of the computation, F_k={v_6,v_7}. What is the correct partition of F_k to describe the network state?

Since v_4 and v_5 are available there exists a path connecting v_6 and v_7; hence both should be in the same block. Since v_0 and v_3 are available v_6 is connected to the source, and hence any block that includes v_6 should be marked. Thus, according to the BS definition introduced by Hardy, Lucet and Limnios [15, 18] the partition should be {[v_6,v_7]*}.

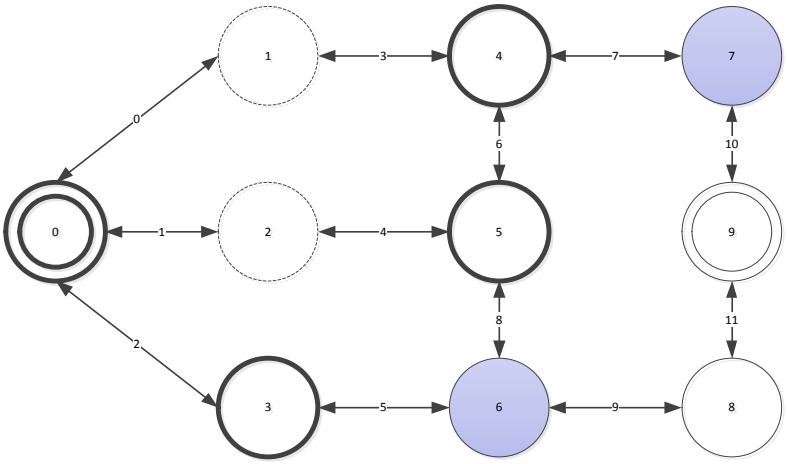


Figure 4.12: Undirected Simple Network

The issue with this is that this partitioning indicates that a reaching path exists to v_7, since v_7 is in the marked block. However there is actually no reaching path to v_7 at this stage, since such a minpath would pass through v_6. If v_6 is failed, that reaching path does not exist and no other reaching path is possible.

This situation occurs because the connection between v_6 and v_7 ([v_6,v_7]) is automatically merged with the connection between v_6 and the source ([v_6]*). This merge occurs because in a partition of F_k, each vertex can only appear once. The solution is to alter the definition of $Part_i$ by removing this requirement, allowing each vertex to appear in multiple blocks. Formally, this is no longer a partition, but this work will continue to use this term for the ease of understanding. In this case, $Part_i$ becomes {[v_6]*[v_6,v_7]}, which correctly encapsulates the network state.

Section 4.5.2 describes the modified mathematical model for the general OBDD-H and Section 4.5.3 provides the relevant pseudo-code. These changes allow OBDD-H to compute REL for Model ‘e’, ‘v’ and ‘ve’ under connectivity Model 1 and 2. Section 4.5.4 discusses the modifications required to extend the OBDD-H to connectivity Model 3. The main changes relate to the updated BS definition discussed in Section 4.5.2.1, which in turn affects the partitions. In particular, partitions for the OBDD-Hv and OBDD-Hve allow vertices to appear multiple

times. This affects the method used to join partitions together when a vertex is found to be available, as discussed in Section 4.5.3.2.

4.5.2. The General OBDD-H Mathematical Model

4.5.2.1 The General Boundary Set

Definition 4.1 assumes that edges are fallible and vertices are perfect. When vertices can fail, the given definition of F_k is not sufficient. If edges are perfect, definition 4.3 is applicable; each vertex v_x in F_k must be adjacent to a vertex that has been decided on a lower level and is removed on level $k=x+1$ since exactly one vertex is decided per level.

$$F_k = \{v_x | \exists \text{ edge } e_y = \{v_a, v_x\} \text{ with } a<x \text{ and } x \geq k\} \quad (4.3)$$

The definition for Model 've' is somewhat more complex since vertices are removed only when all adjacent edges have been decided. Vertices are added to F_k when an adjacent edge is decided, and removed when the last adjacent edge has been decided. Thus the definition shown in (4.4) is identical to (4.1) but uses *ke* instead of *k*.

$$F_k = \{v_x | \exists \text{ edges } e_y = \{v_a, v_x\} \text{ and } e_z = \{v_x, v_b\} \text{ with } a \leq ke,\ b > ke\} \quad (4.4)$$

As with definition (4.1), the definitions above do not function for F_0 so define $F_0=\{v_0\}$.

4.5.2.2 General Partitions

The partitioning of F_k into blocks must be redefined for network models that allow vertex failure. The hybrid algorithm thus requires a method which merges all blocks containing a vertex decided as active, instead of blocks being merged immediately on creation.

The ordering of blocks presented in Section 4.2.2 requires that blocks be sorted in increasing order of their lowest vertex. Two blocks that have the same vertex are ordered with respect to their next-lowest vertex, as with $[v_3]*[v_3, v_4][v_3, v_5][v_4]$. Two blocks that have identical vertices can safely be merged (practically speaking, one is deleted) since this does not cause a loss of information; if one of these blocks was marked the resulting block is also marked.

For the general hybrid algorithm the definition of $Part_i$ and the blocks contained in it removes the second half of Definition 4.2; each block still only contains vertices from F_k but these vertices are no longer restricted to appear in only one block.

$$\forall x.\, v_x^i \in F_k, \bigcup_x v_x^i = \mathrm{F}_k \tag{4.5}$$

These changes affect the way in which child nodes are created, but do not greatly affect either termination testing or isomorphism. However it is now possible to have one vertex in several different blocks until such time as the vertex is decided. If the vertex is failed, it is simply deleted from each block, but if the vertex if active it connects each of the blocks, requiring those blocks to be merged.

4.5.3. The General OBDD-H Algorithm

As with the OBDD-He, isomorphism for the general OBDD-H is based entirely on comparing the partitions of F_k. Even with vertices allowed to be present multiple times in the partition, the ordering of the blocks and the merging of identical blocks means that efficient comparison is still possible.

```
OBDD-H (G):        // General hybrid algorithm
1)  Initialize the root node N0;
2)  Initialize redf, redt, F0, F1, reliability = 0;
3)  Initialize level variables;
4)  Remove the first node, Ni, from QC;
5)  for each edge ex being decided do
6)        Npos = CREATE-POS-CHILDh(Ni, ex, k);
7)  Nneg = CREATE-NEG-CHILDh(Ni, k);
8)  for each child node Nfi do
9)        if redf then
10)             DEL-REDUNDANTh( Nfi, vf );
11)       if redt then
12)             DEL-REDUNDANTh( Nfi, vt );
13)       if ( NON-TERMINAL-NODEh(Nfi) ) then
14)             Check each node on QN for isomorphism with Nfi;
15)             if ( an isomorphic node Nx was found ) then
16)                   Px = Px + Pfi;                  // Merge nodes
17)             else
18)                   Add Nfi onto QN;
19) if ( QC == { } ) then
20)       if ( QN == { } ) then
21)             return reliability;
22)       else
23)             UPDATE-LEVELh( k );  // For 've' use ke instead of k
24) goto 4)
```

Figure 4.13: General OBDD-H Algorithm

Node success and failure is identical for the general OBDD-H as for the OBDD-He except that the computation that sufficient target vertices are active becomes non-trivial when vertices are fallible. The NON-TERMINAL-NODEH and DEL-REDUNDANTH methods remain the same. The methods for the hybrid algorithm

use the 'H' suffix. For example the general NON-TERMINAL-NODEH method is the OBDD-H equivalent of NON-TERMINAL-NODE.

The general OBDD-H algorithm is shown in Figure 4.13 and is entirely identical to the general OBDD-A algorithm except for the initialization of F_0 and F_1. The OBDD-A doesn't explicitly use the BS and hence does not require this initialization. For the general OBDD-H F_0 = S, the set of all source vertices. F_1 is computed as in UPDATE-LEVELH, based on e_0. As with the restricted Hybrid algorithm for Models 1e and 2e, i.e., OBDD-He, the differences between the OBDD-H and OBDD-A are entirely hidden within the methods called.

4.5.3.1 Computing the Boundary Set

As with other general methods, the general UPDATE-LEVELH method for the Hybrid algorithm shown in Figure 4.14 is similar to that shown in Section 4.3. The difference lies in Model 've' requiring two additional level variables, *kv* and *ke*, which determine the next vertex and edge to be decided, respectively. Updating the level variables increments *k*, and for Model 've' it increments *ke* when $ke+kv < k$ and *kv* when REDF is true.

For Models 'e' and 'v', the level variable *k* is passed in as the argument to UPDATE-LEVELH, while for 've' the variable *ke* is passed in, instead.

```
UPDATE-LEVELh ( k ):
1)   Swap Q_C and Q_N;
2)   UPDATE-FKh( k );
3)   Update level variables;
4)   return;
```

Figure 4.14: General UPDATE-LEVELh

```
     UPDATE-FKh ( k ): // e_k={v_f,v_t} and e_{k+1}={v_a,v_b}, for Model 'v', f = k
 1)  redf = false; redt = false;
 2)  Create F_{k+1} as a copy of F_k;
e,ve 3)  if ( f ≠ a ) then                    // Edges are undirected
 4)      if ( v_f is not a target )
 5)          Remove v_f from F_{k+1};
 6)          redf = true;
e,ve 7)      for each ( x : k < x < |E| ) do      // e_x = {v_y,v_z}
e,ve 8)          if ( y ≥ t ) then
e,ve 9)              break;
e,ve 10)     if ( ( y > t ) or ( x == |E| ) ) then
e,ve 11)         Remove v_t from F_{k+1};
e,ve 12)         redt = true;
e,ve 13)     else
 14)         if ( v_t ∉ F_{k+1} ) then
 15)             Add v_t to F_{k+1};
 16) return;
```

Figure 4.15: General UPDATE-FKh

The general UPDATE-FKH method is similar to the restricted one shown in Section 4.3, but must take account of the various different methods for tracking which vertex is being decided. Models ‘e’ and ‘ve’ are handled the same way but Model ‘v’ is simpler; every level of Model ‘v’ decides one vertex and makes it redundant.

Note that some lines (*i.e.*, 3, 7-13) apply only to certain models as shown in the left-hand column; such lines are skipped when computing other models. For example line 3 is only executed by the OBDD-He and OBDD-Hve, but not the OBDD-Hv.

4.5.3.2 Joining Blocks for Active Vertices

In order to merge the various blocks containing a single vertex, the JOINH method (shown in Figure 4.16) is called when a vertex is decided as active. JOINH locates each block containing the vertex and merges them into one block.

Note that JOINH is always called on a vertex decided as active, and the vertex being decided, v_k, is always the lowest vertex. Hence the ordering of blocks assures that b_0 contains v_k. Furthermore, any block containing v_k has this vertex as the first vertex of the block. Hence the implementation of JOINH must only check the first vertex of each block of $Part_i$. Specifically, each block with first vertex v_k is merged with b_0 until the loop in Line 1 finds a block whose first vertex is not v_k; when this occurs the block ordering guarantees that each remaining block will not contain v_k.

```
JOINh (N_i, v_k):
1)   for each partition b_x other than b_0 containing v_k do
2)        b_0 = b_0 ∪ b_x;
3)        if ( b_x is marked ) then
4)             Mark b_0;
5)             Delete partition b_x;
6)        else
7)             break;
8)   return N_i;
```

Figure 4.16: General JOINh

Note that it is possible to implement the general OBDD-H for Models ‘e’ using the methods above; if doing so JOINH is called on both endpoints of every edge being decided. Doing so removes the assumption that the method is operating on v_k, which makes it far less efficient; hence the pseudo-code joins blocks inline for Model ‘e’.

For Model ‘ve’, the method given here is only used when deciding the last edges adjacent to a particular vertex, and only when that vertex was decided as active. It

would be possible to immediately call JOINH after each edge is decided, but in this case it is more efficient to only call JOINH when all adjacent edges have been decided.

4.5.3.3 General OBDD-H Child Creation

The general OBDD-H child creation methods, CREATE-POS-CHILDH, and CREATE-NEG-CHILDH, call the Hybrid functions EDGE-CONTRACTH and EDGE-DELETEH. The CREATE-POS-CHILDH method is shown in Figure 4.17.

```
     CREATE-POS-CHILDh(N_i, g, k):          // g is list of active edges for this child
     1)  Create N_j as a copy of N_i;
     2)  P_change = 1.0;
     3)  if ( deciding a vertex this level )
     4)      P_change = P_change × Pr(v_k);
ve   5)      s_j = true;                    // Remember that the vertex is active
     6)  for each (e_x = {v_k, v_t} in g ) and ( t > k ) do
     7)      EDGE-CONTRACTh(N_j, e_x);      // General Version
     8)      if ( deciding an edge this level )
     9)          P_change = P_change × Pr(e_x);
v,ve 10) JOINh(N_i, v_k);
     11) P_j = P_j × P_change;
     12) return N_j;
```

Figure 4.17: General CREATE-POS-CHILDh

The CREATE-NEG-CHILDH method, shown in Figure 4.18, calls only EDGE-DELETEH. Note that when a vertex is found to have failed, EDGE-DELETEH is called for every undecided edge adjacent to this vertex. The main purpose of this is to add singleton blocks for any vertices that have been added to F_{k+1} but are not yet in $Part_i$.

```
     CREATE-NEG-CHILDh(N_i, k):
     1)  Relabel N_i as N_j;                // N_i no longer needed
e    2)  P_j = P_ij× (1 - Pr(e_k));
v    3)  P_j = P_j × (1 - Pr(v_k));
v    4)  for each ( undecided edge e adjacent to v_k )
v    5)      EDGE-DELETEh(e);
ve   6)  if kv + ke == k then
ve   7)      P_j = P_j × (1 - Pr(v_kv));
ve   8)      s_j = false;
ve   9)  else      // Deciding v_ke with s_i==true
ve   10)     P_j = P_j × (1 - Pr(e_ke));
     11) return N_i;
```

Figure 4.18: General CREATE-NEG-CHILDh

It can be seen that the method varies markedly for the different network models; line 2 is only executed by the OBDD-He, lines 3-5 only by the OBDD-Hv and lines 6-10 are only applicable to the OBDD-Hve.

```
     EDGE-CONTRACTh (N_i, k):                    // e_k = {v_f, v_t}
e    1)  Locate block b_y containing v_t in VI_i; // Set y=B if no block contains v_t
e    2)  if ( y == B ) then                      // No block contains v_t
e    3)      Add v_t to b_x;
e    4)  else if ( y > 0 ) then                  // v_f and v_t in different blocks – merge
e    5)      for each v_a∈b_y do
e    6)          Delete v_a from b_y;
e    7)          Add v_a to b_0;
e    8)      if ( b_y is marked ) then
e    9)          Mark b_0;
e    10)     Delete b_y;
v,ve 11) Create block b=[v_f, v_t];
v,ve 12) if (v_f or v_t is a target vertex ) then
v,ve 13)     Mark b;
v,ve 14) Add b to Part_i;
     15) return N_j;
```

Figure 4.19: General EDGE-CONTRACTh

This modification to the EDGE-CONTRACTH method is shown in Figure 4.19; if vertices are fallible a contracting edge simply adds a block with that edge's endpoints to the node.

The EDGE-DELETEH method shown in Figure 4.20 needs only a minor alteration; since a vertex can be in multiple blocks it no longer makes sense to attempt to locate a single block. Instead the method checks for v_t being present in any block, and if it is found the method exits.

```
EDGE-DELETEh (N_i, k):
1)  if ( no partition contains v_t ) then
2)      Add new block b_j =[v_t];
3)      if ( v_t is a target ) then
4)          Mark b_j;
5)  return N_j;
```

Figure 4.20: General EDGE-DELETEh

4.5.4. Extension to Model 3

The OBDD-H described in Section 4.5.3 can solve all component failure Models ('e', 'v', and 've') for Models 1 and 2, however it cannot solve Model 3. The BS model used by OBDD-H doesn't track the number of targets connected to the source; BS assumes that every target has to connect to each other and hence counting is irrelevant.

While Models 1 and 2 do not truly require a differentiation between source and targets, Model 3 does. The difference exists because while the source must be connected to a certain number of target vertices, it is not sufficient for a number of target vertices to be connected to each other. For this reason it isn't possible to mark blocks to indicate a connection to the source or a target vertex. The OBDD-H

for Model 3 only marks the initial block containing the source vertex; blocks created containing target vertices aren't marked.

If the marked block becomes empty, no more targets can be connected to the source vertex. If such a node isn't already successful, it is failed. Hence the failure test remains unchanged for Model 3. The success test must change, however.

For Model 3.1, reaching any target is sufficient. In other words, if a target vertex enters the marked block and is decided as available, a minpath has been found and the node is successful. For Model 3.2, at least c target vertices must be reached, while for Model 3.3 c_i targets need to be reached for each group g_i.

In each case a target vertex can become redundant without the node failing; it is always possible that another target will become connected to the source. The OBDD-H for Model 3 must count the number of targets that are successfully connected to the source in some way. This requires information to be added to each node; either counting the number of targets reached from each target group or keeping track of every target reached. This is analogous to what is done for the OBDD-A, as described in Appendix A. Doing this, the success tests in NON-TERMINAL-NODEH now check that sufficient targets have been reached in each group.

The only changes are that new blocks containing target vertices in EDGE-CONTRACTH and EDGE-DELETEH are not marked and that a number of optimizations can be made because there is only a single marked block. Many of these optimizations are not used in the implementation for this thesis in order to keep comparisons between models more accurate.

For completeness, the required modifications to the pseudo-code of the OBDD-H for Model 3 are given in Appendix D.

4.6. Example

Consider the undirected sample network as shown in Figure 4.21. Let $s=v_0$ and $t=v_3$ with each vertex having a probability of 0.9 of being active, and all edges being perfect. This is similar to the example in Section 4.4 but uses Model 1v instead of Model 1e.

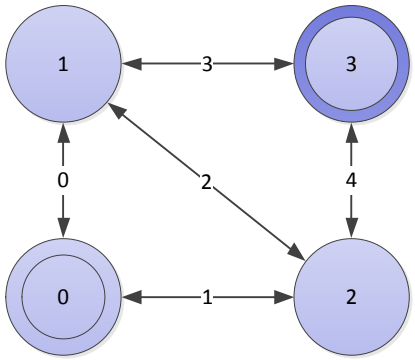


Figure 4.21: Undirected Sample Network

The aim of this example is to demonstrate the difference caused by the change in the definition of $Part_i$. The full diagram is shown in Figure 4.22. Consider the root node, N_0=[({[v_0]*}, 1.0)] being processed to give the positive child. The CREATE-POS-CHILDH method is called once for every undecided edge adjacent to the decision vertex (lines 6-7 in Figure 4.13); in this case e_0 and e_1. The method EDGE-CONTRACTH functions in a similar manner to that for Model 1e, except that for Model 'v' it doesn't need to locate v_t; a new block [v_f,v_t] is added instead (lines 11-14). This gives N_2=[({[v_0]*[v_0,v_1][v_0,v_2]}, 0.9)] since neither v_1 nor v_2 are target vertices.

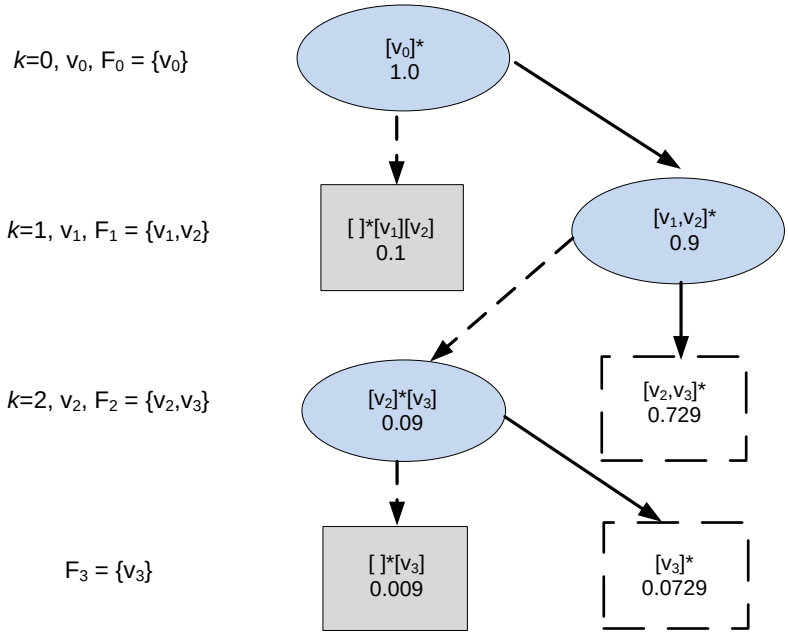


Figure 4.22: OBDD-Hv for Network in Fig. 4-22

The decision vertex, v_0, currently exists in three blocks of the partition. Because the positive child is being created v_0 is active, and hence will connect these blocks. This is the purpose of calling JOINH (line 10 in Figure 4.17). At this point it is

certain that the first block, b_0 contains the decision vertex v_k, and hence all other blocks containing v_k are joined with b_0 using set union. This gives N_2=[({ $[v_0,v_1,v_2]$*}, 0.9)] as expected.

The JOINH method and its associated change in the partitioning are the main difference between the algorithm for Model 'e' and Models 'v' and 've'. The difference between network connectivity Models 1 and 2 and Model 3 is discussed in detail in Section 4.5.4.

4.7. Comparing the OBDD-H and OBDD-A Algorithms

While the computation of REL for Models 1e and 2e is possible using the OBDD-Ae as presented in Chapter 3, the notation used is more complex (and hence less efficient) than that of BS, which was presented by Hardy *et al*. [15, 18]. While the OBDD-Ae model uses conditions in order to track potential paths whose endpoints have not yet been reached, BS only tracks the connections between the vertices of F_k. The overhead of checking for condition pairs makes OBDD-A slower than BS. However BS requires more memory since all nodes must be kept in memory.

The Hybrid OBDD-A (OBDD-H) was introduced [102, 106] to combine the benefits of both OBDD-A and BS. This OBDD-H replaces the VS_i/CI_i notation discussed in Section 3.2.3 with a partition of the F_k. While the VS_i notation is efficient, CI_i can potentially contain a large number of conditions and the partition notation used in BS and OBDD-H is far more efficient. However CI_i allows for a condition (v_a,v_b) to be part of a node without its reverse, (v_b,v_a) being present and hence can be used for directed networks. Partitions only record which vertices are connected to each other, and assume that the connection occurs in both directions; hence the restriction to undirected networks.

The OBDD-H model is more elegant than the OBDD-A model because conditional paths do not need to be tracked separately. All connectivity is modelled through the partitioning of the F_k. However the requirement that all edges must be undirected may not be suitable for some networks, and components (used for performability problems) become more complicated.

In addition, the BS notation allows multiple marked blocks for Model 2.1e, referred to as K-REL. The BS notation allows K-REL to be addressed in terms of connecting the vertices in K; OBDD-A requires choosing one of these vertices to be a source and the rest to be targets. Further, this method makes checking for disconnected members of K elegant and efficient, in a similar manner to such

checks for Model 2.2. Moreover, OBDD-H does not have to track redundant target vertices (or avoid targets becoming redundant) for Models 1 and 2.

For Model 1e, there is a direct correlation between OBDD-A node information and OBDD-H node information. The marked partition of OBDD-H node N_i contains exactly those vertices that are in VS_i for the OBDD-A, and the other partitions of the OBDD-H containing more than one vertex are equivalent to OBDD-A conditions. For example an OBDD-H node that contains the partitioning $[v_0, v_1][v_2, v_3][v_4]$ is equivalent to the OBDD-A node with $VS_i = \{v_0, v_1\}$ and $CI_i = \{ (v_2,v_3), (v_3,v_2) \}$. Note that v_4 does not appear in the OBDD-A node at all, and that the OBDD-H node assumes that connections are undirected, whereas the OBDD-A conditions make this explicit. This correlation does not exist for Model 2.1e since BS notation does not consider specific source/target vertices in these cases.

This indicates that the partition notation in OBDD-H is more efficient than the VS_i/CI_i notation in OBDD-A. For example, the unmarked partition $[v_2,v_3,v_4]$ is equivalent to the conditions $\{ (v_2, v_3), (v_2, v_4), (v_3, v_2), (v_3, v_4), (v_4, v_2), (v_4, v_3) \}$. Similarly, an unmarked partition containing four vertices is represented by 12 conditions; in general an unmarked partition of x vertices is equivalent to $\binom{x}{2}$ conditions. These conditions require more storage space than the partition, and require additional processing to generate and modify.

The main OBDD-He algorithm for Models 1e and 2e (K-REL), shown in Figure 4.1, is identical to the corresponding OBDD-A algorithm for Models 1 and 2 except for the method of starting a new level of the diagram. Most of the changes necessitated by the change of notation from VS/CI to partitions are hidden within the methods being called. The changes are that CHECK-REDUNDANT is subsumed by the computation of F_{k+1}, which in turn is included in UPDATE-LEVELH. Note that while OBDD-A makes use of the concept of a BS, it is never explicitly computed. The more general OBDD-H algorithm described in Section 4.5.3 is similarly virtually identical to the OBDD-A version.

A number of issues are identical between the OBDD-A and OBDD-H. For example the variable ordering used for both is equivalent, although different variable orderings may be optimal for each approach.

The difference from the updating of levels for OBDD-A is the computation of F_k (UPDATEFKH), which replaces the call to CHECK-REDUNDANT. The computation of which vertices to remove from F_{k+1} in UPDATE-FKH is analogous

to checking for redundancy for the OBDD-A. The advantage over the OBDD-A CHECK-REDUNDANT method is that, since all edges are undirected, it is not required to check on the necessity of swapping the endpoints of either edge. If OBDD-A is similarly constrained to undirected edges then the same modification can be made to CHECK-REDUNDANT.

The creation of children for the OBDD-H follows a similar pattern to the OBDD-A; the positive child has information on the available vertices and/or edges added and both children have redundant information removed. Unlike the OBDD-A, the negative child of the OBDD-H has information added; even unconnected vertices are included in the partitioning. The OBDD-A only records connections – in terms of connectivity to the source(s) and conditional connections to other vertices – it doesn't record vertices that aren't connected to any other vertices.

The JOIN$_H$ method of the general OBDD-H is similar to the RESOLVE method for the general OBDD-A, which merges all conditions on a vertex decided as active; see the general OBDD-A in Appendix A. However while TRIGGER/RESOLVE for the OBDD-A are quite complicated methods, the elegance of the BS notation makes JOIN$_H$ similarly elegant. In particular, the OBDD-H method is more efficient since the same vertex is likely to be in a far larger number of conditions than blocks. In addition the vertex for which the merge is taking place is often the first vertex of the first several blocks, although an OBDD-A modified for undirected networks would have the same benefit.

4.8. Performance Evaluation

4.8.1. Performance Comparison with OBDD-A and BS

It has been shown in Section 3.6 that OBDD-A performs at least as well as other solutions for all network models being considered, with BS also being very efficient. This section compares the performance of the OBDD-H to the OBDD-A and BS algorithms. The implementation in this thesis uses the algorithm discussed in Section 4.3 to compute REL for Models 1e and 2e and the algorithm from Section 4.5 for all other models. Comparisons are not made with EED_BFS, EF or other algorithms since these have been shown to be inferior to BS and OBDD-A.

The OBDD-H algorithm has been implemented, with a focus on code-sharing with the OBDD-A algorithm to make processing comparisons as useful as possible. As with the OBDD-A implementation, this has the effect of reducing the amount of code optimization possible.

4.8.1.1 Model 'e'

It can be seen in Table 4.1 that the processing time for the OBDD-He is generally similar to that of the OBDD-Ae due to the small network sizes. Both OBDD-Ae and OBDD-He have relatively similar times for all of these networks, indicating that the basic overheads play a large part for such small networks.

Table 4.1: Performance of BS, OBDD-Ae and OBDD-He for Model 1e

	BS		OBDD-Ae			OBDD-He		
Network	Time	Nodes	Time	Nodes	Max N	Time	Nodes	Max N
2x100	0.03	154	0.03	889	3	0.04	889	3
2x20	0.06	1810	0.03	169	3	0.04	169	3
3x10	0.09	5370	0.03	384	9	0.04	384	9
3x12	0.35	65665	0.03	474	9	0.04	474	9
5x5	0.11	8452	0.04	1694	90	0.05	1694	90
K6	0.06	356	0.04	190	27	0.05	190	27
path18	0.05	440	0.03	284	27	0.04	266	27
path19	0.05	2425	0.06	4912	422	0.06	4912	422
ring 2x6	0.08	152	0.03	264	28	0.04	264	28
ring 2x8	0.02	792	0.03	432	28	0.04	432	28

Note that the number of nodes generated for OBDD-A and OBDD-H are the same, since both used identical input files. BS produces a different number of nodes due to a different network ordering, and possible use of heuristics. Further, note that OBDD-H has the same maximum number of nodes per level as the OBDD-A as shown in the Max N columns.

Table 4.2: Performance of OBDD-Ae and OBDD-He on Large Networks for Model 1e

	OBDD-Ae			OBDD-He		
Network	Time	Nodes	Max N	Time	Nodes	Max N
8x8	3.45	120,847	3432	3.99	120,847	3432
5x15,000	112.14	12,147,644	90	98.69	12,147,644	90
7x1,000	103.50	12,951,730	1001	87.33	12,951,730	1001

While it is not possible to compare the OBDD-He with BS on large networks for Model 1e since no such results for BS are presented, such a comparison can be made with OBDD-A as shown in Table 4.2. It can be seen that the number of nodes generated is still identical, but that the OBDD-He is noticeably faster for the larger networks.

A similar comparison holds for Model 2.2e; OBDD-He is comparable to OBDD-Ae for smaller network as shown in Table 4.3. Note that the number of nodes

generated is different for the OBDD-He and OBDD-Ae; the different notations produce different terminal node detection tests as discussed in Section 4.3.3 for Model 2.2e.

Table 4.3: Performance of BS, OBDD-Ae and OBDD-He for Model 2.2e

	BS		OBDD-Ae			OBDD-He		
Network	**Time**	**Nodes**	**Time**	**Nodes**	**Max N**	**Time**	**Nodes**	**Max N**
K6	0.04	221	0.03	351	61	0.04	214	41
path 18	0.07	160	0.03	288	26	0.04	156	14
path 19	0.12	2062	0.04	1784	154	0.04	2059	132
ring 2x6	0.2	174	0.03	246	25	0.04	171	14
ring 2x8	0.06	258	0.03	357	25	0.04	255	14
3x10	0.08	219	0.03	369	9	0.04	217	5
3x12	0.04	269	0.04	455	9	0.04	267	5
5x5	0.07	822	0.04	1241	69	0.04	820	42
2x20	0.05	116	0.04	170	3	0.04	114	2
2x100	0.09	596	0.06	890	3	0.04	594	2

However the current implementation of OBDD-Ae is unable to compute REL for large networks, in Table 4.4, under Model 2.2e, even with the use of a heuristic. By comparison, OBDD-He can compute even large networks efficiently using standard BS notation as shown in Table 4.4. While these results make it appear that OBDD-He is far more efficient than BS, recall that the processing time comparisons are not necessarily a true performance comparison since it is affected by a range of factors other than the algorithm itself. However the memory performance of OBDD-He is confirmed as being far superior to that of BS; for example OBDD-He stores no more than 84 (42×2) nodes for the 5×15,000 network compared to the 6,509,661 nodes stored for BS.

Table 4.4: Performance of BS and OBDD-He on Large Networks for Model 2.2e

	BS		OBDD-He		
Network	**Time**	**Nodes**	**Time**	**Nodes**	**Max N**
8x8	0.7	179,410	0.43	51,057	1430
5x15,000	65.56	6,509,661	9.50	5,668,930	42
7x1,000	85.18	6,559,609	17.70	5,550,980	429

The results presented indicate that the OBDD-H is comparable or superior to the other methods, including OBDD-A, for Model ‘e’.

4.8.1.2 Model 'v'

As with Model 'e', the OBDD-Hv shows an improvement in processing time when compared to the OBDD-Av. In this case the performance improvement is somewhat less since several advantages of the pure partitioning system used in BS and OBDD-He are lost; in particular the knowledge that the decision vertex is the first vertex in the first block (for sorted partitions) and is not present anywhere else. The JOINH operation is relatively efficient, but still requires a small amount of additional processing. This is shown in Table 4.5.

Table 4.5: Performance of OBDD-Av and OBDD-Hv for Model 1v

	OBDD-Av			OBDD-Hv		
Network	**Time**	**Nodes**	**Max N**	**Time**	**Nodes**	**Max N**
4x1000	0.40	97,720	30	0.40	97,720	30
4x2000	0.79	195,720	30	0.78	195,720	30
4x3000	1.17	293,720	30	1.18	293,720	30
4x4000	1.58	391,720	30	1.56	391,720	30
4x5000	1.96	489,720	30	1.95	489,720	30

Like the OBDD-A, the OBDD-H generates fewer nodes for Model 'v' compared to Model 'e'. However, as shown in Table 4.6, it requires more processing time. The extra processing required per OBDD-Hv node is not sufficiently counter-balanced by the reduced number of nodes generated.

Table 4.6: Performance of OBDD-He and OBDD-Hv for Model 1

	OBDD-He			OBDD-Hv		
Network	**Time**	**Nodes**	**Max N**	**Time**	**Nodes**	**Max N**
4x1000	0.31	195,587	28	0.40	97,720	30
4x2000	0.59	391,587	28	0.78	195,720	30
4x3000	0.87	587,587	28	1.18	293,720	30
4x4000	1.18	783,587	28	1.56	391,720	30
4x5000	1.46	979,587	28	1.95	489,720	30

Note that the number of nodes for the OBDD-Av and OBDD-Hv is not necessarily identical. For the OBDD-Av, conditions become redundant when one of their endpoints have been decided, but also if the *to* endpoint is in VS_i[19]. For the OBDD-Hv, this last optimization cannot be carried out due to the notation used.

For example, consider the computation of REL for the 5x5 grid network under Model 'v'. It can be seen that in the OBDD-Av node [({ v_{18}, v_{19} }, 0.00002542), {

[19] Compare this to the OBDD-Ae where a condition is deleted if either endpoint is in VS_i.

(v_{19},v_{20}) (v_{21},v_{20}) (v_{19},v_{21}) (v_{20},v_{21}) }], v_{19} is connected to both v_{20} and v_{21} but the conditions leading to v_{19} are not present; these are not needed since v_{19} has already been reached. The equivalent node for OBDD-Hv is [{[v_{18}, v_{19}]*[v_{19}, v_{20}, v_{21}] }, 0.00002542].

The OBDD-Av node is processed for edge (v_{18},v_{21}) to give the positive child [({ v_{19}, v_{21} }, 0.00002288), { (v_{19},v_{20}) (v_{21},v_{20}) }]. For the OBDD-Hv the positive child is [{[v_{19}, v_{20}, v_{21}][v_{19}, v_{21}]* }, 0.00002288]. Consider now the OBDD-Hv node on the same level, that has partition {[v_{19}, v_{20}][v_{20}, v_{21}][v_{19}, v_{21}]* }; the equivalent OBDD-Av VI/CI is VI_i={ v_{19}, v_{21} }, CI_i={ (v_{19},v_{20}) (v_{21},v_{20}) }. The OBDD-Hv partition is different from that in the positive child, but the OBDD-Av equivalent is identical to the OBDD-Av positive child. Hence the OBDD-Av child will be found isomorphic to an existing node and merged, while this is not the case for the OBDD-Hv node.

Note that both OBDD-Hv partitions, while not identical, produce the same outcome. Because both v_{19} and v_{20} are in the marked partition, the fact that they are connected is no longer relevant. If either of these two is connected to another node and is available, the other node will be moved into the marked partition.

Table 4.7: Performance of OBDD-Av and OBDD-Hv on Wide Grids for Model 1v

	OBDD-Av		OBDD-Hv	
Network	Nodes	Max N	Nodes	Max N
5×5	540	53	541	53
8×8	40,209	2,303	40,691	2,336
10×10	665,687	30,091	679,188	30,788

The effect of this can be seen in Table 4.7. While the OBDD-Hv generated only one additional node for the 5×5 grid, the number goes up as the width of the grid network increases. It would be possible to add a heuristic to the OBDD-Hv algorithm to transform partitions to match the OBDD-Av optimization, although checking for this particular case would add a processing cost to each node. The implementation in this thesis avoids the use of specific heuristics where possible in order to keep comparisons as fair as possible as discussed in Chapter 6.

Table 4.8: Performance of OBDD-Ave and OBDD-Hve for Model 1ve

	OBDD-Ave			OBDD-Hve		
Network	Time	Nodes	Max N	Time	Nodes	Max N
4x1000	2.26	1,149,095	154	1.84	1,154,077	156

4x2000	4.43	2,301,095	154	3.64	2,311,077	156
4x3000	6.63	3,453,095	154	5.45	3,468,077	156
4x4000	8.84	4,605,095	154	7.29	4,625,077	156
4x5000	11.06	5,757,095	154	9.09	5,782,077	156

4.8.1.3 Model 've'

The results for the OBDD-Hve are in line with that of the OBDD-Hv; the OBDD-Hve has better processing time than the OBDD-Ave but generates slightly more nodes. Indeed, because Model 've' allows more possible combinations for conditions and marked vertices, the number of extra nodes increases slightly. This is shown in Table 4.8.

The OBDD-Hve has less per-node processing than the OBDD-Hv; the former only considers one vertex or edge at a time while the latter considers all remaining edges adjacent to the vertex being decided on a level. However this is offset by the OBDD-Hv having far fewer levels, and thus less nodes in total. Similarly, the OBDD-Hve performs far worse in terms of processing time than the OBDD-He on the same networks, and generates far more nodes.

Table 4.9: Performance of OBDD-H Component Failure Models

	OBDD-He		OBDD-Hv		OBDD-Hve	
Network	Time	Nodes	Time	Nodes	Time	Nodes
4x1000	0.31	195,587	0.40	97,720	1.84	1,154,077
4x2000	0.59	391,587	0.78	195,720	3.64	2,311,077
4x3000	0.87	587,587	1.18	293,720	5.45	3,468,077
4x4000	1.18	783,587	1.56	391,720	7.29	4,625,077
4x5000	1.46	979,587	1.95	489,720	9.09	5,782,077

4.8.2. The Effects of F_{max} on OBDD-H Performance

The OBDD-H shares the OBDD-A property of having constant space performance and linear time performance for families of networks with identical inter-connectivity, as shown in Figure 4.23 and Figure 4.24. This is expected for the OBDD-He since it generates the same number of nodes as the OBDD-Ae. The number of nodes generated for Models 'v' and 've' is not identical, but a constant bound still exists.

As with the OBDD-A, the time performance and space performance are still exponential for general networks. This is expected due to the difficulty of the

problem. The performance for other connectivity models is discussed in Section 4.8.3.

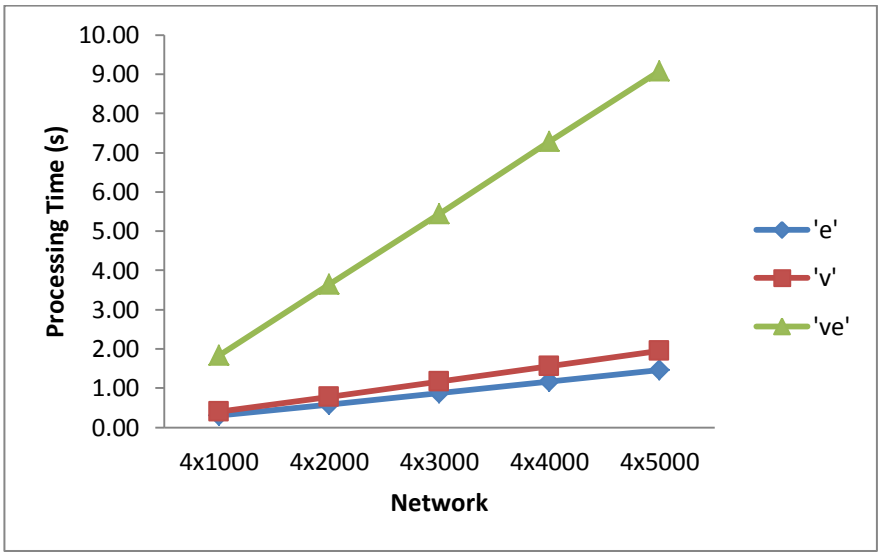


Figure 4.23: Processing Time for 4×*L* Grids - Model 1

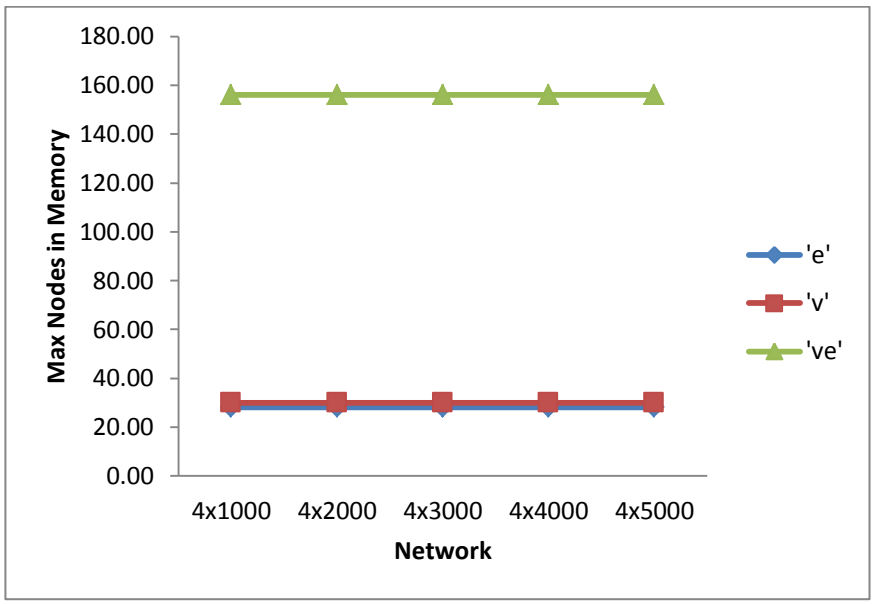


Figure 4.24: Maximum Diagram Nodes in Memory for 4×*L* Grids - Model 1

4.8.3. The Effects of Communication Models on OBDD-H Performance

The OBDD-H, as presented in this chapter, computes REL for Models 1 and 2 only. The performance under Model 1 has been addressed in the previous parts of this section, and is not addressed further. The reliability of different connectivity models has been addressed in Chapter 3, and thus does not need to be addressed

again. However the effect of connectivity requirements on OBDD-H when computing large networks has only been addressed for Model 1 and Model 2.2e.

This section demonstrates the effect of connectivity requirements on OBDD-H when computing Models 2.1, 2.2 and 3. The 5×5 grid is used for Model 2.1 and 3 as in Chapter 3, while the 4×*L* networks are used for Model 2.2. When multiple targets are required for the 5×5 grid, they are chosen as per Figure 3.22 from Chapter 3. For K-REL, the set K is defined as the source plus targets.

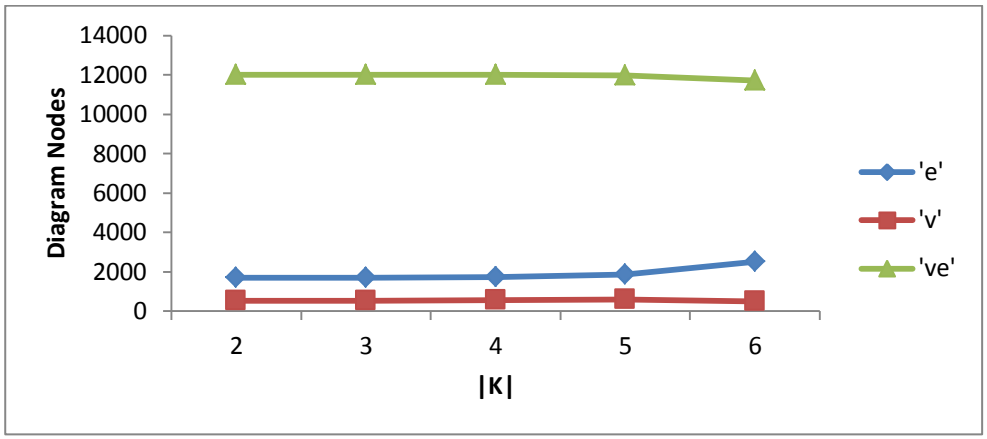


Figure 4.25: Diagram Nodes for 5×5 Grid with Varying |K| - Model 2.1

4.8.3.1 Model 2.1 – K-REL – REL(s,T)

The number of nodes processed for OBDD-A under Model 2.1 was unusual in that different behaviour was evidenced between the component failure models. The same is true for the OBDD-H, with the number of nodes decreasing with |K| for Models ‘v’ and ‘ve’ but increasing for ‘e’.

4.8.3.2 Model 2.2 – ALL-REL – REL(s,V)

Unlike other connectivity models, Model 2.2 does not allow the varying of the size of |T| without changing the number of vertices, |V|, of the network. Hence it is not possible to vary |T| on the 5×5 grid network as has been done for other comparisons both in the rest of this section and in Chapter 3. Hence it remains only to establish that Model 2.2 does indeed follow the same pattern of node increase demonstrated for Model 1.

As the size of |T|, and hence the size of |V|, increases for a family of networks with fixed F_{max}, the number of nodes increases linearly as shown in Figure 4.26. The processing time increases similarly.

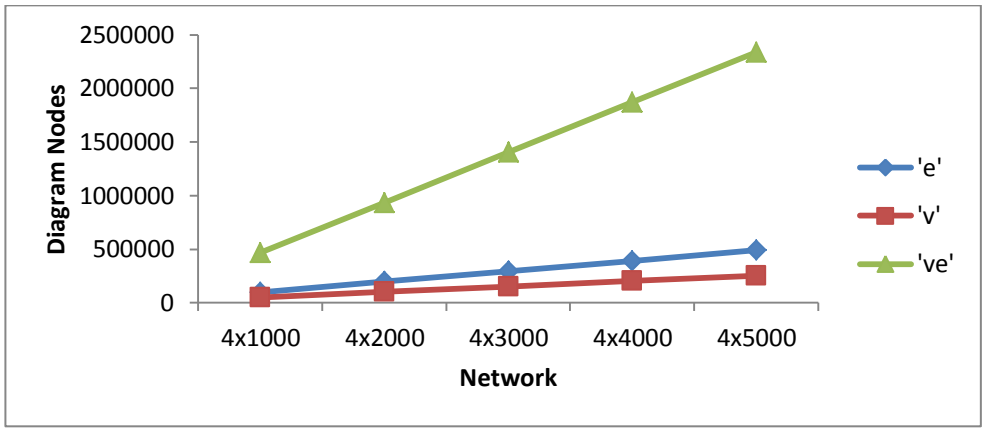


Figure 4.26: Diagram Nodes for 4×*L* Grid - Model 2.2

When F_{max} is not fixed, the increase in both nodes processed and processing time is exponential in |V|.

4.8.3.3 Model 3

The behaviour of the OBDD-H is not as clear under Model 3 as the OBDD-A, largely due to the difference in checking for node success and failure. The changes in the number of nodes generated are generally smaller, and the direction of the change is not identical to that in the OBDD-A.

For Model 3.1 (REL(*s*,1-of-T)), varying the number of target vertices as shown in Figure 4.27 decreases the number of nodes generated for Models 'v' and 've', as is the case with the OBDD-A. However for Model 'e' it can be seen that the number of nodes generated actually increases by a small amount.

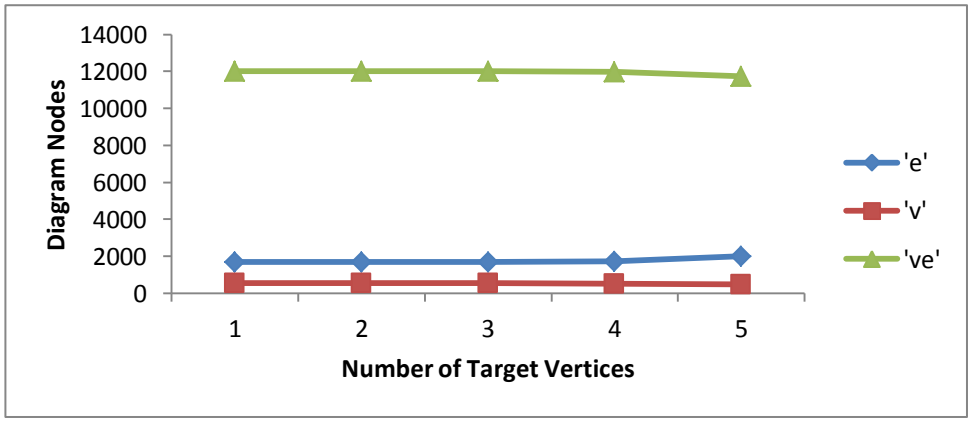


Figure 4.27: Diagram Nodes for 5×5 Grid with Varying |T| - Model 3.1

In contrast, for Model 3.2 (REL(*s*,*c*-of-T)), the number of OBDD-H nodes processed increased with *c*, as shown in Figure 4.28. Recall that the OBDD-A

follows this pattern as well, however the degree of increase for OBDD-H is smaller.

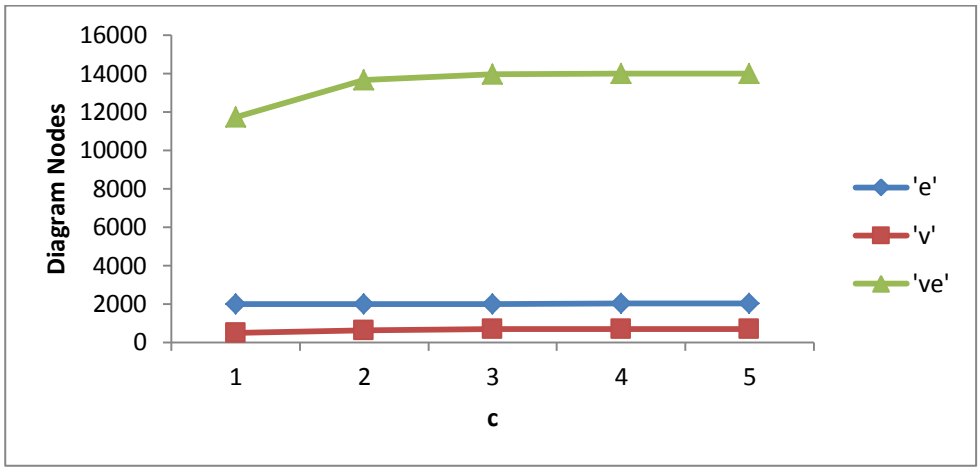


Figure 4.28: Diagram Nodes for 5×5 Grid with Varying *c* - Model 3.2

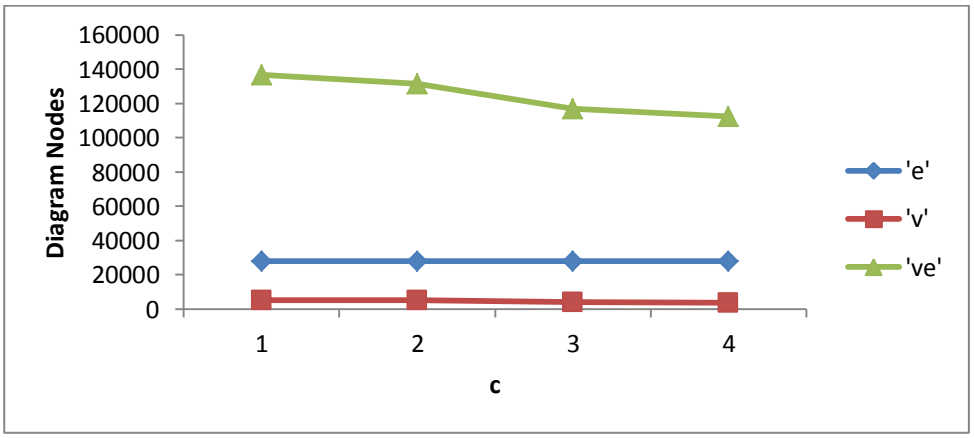


Figure 4.29: Diagram Nodes for 5×5 Grid with Varying c_i - Model 3.3

Model 3.3 (REL(s,c_i-of-T_i)) showed an unusual feature; a noticeable dip around c_i=3. The same dip is apparent for the OBDD-H, as shown in Figure 4.29. This appears to be related to the slight increase in the number of nodes for c_i=2 for the comparable OBDD-A; this increase doesn't occur as noticeably for the OBDD-H; instead the decrease from c_i=1 to c_i=2 is smaller than the decrease between c_i=2 and c_i=3, giving the appearance of a dip.

In all cases, the number of nodes generated for Model 've' is the greatest and the least nodes are generated for Model 'v', as seen with the OBDD-A.

4.9. Chapter Summary

The OBDD-H has been introduced and shown to use the same partitioning system as the BS algorithm [15, 18]. The OBDD-He shares the restriction inherent in the

partitioning system in that it only applies to undirected networks. It has been shown that it generates the same number of nodes as the OBDD-A for Models 1 and 2 ‘e’, and hence is far superior in terms of space complexity as compared to BS. OBDD-He has been shown to be at least competitive with BS in regards to processing time.

The notation introduced for BS has been extended to allow the solving of other component failure and network connectivity models. The general OBDD-H has been shown to have equal or better time performance than the OBDD-A for these models, despite requiring slightly more nodes to be processed. The OBDD-H was further extended to solve Model 3, with results similar to those of the OBDD-A.

In conclusion, the OBDD-H has been shown to be comparable or better than the OBDD-A, especially for network communication Model 2. However it can only be applied to undirected networks, while OBDD-A applies to both directed and undirected networks. Further, as shown in Chapter 5, the OBDD-H is not as efficient as the OBDD-A when computing performability measures such as EHC and EMD.

Chapter 5
Performability

5.1. Chapter Overview

Network reliability (REL) only measures the probability whether the network is connected or not, given the network model in use. The Expected Message Delay (EMD) is a more specific metric, which estimates the delay a message can be expected to experience when passing from the source(s) to the target(s). This is especially important for applications requiring some form of time-based guarantees.

While several algorithms in the literature [32-34, 73] claim to compute EMD, all assume that the delay of each edge is 1 and that each vertex has no delay. This is a special case of the EMD problem called Expected Hop Count (EHC). The algorithms which form part of this thesis are the first to be able to compute the general EMD problem [35], and can also be specialized to compute EHC [21, 22, 35, 45, 104].

The OBDD-A and OBDD-H introduced in Chapter 3 and Chapter 4, respectively, compute REL by storing the probability of the network state that a node represents in that node. This does not allow the computation of metrics, such as EMD, that require message delays. However the nodes of both algorithms can be modified to store message delays, as discussed in this chapter.

The concept of message delay was introduced in Section 2.6.1, and requires being able to measure the message delay in a given network state. This is discussed in Section 5.2. Section 5.3 describes the modification of the OBDD-A required to compute EMD and the extension of the OBDD-H for performability is discussed in Section 5.4. The use of both algorithms to compute EHC is discussed in Section 5.5 and performance results are presented in Section 5.6. Finally, Section 5.7 concludes the chapter.

5.2. The Delay of a Network State

Computing the EMD metric requires a measurement of the delay of paths through the network. For successful network states, the delays of minpaths are recorded together with the probability of getting a path with this delay. Once the

computation completes, these probabilities are used to compute EMD as discussed in Section 2.6.1.

The computation of delay for Models 1 and 2 are straight-forward. In the case of Model 1, the delay of the network state is the least delay of all minpaths because only a single minpath is required. For Models 2 and 3, the least delay between the source and any target is recorded and the greatest of these delays is taken as the delay for the network state. The minpaths with least delay are referred to as the *best* minpaths for the rest of this chapter.

Note that the more general versions of Model 2.1 and 2.2 that have no specific source vertex [15, 18] are not appropriate for EMD. This metric measures the delay of a communication, which indicates a delay between distinct points. If using the general model, it is not clear which pairs of vertices provide this delay; one option is to track the least delay between any pair of vertices and choose the greatest of these. Note that it is possible (and even extremely likely in the case of Model 2.2) that some of these delays would include multiple lower delays between other vertices.

An alternate method of counting the message delay would be to assume that each vertex in K only has to be connected to one other vertex in K, and calculate the delay of the network state appropriately. In either case, the delay between each pair of vertices in K (or V for Model 2.2) must be stored; potentially $|V|^2$ pieces of information. Storing this information requires a large amount of memory, and processing them would be time consuming. This thesis assumes that a specific source is named for each computation, which requires storing only the least delay between the source vertex and each target (only $|V|$ pieces of information).

Computing the message delay of a network state is more complicated for Model 3, especially for Models 3.2 and 3.3. In each case, only the least delay between the source and each target vertex is considered. For Model 3.1, only one minpath is required and hence the best of each of these minpaths is taken as providing the delay of the network state. Note that if some targets have not been reached the least delay of an existing minpath is taken. This has implications for termination testing as discussed in Sections 5.3.1.2 and 5.3.2.2.

For Model 3.2, at least c target vertices must be reached in order for the communication to be considered successful. As with other models, the minpath of least delay is taken for each target reached, however taking the least of these

minimal delays is not appropriate. Consider a network state with c = 3 where minpaths of delay 1, 2, 4 and 6 exist. Assuming the message travels at a uniform speed along each path, the target with delay 1 will be reached first, followed by the target with delays 2 and 4. At this point c=3 targets have been reached and the computation is successful. Hence the delay of this network state is 4. In general, the c^{th} shortest of the delays to a target is taken, where each delay is the least delay of all minpaths to that target.

Model 3.3 is similar to Model 3.2, except that now each group g_i requires that c_i target vertices in that group be reached. Again the least delay of all minpaths to a target is taken as the delay to that target vertex, as for all other models. As for Model 3.3, the c_i^{th} greatest of these delays is taken as the delay for each group g_i. Consider groups with delay 4, 4 and 5. The communication reaches the required number of target vertices first in the first two groups, but at this stage the communication is not yet successful. Only when the communication reaches time 5 are the requirements of the communication satisfied. Hence the greatest of group delays is taken as the delay of the network states.

The network state delay requirements are summarized in Table 5.1, where MINc is the c^{th} smallest of the inputs. For example MIN2(3,4,5,6) = 4.

Table 5.1: Summary of Network State Length Calculation

Model	**Delay of network state Ω, D(Ω)**
Model 1 (s,t)	D(Ω) = the length of the best (s,t)-minpath.
Model 2 (s,T)	D(Ω) = MAX(D(t)), where D(t) is the delay of the best (s,t)-minpath for each target $t\in$T.
Model 3.1 (s,1-of-T)	D(Ω) = MIN(D(t)), where D(t) is the delay of the best (s,t)-minpath for each target $t\in$T.
Model 3.2 (s,c-of-T)	D(Ω) = MINc(D(t)), where D(t) is the delay of the best (s,t)-minpath for each target $t\in$T.
Model 3.3 (s,c_i-of-T_i)	D(Ω) = MAX(D(T_i)) for all $T_i\in$T, where D(T_i) = MINc(D(t_i)) and D(t_i) is the delay of the best (s,t_i)-minpath for each target for $t_i\in T_i\subseteq$T.

5.3. OBDD-A for Computing EMD

The OBDD-A for performability is similar to the one for computing REL, except that more information is stored in each node. The OBDD-A for REL stores only the reliability of each network state. For performability, the diagram must also

store information relevant to the metric being computed. This section addresses the general OBDD-A algorithm.

This means that the algorithm itself does not change, although the methods that the algorithm calls do. For example, the vertex information (VI_i) stored in each node does not change and hence is not addressed in this section. Vertices likewise become redundant in exactly the same way as with REL, although more information must now be removed from each node.

This section analyses the changes needed to the OBDD-A in order to allow the computation of EMD. The alterations to the mathematical model are discussed in Section 5.3.1, while the changes to the pseudo-code are discussed in Section 5.3.2. An example of computing EMD using the modified OBDD-A is given in Section 5.3.3. Note that performance is discussed in Section 5.6, together with the performance of the performability OBDD-H.

5.3.1. The Mathematical Model of the OBDD-A for Performability

5.3.1.1 Nodes and Node Components

For the EMD metric, changes must be made to the mathematical models used to describe networks and diagram nodes. Each vertex v_i and edge e_j of the graph has a delay, $D(v_i)\geq 0$ and $D(e_j)\geq 0$ respectively[20]. It is assumed that the delays of source vertices and target vertices are zero, since information starts at the source vertices and is considered arrived as soon as it reaches the target. For networks with target vertices that can pass the information on (have edges leaving the vertex) the delay of the final target vertex in each minpath is assumed to be zero and all other delays are used normally[21].

Recall that, for REL, an OBDD-A node N_i consists only of a pair, $[VS_i, CI_i]$ (plus a Boolean variable for the OBDD-Ave)[22]. While this is sufficient to track network connectivity, it does not track information needed to compute EMD. For EMD, define $N_i=[VS_i, PI_i, CI_i]$ where VS_i is the same as for REL, CI_i is the modified condition information for EMD as described in Section 5.3.1.3, and PI_i is the additional Path Information required to track the lengths of paths to the vertices in

[20] This thesis assumes that delays are non-negative integers, but floating-point delays do not require a change to the algorithm.

[21] The algorithm can be easily modified to accommodate delay in the source vertex by adding the appropriate delay to the root node. Delays for the final target vertices can likewise be added to the delay of the minpath, if required.

[22] For the rest of this section, this Boolean variable will not be mentioned. For the modifications given, simply add s_i to the nodes as appropriate if using the OBDD-Ave.

VS_i. Each $M_i^x \in PI_i$ is a pair (PL_i^x, P_i^x) where PL_i^x is a list of pairs $\{(v_a, D_a^x), \ldots, (v_z, D_z^x)\}$ with $\{v_a, \ldots, v_z\} = VS_i$ and D_j^x is the least delay of all reaching paths to v_j.

Each M_i^x is referred to as a *component* of node N_i. Each P_i^x is the probability that the network is in the state represented by the component. Note that the sum of the component probabilities forms the probability of the node, making it unnecessary to store the node probability itself. The implementation retains P_i for compatibility reasons but the algorithm presented here omits it. Because each component contains VS_i, we do not write the VS_i of nodes for performability when detailing a node.

The idea behind components is that each represents a state of the network with a delay assigned to each vertex that has been reached. When a success node is found, these delays can be used to compute EMD. The delays stored in the components represent the least delay to that vertex, so if another path is found the delay of this may replace the existing delay if it is quicker. Thus components allow each OBDD-A node to represent a number of different network states, each with common connectivity (VS_i/CI_i) but possibly with different path delays (PI_i).

To be equal, both components must contain information on the same vertices, but since we only merge components whose nodes have identical VS_i (as discussed in Section 5.3.1.4) this is always true. Because components represent the path lengths of a network state, it is required that two equal components have equal path lengths to their respective vertices. Formally components $M_i^x = \{(v_\alpha, D_\alpha^x), \ldots, (v_\omega, D_\omega^x), P_i^x\}$ and $M_i^y = \{(v_\alpha, D_\alpha^y), \ldots, (v_\omega, D_\omega^y), P_i^y\}$ are equal (write $M_i^x = M_i^y$) iff $D_z^x = D_z^y \forall z = \{\alpha, \ldots, \omega\}$.

5.3.1.2 Node Types

Success testing for EMD doesn't only require checking the connectivity, but also checking the delays as discussed in Section 5.2. Consider a component that has a path with delay 5 to the only target vertex. If that component has a path with delay 3 to another (non-target) vertex then it is not necessarily successful because there is a chance that this delay 3 reaching path could be extended to a minpath with delay 4 (or even 3 if the edge followed has zero delay), which is an improvement over the current one. On the other hand if the shortest non-target path had delay 5 then the component would be successful because at best this could create another minpath with delay 5.

The test for component success thus requires both the least delay to each target vertex (if reached) and the least delay to a non-target vertex. These can either be computed each time the connectivity conditions are satisfied by a node, or tracked in an ongoing manner.

Because the components of a node store different delays, it is possible that one component is successful while another component in the same node is not. Hence success testing is done on a component level, and successful components are processed and removed from the node. Thus the test for a successful node reduces to checking if the node has any components remaining. Components are only removed when successful, so a node with no components is successful and is discarded. Note that once a single component of a node is successful, no child of this node will be a failure node since the minimum requirements for success are met. These minimum requirements are exactly those of the node being successful for REL.

5.3.1.3 Condition Information

In addition to lengths stored in components, conditions must also store path lengths. As described in Section 5.3.1.1, $N_i=[VS_i, PI_i, CI_i]$ where CI_i is the set of conditions. Each $C_i^x \in CI_i$ is a condition of the form (v_f, v_t, D^x), where v_f and v_t are the endpoints of the condition as with REL, and D^x is the minimum delay of all paths found between these vertices.

Because the EMD metric requires the best path for a connection, a number of changes must be made to the algorithm. Firstly we no longer delete all conditions adjacent to a vertex when that vertex is added to VS_i; an existing condition might lead to finding a quicker path to that vertex in the future. For the same reason we may add new conditions adjacent to a vertex already in VS_i.

Finally, if adding a condition to a node that already has a condition between the same pair of vertices, we may replace the existing condition if the new one is quicker. In practice, both conditions are identical except for the delay and hence the delay of the stored partition is changed to be the shorter of the two delays.

5.3.1.4 Node Isomorphism

The change to the model used for each node requires that the definition of isomorphism be considered. Currently two nodes are considered isomorphic (or equal) if they have the same VS and CI. Since those two factors determine the

connectivity state of the network represented by the node, it can be seen that two nodes are considered ‘equal’ only if they represent the same connectivity.

Each component in a node represents the ‘goodness’ of the connectivity, in this case measured in terms of message delay. It is possible to define node equality (isomorphism) so that two nodes are only equal if they have the same ‘goodness’ in addition to the same connectivity. This would mean requiring that the component of both nodes be equal, in addition to VS and CI being equal.

While this option seems logical, it generates a very large number of diagram nodes. Because isomorphism checks a new node against each node in Q_N, this greatly increases the number of checks, which has been shown in Section 3.5 to be one of the dominating factors in processing time complexity. The benefit of this approach would be that each node would have a single component, since each node represents only one measure of ‘goodness’.

The node definition in Section 5.3.1.1 gives a list of components, meaning that while each node represents a single connectivity state, the ‘goodness’ of this connectivity state can vary. In this case two nodes are equal if they have the same connectivity, which means that the isomorphism check between nodes does not change. While the number of components in a node can theoretically be large, they do not affect the number of nodes or the comparison of nodes for isomorphism.

When two nodes are isomorphic, their VI and CI are identical, but their PI is not. Hence the two component lists in PI must be merged. In practice, the components in the newly created node are added to the component list of the stored node. If the stored node already has a component that is equal to the new component, its probability becomes the sum of the probabilities of both components to indicate that this configuration is now more likely. If no stored node is identical then the new component is added to the existing list.

If a node has *m* components, then the children of that node will have *m* or less components since each set of path lengths creates exactly one new set of path lengths. However the modification to the path lengths in a component can make it identical to an existing component, in which case both are merged. Since the root node has one component, its children will each have one component. The merging of nodes can create nodes with multiple components.

```
      OBDD-Ap (G):     // For computing EMD
      1)  Initialize the root node N0;
      2)  Initialize k=0, QC = { N0 }, QN = { }, Pr(D)=0;
ve    3)  Initialize ke=0, kv=0, redf=false;
      4)  Remove the first node, Ni, from QC;
ve    5)  if ( (k > kv + ke) and si == false )
ve    6)      Relabel Ni as N2i+1;
ve    7)  else
      8)      N2i+2 = CREATE-POS-CHILDp(Ni, k);
      9)      N2i+1 = CREATE -NEG-CHILDp(Ni, k);
      10) for each child node Nfi do
e     11)     if redf then                    // ek = (vf,vt) or {vf,vt}
e     12)         DEL-REDUNDANTp( Nfi, vf );
e     13)     if redt then
e     14)         DEL-REDUNDANTp( Nfi, vt );
v     15)     DEL-REDUNDANTp( Nfi, vk );
ve    16)     if redf then
ve    17)         DEL-REDUNDANTp( Nfi, vkv );
      18)     if ( NON-TERMINAL-NODEp(Nfi) ) then
      19)         Check each node on QN for isomorphism with Nfi;
      20)         if ( an isomorphic node Nx was found ) then
      21)             MERGE-NODESp(Nx, Nfi);
      22)         else
      23)             Add Nfi onto QN;
      24) if ( QC == { } ) then
      25)     if ( QN == { } ) then
      26)         Compute EMD;
      27)         return EMD;
      28)     else
      29)         UPDATE-LEVELp( k );
      30) goto 4)
```

Figure 5.1: The OBDD-Ap Algorithm for EMD

5.3.2. The OBDD-A Performability Algorithm

The main OBDD-Ap algorithm is shown in Figure 5.1. Before that main algorithm starts, the implementation must load the network details from a file. For EMD, the file must specifically list component delays; when an edge or vertex has no delay listed it is assumed that the delay is zero.

Apart from the changes to the network file, the call to MERGE-NODESP and the use of the Pr(*D*) vector, the algorithm for the OBDD-A computing performability is identical to the general OBDD-A for computing REL shown in Fig. A-14 of Appendix A. The general performability methods are called instead of the general reliability ones, but the structure of the code is the same. The postfix 'P' in the method names denotes that they are the versions dealing with performability.

The computation of EMD is accomplished using Eq (2.3) given in Section 2.6.1. The probabilities Pr(*D*) are summed over all delays *D* to give REL. This then becomes the denominator in the equation, with the numerator summing the products of Pr(*D*)×*D*. If REL is also required, this can be returned through a global variable in addition to EMD.

It would be possible to add directly to both sums (the sum of probabilities and the sum of products) and store these, instead of storing the probabilities Pr(*D*). One advantage of the approach used in this work is that the probabilities allow a range of other metrics to be computed, not just the EMD.

For example, we could compute the probability that the network meets a minimum standard of efficiency. The probability that the delay is less than 10 units can be found using $\sum_{L<10} \Pr(D)$. Such a metric may be of more use than an expected value for some applications.

The full root node (initialized in line 1) is N_0=[{({({v_0},0)},1.0)}, {}][23], as is the case for every performability OBDD-A for Models 1, 2 and 3.

5.3.2.1 Adding and Removing Components

The root node starts with component ({({v_0},0)},1.0); that is it consists of a pair with the source vertex v_0 and the delay 0, in addition to the probability 1.0. This indicates that the communication is at v_0 and has not yet had to travel any distance, and that the probability of being able to reach this state is 1.0. Conditions can be added or removed from a node without affecting the components, but when conditions are triggered the components of the node must be modified appropriately.

```
     TRIGGERp (N_i, v_t):
     1)  for each C = ( v_t, v_x ) ∈ CI_i do
     2)      Add v_x to VS_i;
e    3)      Add x to list L;                  // Remember new vertices
     4)      ADD-COMPONENTp(N_i, C);
e    5)      Delete C from CI_i;
e    6)  for each x in List L do
e    7)      TRIGGERp (N_i, v_x );
     8)  return N_i;
```

Figure 5.2: TRIGGERp for EMD

The TRIGGERP method shown in Figure 5.2 is similar to the general TRIGGER method shown in Fig A-3 of Appendix A, except that it adds a component whenever a vertex is added to VS_i, and that it doesn't delete conditions adjacent to this vertex. As discussed above, these conditions may be used later to complete a

[23] Recall that we do not write VS_i for performability nodes; $VS_0 = \{v_0\}$ is clear from the component. Also recall that we assume $D(v_0)=0$ since v_0 is a source vertex. If the delay is non-zero, replace the zero delay in the root node with the new delay.

shorter reaching path and hence are kept until one of their endpoints becomes redundant.[24]

```
RESOLVEp (Ni, vx):
1)  foreach Cf = (vf, vx, D1) ∈CIi do
2)      foreach Ct = (vx, vf, D2) ∈CIi with f≠t do          // Avoid self loops
3)          if ∃ (vf, vt, D) ∈ CIi then
4)              if ( (D1+D(vx)+D2) < D ) then
5)                  Replace (vf, vt, D) with (vf, vt, D1+D(vx)+D2) in CIi;
6)          else
7)              Add (vf, vt, D1+D(vx)+D2) to CIi;
8)  return Ni;
```

Figure 5.3: RESOLVEp for EMD

The RESOLVEP method shown in Figure 5.3 remains largely unchanged from the RESOLVE in Fig. A-4 of Appendix A, but must be updated to reflect that each condition has a delay. When two conditions are resolved, their delays are added together with the delay of the vertex joining the path. The new path has delay equal to the delay of the existing component to v_f plus the delay of vertex v_f plus the delay of the input condition (lines 5 and 7).

The general RESOLVE method for REL only adds a condition if no such condition already exists in the node. Like TRIGGERP, RESOLVEP may overwrite existing information (in this case a conditional path) if a shorter path has been found (lines 4-5).

```
ADD-COMPONENTp (Ni, C):          // C = (vf, vt, D)
1)  for each M_i^x ∈ PI_i do
2)      Locate (v_f, D^x) ∈ M_i^x;          // Must exist if this method is called
3)      if (v_t, D_2^x) ∈ M_i^x then
4)          if ( (D + D(vf) + D^x) < D_2^x ) then          // New path is shorter
5)              Replace (v_t, D_2^x) with (v_t, D + D(v_f) + D^x) in M_i^x;   // else do nothing
6)      else
7)          Add (v_t, D + D(v_f) + D^x) to M_i^x;
8)  return Ni;
```

Figure 5.4: ADD-COMPONENTp for EMD

When adding information to a component, the condition from TRIGGERP is matched against existing components to the first endpoint. This endpoint must exist in VS_i since it is added immediately prior to the component being added. The ADD-COMPONENTP method shown in Figure 5.4 will replace existing component information if the new path is shorter than any existing path for that component vertex.

[24] For fallible vertices, TRIGGERp is called on v_k only when v_k has been decided as available. In this situation it is always the case that v_k becomes redundant on this level. This is not true for models with perfect vertices, however.

Vertex information is removed from components using the DEL-REDUNDANTP method shown in Figure 5.5. Each component simply has the pair containing the redundant vertex removed. Note that CHECK-REDUNDANT does not involve components in any way and hence is unchanged.

```
DEL-REDUNDANTp ( N_i, v_x ):
1)  if (v_x ∈ VS_i ) then
2)      Delete v_x from VS_i;
3)  for each ( C ∈ CI_i ) do    // C = (v_a, v_b)
4)      if ((x == a) or (x == b)) then
5)          Delete C from CI_i;
6)  for each ( M ∈ PI_i ) do
7)      Remove the (v_x, D) pair from M;
8)  return N_i;
```

Figure 5.5: DEL-REDUNDANTp for EMD

```
COMP-UPDATEp (N_i, Prob):
1)  for each M_i^x ∈ PI_i do
2)      P_i^x = P_i^x × Prob;
3)  return N_i;
```

Figure 5.6: COMP-UPDATEp for EMD

Note that adding or removing vertex information to a component does not change its probability; instead the probability of a node is changed in CREATE-POS-CHILD and CREATE-NEG-CHILD by calling COMP-UPDATEP. Merging two nodes (shown in Figure 5.12) modifies component probability directly, without calling COMP-UPDATEP. The COMP-UPDATEP method is shown in Figure 5.6; it traverses all components in the node and multiplies their probability by the given amount. This is used to modify the components to reflect variables being decided.

Nodes no longer store probabilities directly. Instead, the probability formerly stored in a node is equal to the sum of the probabilities stored in the components of that node. When the probability of a node would be modified for REL, the probabilities of the components are modified for the performability OBDD-A. For example if two nodes are merged, the components of the two nodes are merged; if both nodes have equal components their probabilities are summed, or if not the unmatched component (and its probability) is moved into the existing node.

5.3.2.2 Success and Failure Testing

For the OBDD-A that computes REL, a node is tested for success or failure as a single entity. For EMD a node may not be terminal even if the appropriate targets have been reached. Each individual component now needs to be tested for success,

and it is possible that a node has some components that are successful while others are not, as discussed in Section 5.3.1.2.

When components are found to be successful they are removed from the node and their details are added to the appropriate metrics. For example a component representing a network state with delay 5 and probability 0.1 would add 0.1 to the probability of the delay being 5. If all components are removed, the node is declared a success node; the only way a node can have no components is if they have been removed, meaning they were all successful.

Because the success of a node is now based on the success of its components, it would be possible to test for component (and hence node) success whenever a component has its vertex information altered. This can occur several times during the creation of one child node though, and with EMD the success test requires more processing than that for REL. Hence the success of components is checked once per child node using the modified TEST-NODEP method shown in Figure 5.7.[25]

The TEST-NODEP method is modified from the general TEST-NODE method by replacing the body of the first **if** condition statement with a call to TEST-COMPONENTSP (line 2); when this returns the method checks to see if the component list is empty.

```
TEST-NODEp (Ni):
1)  if ( |Tj∩VSi| > kj ∀j=0…m ) then // only count decided vertices
2)      TEST-COMPONENTSp( Ni );
3)      if ( PIi is empty ) then
4)          return 1;  // a success node
5)      else
6)          return 3; //a non-terminal node
7)  else if ( VSi == { } ) then
8)      return 2;  // a failure node
9)  else
10)     return 3; //a non-terminal node
```

Figure 5.7: TEST-NODEp for EMD

The general TEST-COMPONENTSP method is shown in Figure 5.8. The way in which the shortest non-target length and the c^{th} shortest target length for each group are computed is not specified, since that will depend to some extent on the implementation. When a component is found to be successful with delay D, the probability that the delay is D (written Pr(D) in the general case) is incremented by

[25] An alternative is to store relevant information in each node, such as the minimum delay to a non-target vertex. This requires additional storage and may need to be updated several times per node (especially for Model 'v'), so the implementation used in this thesis does not take this approach.

the probability of the component. The successful component is then deleted from the node. Note that REL is not directly updated when EMD are computed; REL is obtained as part of computing these metrics as discussed in Section 2.6.1.

```
TEST-COMPONENTSp (Ni):
1)  Compute D, the shortest delay to a non-target vertex;
2)  for each M_i^x ∈ PI_i do
3)      for each target group Tx⊆T do
4)          max = 0;
5)          Compute Dx, the cx th shortest delay to vertices in Tx;
6)          if ( Dx > D ) then
7)              break;
8)          else if ( Dx > max ) then
9)              max = Dx;
10)     if ( no Dx was greater than D ) then
11)         Pr(max) += P_i^x;   // Update probability of max
12)         Delete M_i^x from PIi;          // Delete successful component
```

Figure 5.8: TEST-COMPONENTSp for EMD

5.3.2.3 Creating Child Nodes

The CREATE-POS-CHILDP and CREATE-NEG-CHILDP methods for the OBDD-A model being used do not change to accommodate EHC/EMD, other than to use the modified ADD-CONDP, TRIGGERP and RESOLVEP methods in place of TRIGGER and RESOLVE, and to call COMP-UPDATEP to update component probability. The node probability, condition adding and resolution of conditions are unchanged.

```
        ADD-CONDp (Ni, (vf, vt, D)):
        1)  if (∃ (vf, vt, D1)∈CIi) then
        2)      if (D1>D) then
        3)          Replace (vf, vt, D1)∈CIi with (vf, vt, D);      // Only change length
        4)      return Ni;
v,ve    5)  Add (vf, vt, D) to CIi;
e       6)  if ( vf ∈VSi ) then
e       7)      Add vt to VSi;
e       8)      TRIGGERp(Ni, vt );
e       9)  else
e       10)     Add (vf, vt,D) to CIi;
e       11)     for each C = ( vt, vx, D1 ) ∈ CIi and x≠f do
e       12)         ADD-CONDp (Ni, (vf, vx, D + D(vt) + D1 ) );
e       13)     for each C = ( vx, vf , D1) ∈ CIi and x≠t do
e       14)         ADD-CONDp (Ni, (vx, vt, D + D(vt) + D1) );
        15) return Ni;
```

Figure 5.9: General ADD-CONDp for EMD

The ADD-CONDP method shown in Figure 5.9 is considerably different from ADD-COND due to the introduction of delays. If a condition already exists that matches the vertices of the new condition, the new condition may still be added if it

has a shorter delay. Also, since conditions are not deleted when one of their endpoints is reached, new conditions can also be added when an endpoint has already been reached.

```
      CREATE-POS-CHILDp (N_i, k):
      1)  Create N_j as a copy of N_i;
      2)  P_change = 1;
v     3)  f = k; x = k;
e     4)  x = k;      // e_k = (v_f, v_t) or (v_t, v_f) or {v_f, v_t} (f < t)
ve    5)  x = ke;     // e_ke = (v_f, v_t) or (v_t, v_f) or {v_f, v_t} (f < y)
ve    6)  if kv + ke == k then            // Deciding a vertex
ve    7)      P_change = P_change × Pr(v_kv);
ve    8)      s_j = true;                 // Mark the vertex as active
ve    9)  else
v     10) for each (e_x = (v_k, v_t, D) or (v_t, v_k, D) or {v_k, v_t, D }) and ( t > k ) do
      11)     P_change = P_change × Pr(e_x);
      12)     ADD-CONDp(N_j, e_x);            // General Version
      13)     if ( e_x is undirected ) then
      14)         ADD-CONDp(N_j, (v_t, v_f, D) );
      15) P_j = P_j×P_change;
      16) COMP-UPDATEp( N_j, P_change);
ve    17) if (redf) then
v,ve  18)     if v_x∈VS_2i+2 then
v,ve  19)         TRIGGERp(N_j, v_x );        // Add vertices to VS
v,ve  20)     else
v,ve  21)         RESOLVEp(N_j, v_x );        // Combine conditions through v_x
      22) return N_j;
```

Figure 5.10: CREATE-POS-CHILDp for EMD

There are a number of changes to CREATE-POS-CHILDP (shown in Figure 5.10) when compared to the general CREATE-POS-CHILD method for the OBDD-A. For example, because all probability modifications must be applied to all components they are combined into the P_{change} variable and then applied in a single call to COMP-UPDATEP (line 23). In addition, all conditions now have a delay, and new conditions are created by extending added ones (lines 10-14).

```
      CREATE-NEG-CHILDp (N_i, k):
      1)  Relabel N_i as N_j;             // N_i no longer needed
e     2)  P_change = P_i × (1 - Pr(e_k));
v     3)  P_change = P_i × (1 - Pr(v_k));
ve    4)  if kv + ke == k then            // Deciding a vertex
ve    5)      P_change = P_i × (1 - Pr(v_kv));
ve    6)      s_j = false;
ve    7)  else      // Deciding v_ke with s_i==true
ve    8)      P_change = P_i × (1 - Pr(e_ke));
ve    9)  P_j = P_j×P_change;
      10) COMP-UPDATEp( N_j, P_change);
ve    11) if (redf) then // v_kv redundant
ve    12)     if v_kv∈VS_i and s_j==true then
ve    13)         TRIGGERp(N_j, v_kv );       // Add vertices to VS
ve    14)     else
ve    15)         RESOLVEp(N_j, v_kv );       // Combine conditions through v_kv
      16) return N_j;
```

Figure 5.11: General CREATE-NEG-CHILDp for EMD

Like CREATE-POS-CHILDP, the CREATE-NEG-CHILDP method shown in Figure 5.11 uses P_{change} to adjust the node probability and the component probabilities although in this case it is more a case of convenience than performance enhancement. No conditions are added, so the change in mathematical model of the nodes doesn't have an impact.

5.3.2.4 Node Isomorphism

As discussed in Section 5.3.1.4, the test for node isomorphism is extremely similar for the performability OBDD-A when compared to the reliability OBDD-A. The only change is that, when comparing conditions, the delay must be compared in addition to the endpoints. This delay doesn't change the pseudo-code of the method, since it is hidden in the definition of condition equality.

For REL, the existing node is updated by having the probability of the new node added to its own; for EMD the components also need to be added. This makes merging nodes more complicated for performability than for REL; instead of simply considering the node probability the performability OBDD-A must move component information into the existing node. The process is shown in Figure 5.12.

```
MERGE-NODESp (N_old, N_new):
1)  P_old += P_new;
2)  foreach M_new∈PI_new do
3)      foreach M_old∈PI_old do
4)          if ( M_new = M_old ) then        // Component exists
5)              P_old += P_new;
6)              break;
7)      if no matching M_old was found then
8)          Add M_new to PI_old;             // Add to end of list
9)  return N_old;
```

Figure 5.12: MERGE-NODESp for EMD

The comparison of the components of two nodes for merging requires a significant amount of processing, since each component in the second node may potentially be compared to all components in the first node in order to find a match. Even with the number of components reduced by merging, this still requires a considerable amount of processing time. The implementation in this thesis sorts the information in each component and the components in the list to reduce the number of comparisons.

5.3.3. Example

Consider the network shown in Figure 5.13, with $s = v_0$, $t = v_9$, all vertices having a probability of 0.9 of being available and each edge always active. Each edge has its delay shown in brackets (*e.g.*, e_0 has delay 1, e_1 has delay 2, *etc.*) and each vertex has delay 1 apart from v_0 and v_9, which have delay 0.

The root node of the OBDD-A is N_0=[{({(v_0,0)},1.0)}, {}]. The other variables are initialized (k=0; $Q_C = Q_N = \{\ \}$ and all Pr(D) are set to 0) and then N_0 is pushed onto Q_N. The complete OBDD-A is given in Appendix B, since components are difficult to show in a diagram.

The first (and in this case only) node, N_0=[{({(v_0,0)},1.0)}, {}], is removed from Q_C. The CREATE-POS-CHILDP method creates N_2 as a copy of N_0, adds the appropriate edges as conditions. Since the problem uses Model 'v', the algorithm executes line 10 of CREATE-POS-CHILDP; causing lines 11-14 to be repeated for each edge adjacent to v_0.

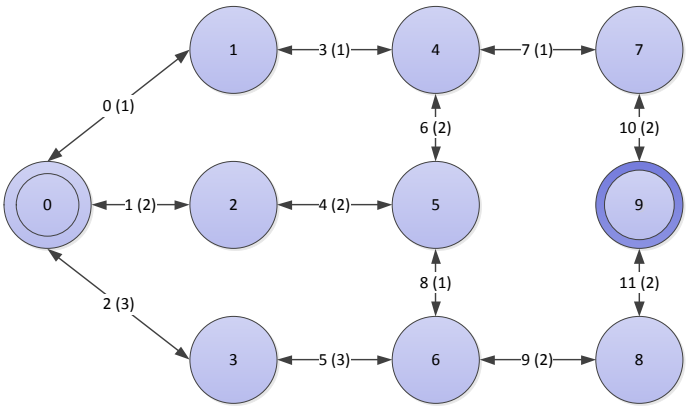


Figure 5.13: Advanced Network with Delays

Edge e_0 is added as condition (v_0,v_1,1), edge e_1 adds condition (v_0,v_2,2), and edge e_2 adds condition (v_0,v_3,3). All conditions are added through ADD-CONDP, and in each case the condition does not already exist (lines 1-4 of Figure 5.9) and is thus added directly to CI_2. When the loop of edges completes, N_2=[{({(v_0,0)},1.0)}, {(v_0,v_1,1), (v_0,v_2,2), (v_0,v_3,3)}].

Once all conditions are added, the probability of the existing components is updated (line 15 of CREATE-POS-CHILDP); in this case P_{change} is based entirely on Pr(v_0) (since each edge is infallible) giving a multiplier of 0.9. This results in N_2=[{({(v_0,0)},0.9)}, {(v_0,v_1,1), (v_0,v_2,2), (v_0,v_3,3)}].

Finally the new conditions (and any existing conditions adjacent to v_0) need to be triggered; if v_0 was not in VS_2 then they would be resolved instead, joining them through v_0. TRIGGERP loops through all conditions adjacent to v_0 (line 1) and adds their *to* endpoint to VS_2 (line 2) and to the component list (line 4). The only component becomes ({(v_0,0), (v_1,1), (v_2,2), (v_3,3)},0.9). Note that if v_0 had a non-zero delay, this would have been added to the delay of each of the new components in line 7 of ADD-COMPONENTP.

The new component information indicates that reaching paths of delay1 have been found to v_1, v_2 and v_3. Note that the algorithm does not store which edges this path uses; this information is irrelevant to the computation of EMD.

Returning to the main loop (line 13) the computation next creates N_1. The negative child, N_1, is created by relabeling N_0, with no change other than to multiply by the probability of v_0 being unavailable; 0.1. Hence N_1=[{({(v_0,0)},0.1)}, {}]. Note that there is no information in the component relating to v_1, v_2 and v_3, since these have not been reached.

Finally information on inactive vertices must be removed from both child nodes. Since v_0 is redundant N_1=[{({},0.1)}, {}] and N_2 = [{({(v_1,1), (v_2,2), (v_3,3)},0.9)}, {}]; only lines 10 and 15 are applicable for Model 'v'.

Each of the two child nodes is tested for termination; N_1 is found to be terminal and is discarded but N_2 is found to be non-terminal. Since Q_N is empty, N_2 is added without the necessity for any comparisons. The queues are swapped, giving Q_N = {N_2} and Q_C = { }. UPDATE-LEVELP then updates *k* to 1.

Execution returns to the top of the main loop and the first (and only) node, N_2 = [{({(v_1,1), (v_2,2), (v_3,3)},0.9)}, {}] is removed from Q_C and processed to give N_5=[{({(v_1,1), (v_2,2), (v_3,3)},0.09)}, {}] and N_6=[{({(v_1,1), (v_2,2), (v_3,3) , (v_4,3)},0.81)}, {}]. Note that the delay to v_4 was computed from the sum of the delay to v_1, the delay of v_1 itself (1) and the delay of the edge (v_1,v_4,1). After calls to DEL-REDUNDANT the nodes become N_5=[{({(v_2,2), (v_3,3)},0.09)}, {}] and N_6=[{({ (v_2,2), (v_3,3) , (v_4,3)},0.81)}, {}], meaning that both are non-terminal.

The full process is not discussed here; all nodes of the diagram are given in Appendix B but this example discussed only the relevant parts of the process.

The negative children of N_{24}=[{({(v_6,7)}, 0.0081)}, {}] and N_{28}=[{({(v_4,3), (v_6,7)}, 0.0729)}, {}] are N_{49}=[{({(v_6,7)}, 0.00081)}, {}] and N_{57}=[{({(v_6,7)}, 0.00729)}, {}] respectively. The two child nodes are both non-terminal and are found to be

isomorphic. Both have a single component, so MERGE-NODESP compares these for isomorphism. Both components contain delay 7 paths to v_6 and no other paths; since the delays of all paths are identical the components are isomorphic. Hence the merged node is N_{49}=[{({(v_6,7)}, 0.0081)}, {}] which has a single, merged component.

The positive children of N_{28}=[{({(v_4,3), (v_6,7)}, 0.0729)}, {}] and N_{30}=[{({(v_4,3), (v_5,5), (v_6,7)},0.6561)}, {}] are N_{58}=[{({(v_5,6), (v_6,7), (v_7,5)}, 0.06561)}, {}] and N_{62}=[{({(v_5,5), (v_6,7), (v_7,5)},0.59049)}, {}]. The two child nodes are both non-terminal and are found to be isomorphic and both again have a single component. In this case the components have delay 6 and 5 paths to v_5, respectively. Since the delays differ, the components are not isomorphic. Merging the nodes gives N_{58}=[{({(v_5,6), (v_6,7), (v_7,5)}, 0.06561), ({(v_5,5), (v_6,7), (v_7,5)}, 0.59049)}, {})], which has two components.

For computing REL under Model 'e', the node N_{414}=[{({(v_8,10), (v_9,17)}, 0.00531441), ({(v_8,10), (v_9,13)}, 0.0531441), ({(v_8,11), (v_9,8)}, 0.00531441), ({(v_8,10), (v_9,8)}, 0.57927069}, {})] would be a success node since v_9 has been reached. Since this example uses Model 'v' the node cannot be a success node since it isn't known whether v_9 is available.

In order to address the issue of component success in EMD, consider this node under Model 'e'. In this case each component is considered separately, to see whether that component is successful. Those that are, have their information stored and are removed, while those that are not remain in the node. If all components are successful the node is a success node; otherwise the node is non-terminal and is stored.

The first and second components have quicker reaching paths to v_8 than their minpaths to v_9. In both cases, this means that it is possible that a quicker minpath (*i.e.*, one with less delay) can still be found. For this reason these components are not found to be successful.

The last two components have a path leading to v_9 with a shorter delay than any other reaching paths (they have delays 11 and 10, respectively to v_8). Hence both of these components would be successful under Model 'e' and their information would be stored (*e.g.*, for component ({(v_8,11), (v_9,8)}, 0.00531441) the probability of a delay 8 path would be increased from 0 to 0.00531441). Hence, under Model

'e', the node would be stored on Q_N as N_{414}=[{({(v_8,10), (v_9,17)}, 0.00531441), ({(v_8,10), (v_9,13)}, 0.0531441)].

Returning to the example, after the main loop completes the stored probability values are Pr(D=8)=0.59049, Pr(D=13)=0.18452222, Pr(D=14)= 0.0004783, and Pr(D=17)= 0.00047830. Summing these probabilities gives REL= 0.77596881, which is identical to the REL that would have been achieved if the OBDD-A computing REL had been used. The EMD is computed from REL and the individual probabilities as $\frac{8\times0.59049+13\times0.1845+14\times0.0005+17\times0.0005}{0.7760}$ = 9.21. This means that, on average, we can expect a message to take 9.21 units of time to reach the target, and there is a 78% chance that it will successfully get there.

5.4. OBDD-H for Computing EMD

The extension of the general OBDD-H to performability is somewhat more complex than for the general OBDD-A since not only the distance to the source(s), but also the distance to each other member of the partition must be tracked. This section introduces the modifications required to compute EMD with the general OBDD-H.

Section 5.4.1 introduces the theoretical concepts required to compute EMD with the OBDD-H, and the pseudo-code for doing so is presented in Section 5.4.2. An example is given in Section 5.4.3.

5.4.1. The Mathematical Model

5.4.1.1 Nodes and Node Components

The general OBDD-H for computing EMD requires additional information beyond what is required for REL; as with the OBDD-A components are used to store this information. The difference is that the OBDD-A tracks reaching paths and minpaths, meaning that OBDD-A components store the delay of each of these. The OBDD-H algorithm instead tracks inter-connectivity and hence the components must do likewise. Hence the OBDD-H node for performability has the format N_i=[$Part_i$, PI_i].

OBDD-A components are relatively efficient since they only need to store a number of delays equal to the size of VS_i along with a probability, as well as adding a delay to each stored condition. However for the OBDD-H, each block of size *b* needs (*b*-1)! delays; the minimum delay of each vertex to every other vertex

in the partition. Because the network is assumed to be undirected, we only need to store the delays in one direction. For example, for a node with partition $[v_2,v_4]^*[v_3,v_5,v_7][v_6]$ the component must store the delay between v_2 and v_4 and the lengths between v_3, v_5 and v_7. In addition, the component must also store the path length between the vertices v_2 and v_4 in the marked partition and the closest source vertex.

The delays in each Hybrid component can either be stored as a linked list of $ng = \sum_{i=1}^{b-1} i = \frac{(b-1)(b-2)}{2}$ elements or as half of a (b-1)×(b-1) matrix. For the matrix we require (b-1)^2 items because the position in the matrix implicitly gives information on the vertices, however more than half of this information is redundant. Furthermore the use of a matrix requires either that the size of each block be computed beforehand (*i.e.*, that the partition be fully computed before the components are created) or that the matrices in the components be resized when the partition size changes. This is further complicated by the process of merging partitions using JOINHP since this also impacts the components. Note that performability OBDD-H methods are marked as such with the suffix 'HP'.

When deciding one edge per level, such as the case with the OBDD-He and OBDD-Hve, the first option can be easily implemented. However when multiple edges are decided on one level, such as with OBDD-Hv, the latter may be required since the algorithm applies the modification due to one edge at a time to the positive child. The advantage of using a matrix is that each delay can be easily accessed without requiring a search through the linked list. The disadvantage is the need to resize the matrix when elements are added or removed.

A matrix for the partitioning $[v_2,v_4]^*[v_3,v_5,v_7][v_6]$ may look like $\begin{bmatrix}1 & 2\\ - & 3\end{bmatrix}\begin{bmatrix}1 & 3\\ - & 2\end{bmatrix}[]$. In this case the first matrix gives the delays between v_2 and the source (1), v_4 and the source (2) and v_2 and v_4 (3). The lower-left value is not needed since it would be identical to the value in the upper-right and the diagonal giving the delay of self-loops has been removed. Similarly the second matrix gives the delay between v_3 and v_5 (1) v_3 and v_7 (3) and v_5 and v_7 (2). There is no matrix required for the third partition ([6]). Without these reductions the matrices would have been $\begin{bmatrix}0 & 1 & 2\\ 1 & 0 & 3\\ 2 & 3 & 0\end{bmatrix}\begin{bmatrix}0 & 1 & 3\\ 1 & 0 & 2\\ 3 & 2 & 0\end{bmatrix}[0]$, which contains the same information but at a greater cost in both storage and processing.

If using a linked list, each element must store two vertices plus the delay between them. A better version of the linked list stores the elements in the same order as the matrix, meaning that each list element is only required to store the delay and not the vertices relating to the delay. Hence the partitioning [2 4]*[3 5 7][6] can be represented by either of the following lists:

$$[(v_0,v_2,1),(v_0,v_4,2),(v_2,v_4,3)],[(v_3,v_5,1),(v_3,v_7,3),(v_5,v_7,2)]$$

$$[(1)(2)(3)][(1)(3)(2)]$$

Each of these lists is composed of two sub-lists, each representing one of the two blocks in the partitioning. This makes searching the list quicker as well as making the structure of the list similar to that of the partitioning. This work uses matrices for the examples, to assist reader comprehension. The implementation uses nested linked lists.

Whatever the implementation, component x of node N_i is defined as $M_i^x = [L_0,L_1,\ldots,L_{g-1},P_i^x]$ where each L_j is a list (or matrix) representing the inter-connectivity of block j of $Part_i$ and P_i^x is the probability of M_i^x. Hence the OBDD-H node is $N_i = [Part_i, PI_i]$ where PI_i is a list $(M_i^0, \ldots M_i^y)$ of y components. $Part_i$ is retained in the node to serve as a reference to the partitions in each component of PI_i.

Note that $Part_i$ does not change; all delays are recorded in the components, including the delays that are stored in CI_i for the OBDD-A. This means that node isomorphism for the performability OBDD-H is identical to the reliability OBDD-H. For this reason neither the partitioning nor isomorphism is discussed again in this section.

5.4.1.2 Node Type

The same issues discussed in Section 5.3.1.2 apply equally to OBDD-H. Again, success is determined by the success of components. Each component is tested in turn, with successful components being removed from the node. A node is successful when no components remain.

The success of a component is determined in a similar way that of OBDD-A components. Only the first delays in a component, representing the distance between the vertices in the marked block and the source, need to be checked because these are the delays of the reaching paths and minpaths in the network state.

5.4.2. The OBDD-H Performability Algorithm

The OBDD-H algorithm for performability is similar to that for reliability, except that methods called are the ones specific to performability as denoted by the 'HP' suffix. All changes to the algorithm are hidden within these methods except that the merging of nodes (line 16) now requires a method instead of simply summing probabilities.

```
OBDD-H (G):        // General algorithm for EMD
1)   Initialize the root node N0;
2)   Initialize redf, redt, F0, F1, reliability = 0;
3)   Initialize level variables;
4)   Remove the first node, Ni, from QC;
5)   for each edge ex being decided  do
6)        Npos = CREATE-POS-CHILDhp(Ni, ex, k);
7)   Nneg = CREATE-NEG-CHILDhp(Ni, k);
8)   for each child node Nfi do
9)        if redf then
10)            DEL-REDUNDANThp( Nfi, vf );
11)       if redt then
12)            DEL-REDUNDANThp( Nfi, vt );
13)       if ( NON-TERMINAL-NODEhp(Nfi) ) then
14)            Check each node on QN for isomorphism with Nfi;
15)            if ( an isomorphic node Nx was found ) then
16)                 MERGE-NODEShp(Nx, Nfi);
17)            else
18)                 Add Nfi onto QN;
19)  if ( QC == { } ) then
20)       if ( QN == { } ) then
21)            return Pr(D) array;
22)       else
23)            UPDATE-LEVELh( k );
24)  goto 4)
```

Figure 5.14: The OBDD-Hp Algorithm for EMD

The manipulation of components is carried out in the creation of the child nodes (lines 5-7). When redundant information is deleted (lines 9-12), redundant information is also removed from components. The check for a node termination (line 13) is changed to account for testing of components, and merging two nodes (line 16) now requires dealing with components as well.

5.4.2.1 Adding and Removing Components

Components are added or modified when a partition is added for an available edge, but this does not change the upper-most child creation methods. CREATE-POS-CHILDHP (Figure 5.15) is almost identical to the general REL versions except for two things; it calls the performability version of EDGE-CONTRACTHP and updates the probabilities of components rather than the probability of the node itself.

```
     CREATE-POS-CHILDhp(N_i, g, k):          // g is list of active edges for this child
     1)  Create N_j as a copy of N_i;
     2)  P_change = 1.0;
     3)  Set x to be the index of edge being decided; // e_x = {v_f, v_t} (f < t)
     4)  if ( deciding a vertex this level )
     5)      P_change = P_change × Pr(v_f);
ve   6)      s_j = true;                    // Remember that the vertex is active
     7)  for each (e_x = {v_f, v_t} in g ) and ( t > f ) do
     8)      EDGE-CONTRACThp(N_j, e_x);     // General Version
     9)      if ( deciding an edge this level )
     10)         P_change = P_change × Pr(e_x);
     11) COMP-UPDATEhp( N_j, P_change);
     12) return N_j;
```

Figure 5.15: CREATE-POS-CHILDhp Method for EMD

Changing component probability results in a greater change for CREATE-NEG-CHILDHP since the probability accumulator P_{change} was not previously required for this method. Strictly speaking it is only required for Model 've', but is used for all models for readability purposes.

```
          CREATE-NEG-CHILDhp(N_i, k):
          1)  Relabel N_i as N_j;                  // N_i no longer needed
e         2)  P_change = 1 - Pr(e_k);
v         3)  P_change = 1 - Pr(v_k);
v         4)  for each ( undecided edge e adjacent to v_k )
v         5)      EDGE-DELETEhp(e);
          6)  P_change = 1;
ve        7)  if kv + ke == k then
ve        8)      P_change = P_change × (1 - Pr(v_kv));
ve        9)      s_j = false;
ve        10) else      // Deciding v_ke with s_i==true
ve        11)     P_change = P_change × (1 - Pr(e_ke));
          12) COMP-UPDATEhp( N_j, P_change);
          13) return N_j;
```

Figure 5.16: CREATE-NEG-CHILDhp for EMD

```
          EDGE-CONTRACThp (N_i, k):          // e_k = {v_f, v_t}
e         1)  Locate partition b_y containing v_t in VI_i; // Set y=B if no partition contains v_t
e         2)  if ( y == B ) then             // No partition contains v_t
e         3)      Add v_t to b_x;
e         4)  else if ( y > 0 ) then         // v_f and v_t in different blocks – merge
e         5)      for each v_a∈b_y do
e         6)          Delete v_a from b_y;
e         7)          Add v_a to b_0;
e         8)      Delete b_y;
v,ve      9)  Create partition b=[v_f, v_t];
v,ve      10) Add b to Part_i;
          11) ADD-COMPONENThp( N_i, k )
          12) return N_j;
```

Figure 5.17: EDGE-CONTRACThp for EMD

Note that the EDGE-DELETEHP method for performability does not differ from the one for REL; hence no pseudo-code is given for EDGE-DELETEHP. Similarly

the COMP-UPDATE$_{HP}$ method is entirely identical to the comparable method for the OBDD-A, COMP-UPDATE$_{P}$, and hence is not given again.

EDGE-CONTRACT$_{HP}$ is modified to call the ADD-COMPONENT$_{HP}$ method. The former method assumes that the partition is sorted, making v_f the first vertex of the first partition. This applies only because we know that edges are undirected.

Components are added much like partitions because they mirror the structure of the partition. This process is shown in Figure 5.18.

```
ADD-COMPONENThp (N_i, k):          // e_k = {v_f, v_t, D}
1)  Locate partition b_y containing v_t in VI_i;   // Set y=B if no partition contains v_t
2)  if ( y == B ) then             // No partition contains v_t
3)      for each ( M∈PI_i ) do     // M contains (v_f,v_x,D_x) ∀v_x∈b_y with x≠f
4)          Add (v_x,v_t,D+D_x+D(v_f)) to M ∀v_x∈b_y with x≠f;
5)  else if ( y > 0 ) then         // v_f and v_t in different partitions – merge
6)      for each ( M∈PI_i ) do
7)          for each v_a∈b_y ( a ≠ t ) do          // With (v_t,v_a,D_a)
8)              for each v_b∈b_0 ( b ≠ f ) do      // With (v_f,v_b,D_b)
9)                  Add (v_a,v_b,D_a+D_b+D+D(v_f)+D(f_t)) to M;
10)     Delete b_y;                // All elements now merged with b_0
11) return N_i;
```

Figure 5.18: ADD-COMPONENThp for EMD

The EDGE-DELETE$_{H}$ method is identical to that for reliability, but is shown again in Figure 5.19 for reference. Note that EDGE-DELETE$_{H}$ does not call ADD-COMPONENT$_{HP}$ since existing partitions aren't modified, and if information is added it takes the form of a single-element partition which does not require component information.

```
EDGE-DELETEh (N_i, k):
1)  if ( no partition contains v_t ) then
2)      Add new empty partition b_B;
3)      Add v_t to b_B;
4)  return N_i;
```

Figure 5.19: EDGE-DELETEh for EMD

5.4.2.2 Success and Failure Testing

Node testing for the EMD OBDD-H algorithm is an extension of that for the general OBDD-H algorithm. The major change is the addition of TEST-COMPONENTS$_{HP}$, making the general version of the NON-TERMINAL-NODE$_{HP}$ shown in Figure 5.20 similar to the performability OBDD-A version.

```
NON-TERMINAL-NODEhp (N_i):
1)  if ( at least k_j members of T_j are in the marked block and available ∀j=0…m ) then
2)      TEST-COMPONENTShp( N_i );
3)      if ( PI_i is non-empty ) then
4)          return 3;                          // a non-terminal node
5)      return 1;                              //a success node
6)  else if ( part_i == { } ) then
7)      return 2;  // a failure node
8)  else
9)      return 3; //a non-terminal node
```

Figure 5.20: NON-TERMINAL-NODEhp for EMD

The pseudo-code of the TEST-COMPONENTSHP method shown in Figure 5.21 is identical to the OBDD-A version. All of the changes are subsumed in several statements of the method. In general, the reaching path and minpath lengths are now read from the OBDD-H component structure instead of the one used for the OBDD-A algorithm. The exact details of these will depend on the implementation chosen.

The differences are that delays are read from the marked block of the matrix chain or linked list. The computation of D (line 1) now considers all delays in unmarked blocks plus the shortest minpath delay. For example if the marked blocks have minpaths with delay 2,5 and 7 and c=2, then a path of delay 4 in the vertex (or linked list element) of an unmarked block could result in a new minpath of delay 4+0+0 < 5 (assuming the delays of the vertex and edge adding to the path are zero), and hence the component is not successful. The method has the suffix 'hp' since, even though the pseudo-code is the same, the implementation will be different from the TEST-COMPONENTSP method for the OBDD-A.

```
TEST-COMPONENTShp (N_i):
1)  Compute D, the smallest path delay to a non-target vertex;
2)  for each M_i^x ∈ PI_i do
3)      for each target group T_x⊆T do
4)          max = 0;
5)          Compute D_x, the c_x^th smallest path delay to vertices in T_x;
6)          if ( D_x > D ) then
7)              break;
8)          else if ( D_x > max ) then
9)              max = D_x;
10)     if ( no D_x was greater than D ) then
11)         Pr( max ) += P_i^x; // Update probability of max
12)         Delete M_i^x from PI_i;          // Delete successful component
```

Figure 5.21: TEST-COMPONENTShp for EMD

The DEL-REDUNDANTHP method, shown in Figure 5.22 is similar to the version for REL, but must also delete redundant information from components (lines 10-11).

```
DEL-REDUNDANThp ( N_i, v_x ):
1)  for each (b_j∈Part_i ) do
2)      if ( v_x∈b_j ) then
3)          Delete v_x from b_j;
4)          if ( b_j is empty ) then
5)              if ( b_j is marked ) then
6)                  Part_i = { };
7)              else
8)                  Delete b_j from Part_i;
9)              break;              // Only present once
10) for each ( M_x∈N_i ) do
11)     Delete information regarding v_x from M_x;
12) return N_i;
```

Figure 5.22: DEL-REDUNDANThp for EMD

5.4.2.3 Node Isomorphism and Merging

While the conditions for Hybrid nodes being isomorphic do not change for the general algorithm, the merging of nodes does. As with the OBDD-A performability case, the merging of components must be considered.

```
MERGE-NODEShp (N_old, N_new):
1)  P_old += P_new;
2)  for each M_new∈PI_new do
3)      for each M_old∈PI_old do
4)          if ( M_new = M_old ) then       // Component exists
5)              P_old += P_new;
6)              break;
7)      if no matching M_old was found then
8)          Add M_new to PI_old;// Add to end of list
9)  return N_old;
```

Figure 5.23: MERGE-NODEShp for EMD

Again the solution, shown in Figure 5.23, is identical to the OBDD-A method since the implementation details are abstracted away by the pseudo-code. Two components for the Hybrid method are equal if each of the blocks of the components are equal; in this case each block is either a matrix or a linked list but equality is well defined for both.

Note that OBDD-H nodes do not contain conditions and the components in a node do not affect isomorphism. For this reason the OBDD-H is expected to process fewer nodes than the equivalent OBDD-A, which has path lengths stored in conditions. On the other hand, because the OBDD-H stores information on the lengths between vertices in each partition, there will be less equality between components and hence more components per node.

5.4.3. Example – Computing EMD using OBDD-Hpv

Consider the undirected network as shown in Figure 5.24. Let s=v_0 and t=v_9 with each vertex having a probability of 0.9 of being active, and all edges perfect. To compute EMD, each edge is assigned a delay as shown on the graph and each vertex a delay of 1.0 apart from s and t. This is identical to the OBDD-A given in Section 5.3.3 but uses the hybrid approach. The full OBDD-H diagram for this network is not shown due to the complexity of the nodes, but all nodes are given in Appendix C, which contains the output of the implementation.

The root node is N_0=[({[v_0]*},{({[0]},1.0)}]; indicating that the source vertex v_0 is connected to itself with delay 0.

On the first loop, N_0 is processed to initially give the partitions {[v_0,v_1,v_2,v_3]*} and {[v_0]*[v_1][v_2][v_3]}. The associated components are ({$\begin{bmatrix} 0 & 1 & 2 & 3 \\ - & - & 3 & 4 \\ - & - & - & 5 \\ - & - & - & - \end{bmatrix}$},0.9) and ({[0]},0.1) respectively. The first row of the first of these matrixes gives the minimum distance of the corresponding vertex to the source. Hence v_0 still has distance zero to a source as expected, and v_1, v_2 and v_3 are varying distances away from the closest source. It is irrelevant that the closest source is v_0.

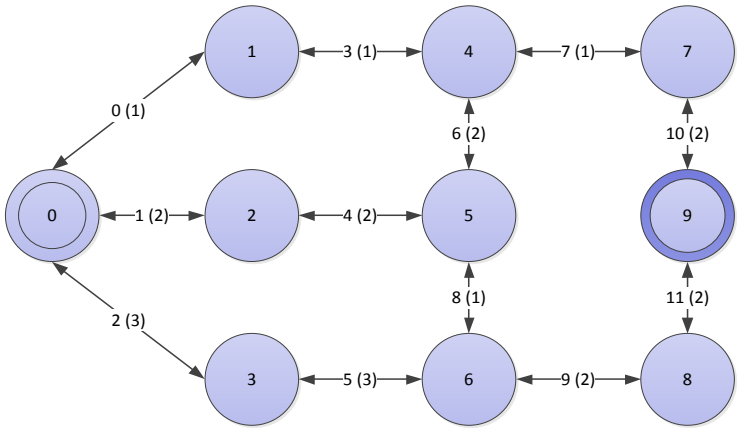


Figure 5.24: Undirected Advanced Network

The distance between v_0 and v_1 is equal to the delay to v_0 plus the delay of v_0 itself plus the delay between v_0 and v_1. The delay to v_0 is taken from the component of the parent node; this indicates a delay of zero. The delay of v_0 is looked up directly, and also found to be zero. Finally the delay between v_0 and v_1, which is the delay of e_0, is looked up and found to be one. Hence the total delay to v_1 is one. Similar processes result in delays of two and three to v_2 and v_3, respectively.

Each other row is the delay between the corresponding vertex and all other vertices. For example the second row is the delay between v_1 and the other vertices in the partitioning. The first and second values in this row correspond to the delay between v_1 and v_0 and the delay between v_1 and itself. The first of these values is not required because it is already present in the matrix and the second is irrelevant.

The next value, 3, is the delay between v_1 and v_2. This is calculated as the sum of the delay between v_0 and v_1 and the delay between v_0 and v_2, plus the delay of the connecting vertex (in this case v_0). If there are multiple different paths between two vertices, the lowest delay is stored. Similarly 4 is the delay between v_1 and v_3 and 5 is the delay between v_2 and v_3.

When the vertex v_0 is deleted, the nodes become N_1 = [{[v_1][v_2][v_3]},{({ },0.1)}] and N_2=[{[v_1,v_2,v_3]*}, {({$\begin{bmatrix}1 & 2 & 3\\ - & 3 & 4\\ - & - & 5\end{bmatrix}$},0.9)}][26]. N_1 is terminal and N_2 is non-terminal, so only N_2 is added to Q_N. Since Q_C is empty, the level is incremented to k=1, the decision variable is now v_1, F_2 is computed to be {v_2,v_3,v_4}, and the queues are swapped.

Node N_2 is removed from Q_C and contracted to give N_6=[{{[v_2,v_3,v_4]*},{({$\begin{bmatrix}2 & 3 & 3\\ - & 5 & 5\\ - & - & 6\end{bmatrix}$},0.81)}] and deleted to give N_5=[{[v_2,v_3]*[v_4]},{({$\begin{bmatrix}2 & 3\\ - & 5\end{bmatrix}$},[]},0.09)}]. The component of N_6 is initially created from $\begin{bmatrix}1 & 2 & 3\\ - & 3 & 4\\ - & - & 5\end{bmatrix}$ and the addition of $e_3 = (v_1,v_4,1)$. Firstly a new column and a new row is created for v_4, giving $\begin{bmatrix}1 & 2 & 3 & -\\ - & 3 & 4 & -\\ - & - & 5 & -\\ - & - & - & -\end{bmatrix}$. In the top position of the new column, the delay between the source and v_1 is one and hence the delay to v_4 is 1 + $D(v_1)$ + $D(e_3)$ = 1 + 1 + 1 = 3. The second value in the column is the delay between v_1 and v_4, which is $D(e_3)$ = 1.

The third field in the column is the delay between v_2 and v_4. The only known connection between these is through the source vertices, and hence the delay uses the delay between v_0 and v_2 as well as the delay between v_0 and v_4. The delay of

[26] Note that the implementation uses a linked list of tuples to represent the matrix to avoid resizing computations. This means that Appendix C give N(2)=[{[1 2 3]*}, 0.90000000, R: C: [0.90000000, { (v0,v1,1) (v0,v2,2) (v0,v3,3) (v1,v2,3) (v1,v3,4) (v2,v3,5) }]. The relationship between the formats is discussed in Section 5.4.1. Note that the tuples (v*x*,v*y*,d) are sorted by *x* and then *y*, instead of being sorted in order of the partitions.

the joining vertex, v_0, is zero, so the required value is the sum of the other two delays. Similarly, the fourth value in the column is computed from the delays between v_0 and v_3 and v_0 and v_4. Hence the matrix becomes $\begin{bmatrix} 1 & 2 & 3 & 3 \\ - & 3 & 4 & 1 \\ - & - & 5 & 5 \\ - & - & - & 6 \end{bmatrix}$. Note that when v_1 is deleted, the entire first column and the second row are deleted since these refer to v_1.

The negative child has no modifications made to the component before v_1 is deleted, other than the modification of the component probability.

The rest of the example will not be given in detail. Instead, certain key steps will be discussed in order to demonstrate the HDD algorithm. Because the components are not easily displayed in a diagram, all processed nodes can be seen in Appendix C.

Consider the positive child of N_{24}=[{[v_4][v_5][v_6]*},{({[7]},0.0081)}]. The decision variable is v_4, so all edges adjacent to this vertex are contracted. Firstly e_6={v_4,v_5,2} is contracted to give the partitioning {[v_4,v_5][v_6]*}; both vertices already exist in separate partitions so those partitions are merged. The component is similarly modified, becoming ({[2],[7]},0.0081) because D(e_6)=2. Note that the component has only a 1×1 matrix for the first partition because it is not marked; if it were marked the matrix would need to include another row for the distance to the source.

Secondly e_7={v_4,v_7,1} is contracted to give partition {[v_4,v_5,v_7][v_6]*} and component ({$\begin{bmatrix} 2 & 1 \\ - & 4 \end{bmatrix}$,[2]},0.0081). The delay between v_4 and v_7 is D(e_7)=1, and this distance is added to the delay between v_4 and v_5 and the delay of v_4 itself (1) to give the distance between v_5 and v_7. After the decision vertex is removed the node becomes N_{50}=[{[v_5,v_7][v_6]*},{({[4],[7]},0.00729)}].

The positive child of N_{27}=[{[v_4]*[v_5][v_6]}, {({[3]},0.0081)}] is N_{56}=[{[v_5,v_7]*[v_6]}, {({$\begin{bmatrix} 6 & 5 \\ - & 4 \end{bmatrix}$},0.00729)}]. The matrix of the component is created from [3] for the partition {[v_4,v_5,v_7]*[v_6]} by adding a row and column for v_5 as well as for new vertex v_7, giving $\begin{bmatrix} 3 & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$. The edges are e_6={v_4,v_5,2} and e_7={v_4,v_7,1}, which add their respective delays into the matrix, giving $\begin{bmatrix} 3 & - & - \\ - & 2 & 1 \\ - & - & 4 \end{bmatrix}$. The delay between a source and v_5 is computed from the delay from the source to v_4, giving 3 + D(v_4)

+ $D(e_6) = 6$. Similarly the delay to v_7 is computed from the delay to v_4, giving 3 + $D(v_4) + D(e_7) = 5$. This gives the matrix $\begin{bmatrix} 3 & 6 & 5 \\ - & 2 & 1 \\ - & - & 4 \end{bmatrix}$. Deleting v_4 gives the final component.

Similarly, the positive child of N_{29}=[{[v_4,v_5]*[v_6]}, {({$\begin{bmatrix} 3 & 5 \\ - & 8 \end{bmatrix}$},0.0729)}] is N_{59}=[{[v_5,v_7]*[v_6]}, {({$\begin{bmatrix} 5 & 5 \\ - & 4 \end{bmatrix}$},0.06561)}]. The component matrix $\begin{bmatrix} 3 & 5 \\ - & 8 \end{bmatrix}$ has a single row and column added for v_7, giving $\begin{bmatrix} 3 & 5 & - \\ - & 8 & - \\ - & - & - \end{bmatrix}$. The edge delays are added in, giving $\begin{bmatrix} 3 & 5 & - \\ - & 2 & 1 \\ - & - & - \end{bmatrix}$. Note that the delay of 8 between v_4 and v_5 is replaced by a delay of 2 as a quicker path is found. The rest of the delays are inferred from these and v_4 is deleted, resulting in the final component shown above.

Both N_{56}=[{[v_5,v_7]*[v_6]}, {({$\begin{bmatrix} 6 & 5 \\ - & 4 \end{bmatrix}$},0.00729)}] and N_{59}=[{[v_5,v_7]*[v_6]}, {({$\begin{bmatrix} 5 & 5 \\ - & 4 \end{bmatrix}$},0.06561)}] have the same partition information, and hence are found to be isomorphic. The newly created N_{59} is merged with the existing N_{56}, and the components are checked for isomorphism. Because the distance between v_5 and the source is different for both components, they are not merged. Hence N_{56}=[{[v_5,v_7]*[v_6]}, {({$\begin{bmatrix} 6 & 5 \\ - & 4 \end{bmatrix}$},0.00729), ({$\begin{bmatrix} 5 & 5 \\ - & 4 \end{bmatrix}$},0.06561)}].

Note that when the components are compared, the OBDD-A is comparing only the delay between the vertices in F_{k+1} and the source(s), while OBDD-H also considers the delays between any other pairs of connected vertices. This implies that OBDD-H components will be identical less often than OBDD-A components. Conversely, OBDD-A nodes include the delays of conditions, which are stored in components for the OBDD-H. Thus it is expected that OBDD-A would have less isomorphism between nodes than the OBDD-H. This is discussed further in Section 5.6.

For example, consider the OBDD-A and OBDD-H versions of N_{106} show below.

N_{106}=[({(v_6,7), (v_7,10)}, 0.006561), ({(v_6,8), (v_7,5)}, 0.006561), ({(v_6,7), (v_7,5)}, 0.6561)}, {(7,6,6)}]

N_{106}=[{[v_6,v_7]*}, {({$\begin{bmatrix} 7 & 10 \\ - & 6 \end{bmatrix}$},0.06561), ({$\begin{bmatrix} 8 & 5 \\ - & 6 \end{bmatrix}$},0.006561), ({$\begin{bmatrix} 7 & 5 \\ - & 6 \end{bmatrix}$},0.64953900), ({$\begin{bmatrix} 7 & 5 \\ - & 12 \end{bmatrix}$},0.06561000)}]

Note that the two components ({$\begin{bmatrix} 7 & 5 \\ - & 6 \end{bmatrix}$},0.64953900) and ({$\begin{bmatrix} 7 & 5 \\ - & 12 \end{bmatrix}$},0.06561000) of the OBDD-H combine to form the single component ({(v_6,7), (v_7,5)}, 0.6561) because the distance between v_6 and v_7 being different doesn't affect the OBDD-A.

The success node of the OBDD-H is N_{1610}=[{[v_9]*}, {({[13]}, 0.18452222). ({[14]}, 0.00047830), ({[17]}, 0.00047830), ({[8]}, 0.59049000)}, which is analogous to the OBDD-A success node and results in the same EMD of 9.21.

5.5. Expected Hop Count

The modified OBDD-A and OBDD-H algorithms introduced in this chapter compute the EMD of a network. While several algorithms in the literature [32-34, 73] claim to compute EMD, they each assume that the delay for all edges is 1 and the delay for all vertices is 0. This problem is more commonly known as the EHC problem, as discussed in Section 2.6.1, which is a more specific case of EMD. For EHC we refer to paths having *length* instead of *delay*, since the metric measures the number of hops.

While the problems are similar, they are not identical. Because the lengths of the EHC problem are known, an algorithm designed specifically to compute EHC can make several optimizations. For example, when extending a reaching path with delay d along edge e_k and vertex v_x, the EMD algorithm computes the new delays as d+D(e_k)+D(v_x) while the EHC algorithm simply increments d by one. This means every such extension replaces the delay lookups of e_k and v_x and their summation with a single increment operation. Thus the processing time per node for EHC is expected to be lower than for EMD.

Another optimization relates to the success test; with EMD a node is not successful while there is a reaching path that is shorter than the current delay of the network state represented by that node. This is because it is possible that such a reaching path could be extended along an edge that potentially has delay 0 to a target (also delay 0), giving a shorter minpath and changing the delay of the network state. For EHC, we know every edge has length 1, and hence a reaching path that is exactly 1 hop shorter than the current length of the network state can be ignored; hence a

component or node that would be non-terminal for EMD may be successful for EHC, reducing the number of components and nodes processed.

When computing EHC with the OBDD-A, any two conditions that have the same number of hops will have the same length, and hence will not prevent nodes from being merged. This is not the case for EMD, where the varying delays between edges and vertices cause less conditions and components to have equal lengths. This means that two nodes that could be isomorphic for EHC may not be isomorphic for EMD. This further increases the differences in the number of nodes and components between EHC and EMD. Note that this difference will increase if the EMD delays vary more widely since this reduces the chance of different paths generating the same delay to a given vertex. Since it has been shown that increased node isomorphism may also increase the number of components per node, some networks may have greater component maximums per node for EHC as compared to EMD.

The impact of these changes on algorithm performance is discussed in Section 5.6.5. Each of the impacts discussed are shown to occur in the simulation.

5.6. Performance Evaluation

5.6.1. Performability Example

Consider a network of WSN devices laid out in a grid as shown in Figure 5.25, with a base station at v_0 and an event occurring at v_{24}. Assume that all WSN devices have a probability of being active of 0.9, and that communications are perfect (Model ‘1v’). The reliability of the network is 0.7867; recall that a large portion of the failure chance is because either the source or target can fail. If the source and target vertices are set to be perfect the reliability of the network becomes 0.9717.

WSN nodes are prone to failure due to their batteries becoming empty [2], and hence it is useful to know not only the chance of successfully detecting an event, but also how many devices an intrusion notification message must pass through; more devices passing on the communication means more overall battery power being used. This is an instance of the EHC problem.

OBDD-Ap computes EHC for this network in 0.061s, processing 2418 nodes with a maximum of 290 nodes per level and at most 9 components in a node. The OBDD-Hp requires 0.055s and 541 nodes with a maximum of 53 nodes per level

and 48 components per node. Note that the OBDD-Hp generates fewer nodes but more components per node. For this problem the number of additional components does not create so much additional processing that the OBDD-Hp is slower than the OBDD-Ap, although for larger networks this is often the case as shown in Section 5.6.2.

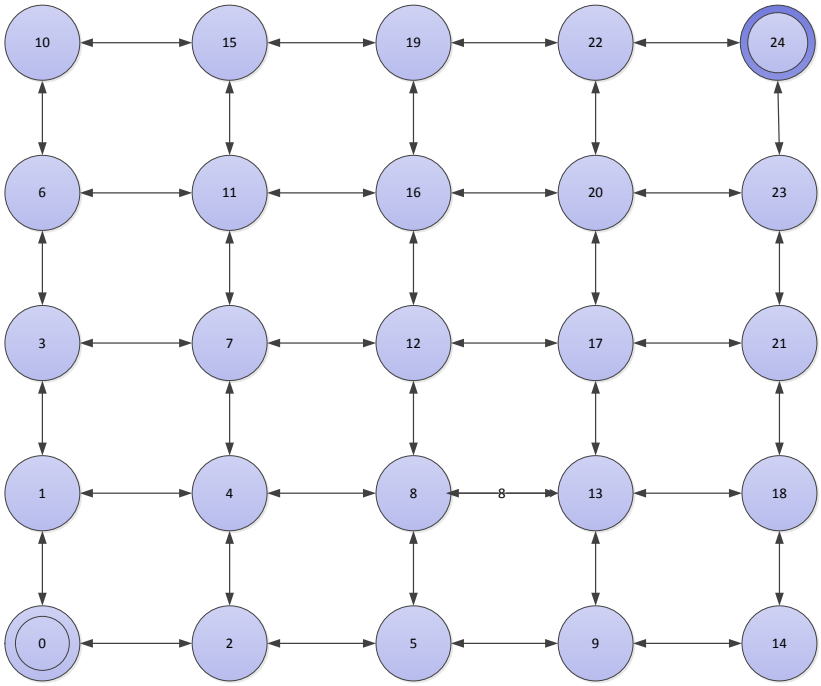


Figure 5.25: Sample WSN

Both algorithms compute the EHC to be 8.0008, which means that due to the high degree of redundancy in the network it is almost certain that only the minimum 8 hops will be required. Indeed, the probability of more than 8 hops being required is 0.0002. This can be computed from the results of the algorithm which gives each Pr(L), the probability of the message requiring L hops, separately for every possible hop count L. For example Pr(14) = 0.00000016 means that while it is possible that the message will require 14 hops, it is extremely unlikely. This is another benefit of the OBDD-Ap and OBDD-Hp algorithms over an algorithm that computes only the EHC.

The problem described above, with each edge having a delay value of 1 and each vertex having a delay value of 0, has been described in the literature [32-34, 73] as message delay. This estimate of delay assumes that all communications take an equal amount of time and that devices introduce no delay at all.

Sensor nodes mainly use the broadcast communication method [2], which means that a WSN device is likely to receive messages from each of its neighbours. This

implies that communication with devices that have more neighbours is likely to be subject to greater delays from packet collision than those with fewer neighbours. For this reason it is more realistic to assign a higher delay to edges that represent communications with interior nodes of the network.

In addition, WSN devices often process received information before sending it, in order to reduce power usage [2]. Assuming that a device receives broadcast information from each of its neighbours it must process more information if it has more neighbours. Hence the device delay should also be increased for devices with more neighbouring WSN devices.

Consider the network as shown in Figure 5.25, as before, but with the delay of each vertex equal to its degree (the number of adjacent edges) and the delay of its edge the sum of the delays of its endpoints, except that the source and target have delay 0. Hence the corner vertices v_{10} and v_{14} have delay 2, side vertices (*e.g.*, v_3 and v_{21}) have delay 3 and interior vertices (*e.g.*, v_7 and v_{13}) have delay 4. Similarly, an edge between two side vertices has delay 6 (3+3) and an edge between a side vertex and an interior vertex has delay 7 (3+4). These delays do not represent an actual message delay, but are a scalable measure of delay caused by broadcast communication.

The OBDD-Ap performs the EMD computation on this network in 0.065s, generating 3355 nodes of which at most 449 are on any one level, and storing at most 61 components per nodes. The OBDD-Hp generates only 541 nodes in 0.047s, with at most 53 per level, but stores up to 469 components per node. The EMD is found to be 62.76, with the most likely delay (60) found to have a probability of 0.3874 and the greatest delay (139) found to be very unlikely (less that 1×10^{-8}). Note that once again the OBDD-Hp is faster than the OBDD-Ap.

It should be noted that existing methods [32-34, 73] are able to compute the solution to the first of these but not the second (*i.e.*, networks with varying vertex/edge delay). Any problem that requires delays other than one per edge and zero per vertex requires the full EMD algorithm, which to the best of our knowledge only OBDD-Ap and OBDD-Hp can solve.

5.6.2. Effect of F_{max} and Communication Models on Efficiency

As with computing REL, the performance of the OBDD-Ap and OBDD-Hp is dependent on the network. This section examines how various attributes of the network (*i.e.*, F_{max} and Communication Models) affect the number of nodes and

components generated, as well as the processing time. Since EMD is a new problem we also discuss the change of EMD as the network changes.

Recall that for REL, the main property of the network that affects the OBDD-Ap and OBDD-Hp is F_{max}; as F_{max} increases the performance decreases sharply, but when F_{max} is constant the processing time increases linearly and the maximum number of nodes stored is constant. For EMD the delays for each edge and vertex must also be considered; widely varying delays decrease the chance of components being equal and hence getting merged.

Section 5.6.2.1 investigates the performance of OBDD-Ap and OBDD-Hp for networks with fixed F_{max} while Section 5.6.2.2 considers the performance as F_{max} changes. For both of these sections, the delays of the networks are set according to the example in Section 5.6.1; the delay of each vertex is equal to its degree and the delay of an edge is equal to the sum of the delays of its endpoints. This delay definition extends in a homogenous fashion as the size of a network increases, and all delays are relatively small.

5.6.2.1 Fixed F_{max}

When F_{max} is fixed, the main variable is the size of the network. This section examines the change in OBDD-Ap and OBDD-Hp performance as well as the change in EMD. In order to be able to test the 've' component failure models, all tests are run on $3\times L$ grid networks.

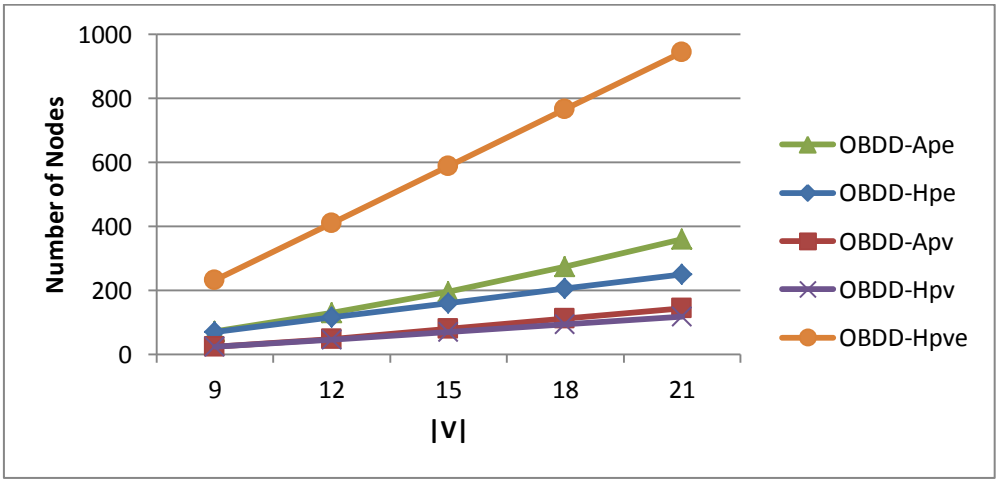


Figure 5.26: Nodes for $3\times L$ Grid with Constant F_{max} and Delay

The number of nodes generated is shown in Figure 5.26 and Figure 5.27; the OBDD-Apve nodes must be shown in a separate graph since the number is much larger than that of the other tests. Note that the number of OBDD-Hp nodes

increase linearly with |V| for all component failure models; this is expected since the number of nodes generated for the performability OBDD-Hp is equal to the number of nodes for the OBDD-Hp computing REL.

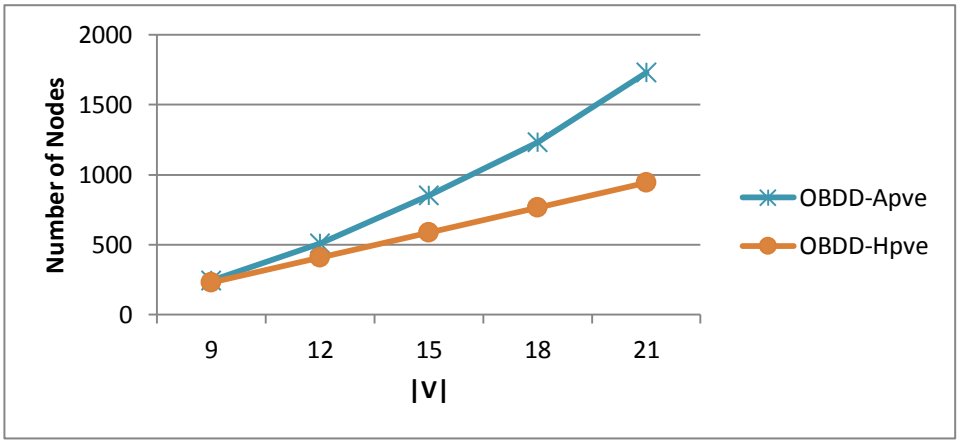


Figure 5.27: Nodes for 3×*L* 've' Grid with Constant F_{max} and Delay

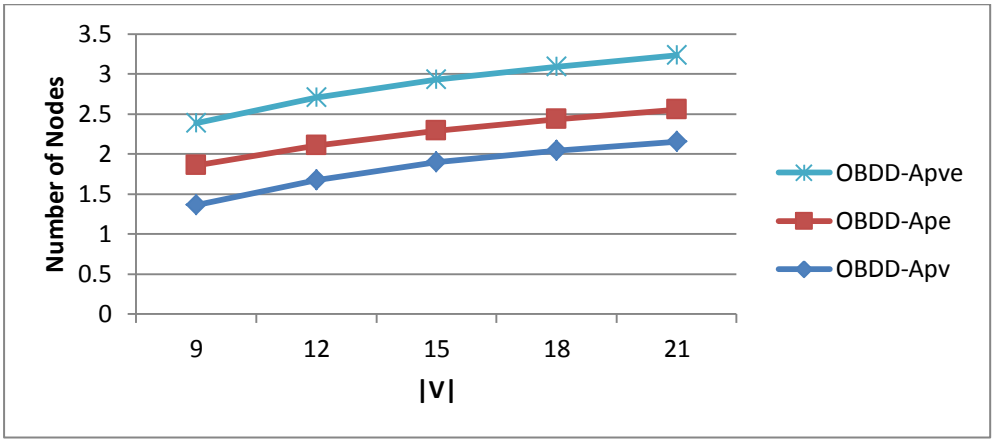


Figure 5.28: Number of OBDD-Ap Nodes (Log_{10})

By contrast, the number of nodes generated for the performability OBDD-Ap is seen to increase more than linearly. This pattern is difficult to see for Models 'e' and 'v' due to the scale, but can be clearly seen for Model 've'. This is further shown in Figure 5.28, which shows the $\log_{10}$ of the number of nodes of the OBDD-Apve. The graphs are not linear, and hence the increase is not entirely exponential.

For REL, the processing time was dependent on only F_{max} and the size of the network, but this is not the case for EMD. Because each node has components which may require processing, the computation time is also dependent on how many components each node has. This change can be seen in Figure 5.29 (OBDD-Ap) and Figure 5.30 (OBDD-Hp) where the processing time increases exponentially or worse with the size of the network.

Note that the processing time for OBDD-Hp quickly becomes much greater than for the OBDD-Ap for Models ‘e’ and ‘ve’ while remaining similar for Model ‘v’. This is largely due to the number of components generated, which is least for Model ‘v’. For small numbers of components, the processing time depends largely on the number of diagram nodes generated, but as the number of components grows they begin to strongly influence performance.

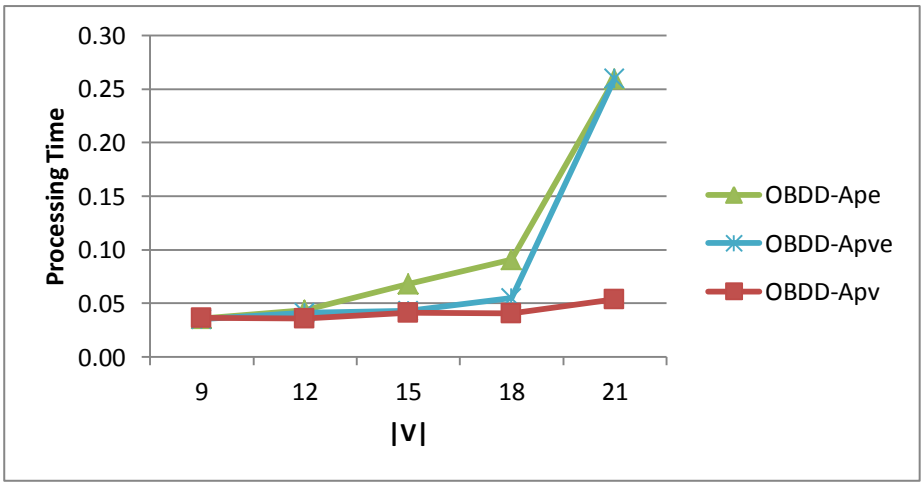


Figure 5.29: Processing Time for OBDD-Ap with Constant F_{max} and Delay

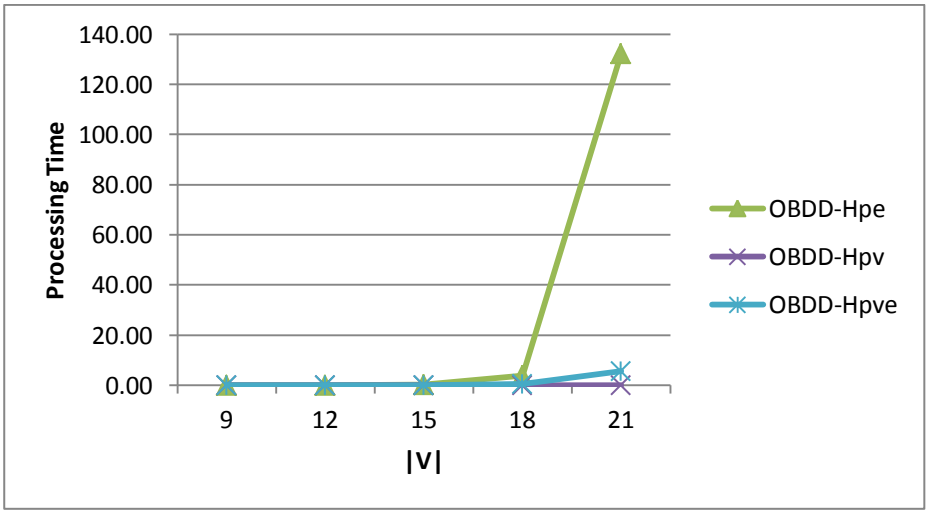


Figure 5.30: Processing Time for OBDD-Hp with Constant F_{max} and Delay

Note that the OBDD-Ape actually requires more processing time than the OBDD-Apve when computing the larger networks (*i.e.*, the 3×7 grid), and the same is true for the OBDD-Hp. This is due to the balance between the number of nodes and components generated. This is discussed further in Section 5.6.3.

The maximum number of components in any node of the diagram is shown in Figure 5.31 (OBDD-Ap) and Figure 5.32 (OBDD-Hp). It can be seen that the

OBDD-Ap has far fewer conditions for the larger networks than the OBDD-Hp; both graphs appear similar but have entirely different scales.

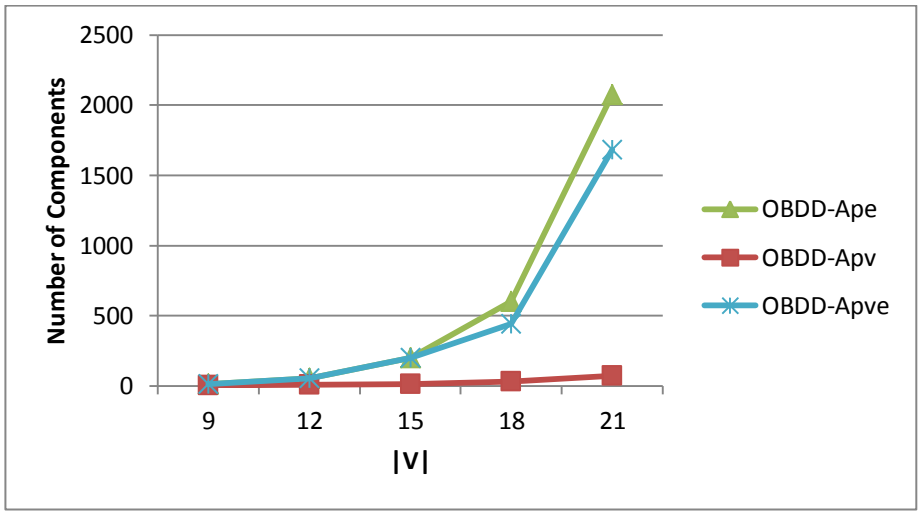


Figure 5.31: Max Components for OBDD-Ap with Constant F_{max} and Delay

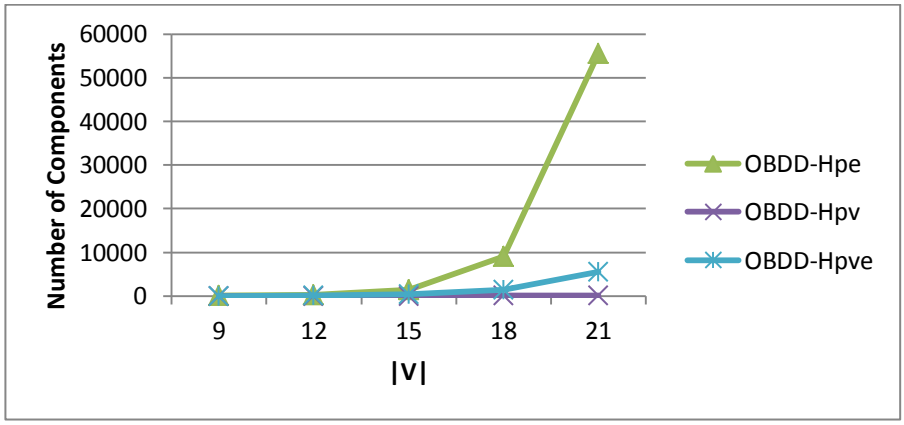


Figure 5.32: Max Components for OBDD-Hp with Constant F_{max} and Delay

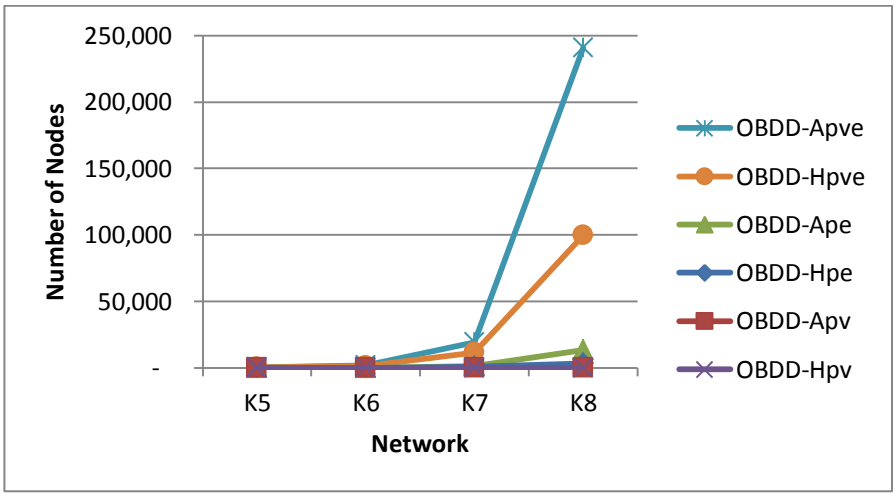


Figure 5.33: Nodes Processed for Fully Connected Networks

With the largest network having far more components to a node for the OBDD-Hp compared to the OBDD-Ap (by roughly a factor of 10 for Models ‘e’ and ‘ve’) the processing time comparison between the algorithms is explained. Despite OBDD-Hp generating far fewer nodes, it has a far larger processing workload per node.

Table 5.2: Performance for Fully Connected Networks

	OBDD-Ape EMD				OBDD-Hpe EMD			
	Time	**Nodes**	**Max N**	**Max C**	**Time**	**Nodes**	**Max N**	**Max C**
K5	0.04	59	9	10	0.03	57	9	18
K6	0.04	263	39	43	0.06	221	31	182
K7	0.07	1,381	216	213	1.28	879	120	3,352
K8	2.50	13,580	2,158	5,299	1002.67	3,589	469	42,142
	OBDD-Apv EMD				**OBDD-Hpv EMD**			
	Time	**Nodes**	**Max N**	**Max C**	**Time**	**Nodes**	**Max N**	**Max C**
K5	0.03	5	1	1	0.03	5	1	1
K6	0.03	6	1	1	0.03	6	1	1
K7	0.03	7	1	1	0.03	7	1	1
K8	0.03	8	1	1	0.03	8	1	1
	OBDD-Apve EMD				**OBDD-Hpve EMD**			
	Time	**Nodes**	**Max N**	**Max C**	**Time**	**Nodes**	**Max N**	**Max C**
K5	0.04	313	60	13	0.04	296	51	24
K6	0.06	2,115	442	38	0.06	1,663	315	166
K7	0.50	19,515	3,340	131	0.59	11,447	2,082	1,880
K8	67.81	240,674	41,382	455	116.36	99,698	14,909	21,754

5.6.2.2 Varying F_{max}

For varying F_{max} the OBDD-Ap and OBDD-Hp are again tested on fully connected networks. Since both algorithms show exponential performance for even fixed F_{max}, the performance for increasing F_{max} is even worse. The increase in the number of nodes is shown in Figure 5.33. It can be seen that while the number of nodes generated is worst for Model ‘ve’ in each case, the performance for Model ‘e’ is also exponential.

The extreme variance in results between the different models makes displaying them in graphs difficult. The full results for the tests are shown in Table 5.2. As with REL, the fully connected networks are trivial for Model ‘v’. For Models ‘e’ and ‘ve’, the OBDD-Hp processes fewer nodes but generates more components than the OBDD-Ap. As with the $3\times L$ grid networks, this causes the OBDD-Ap to have better time performance for large networks as compared to OBDD-Hp.

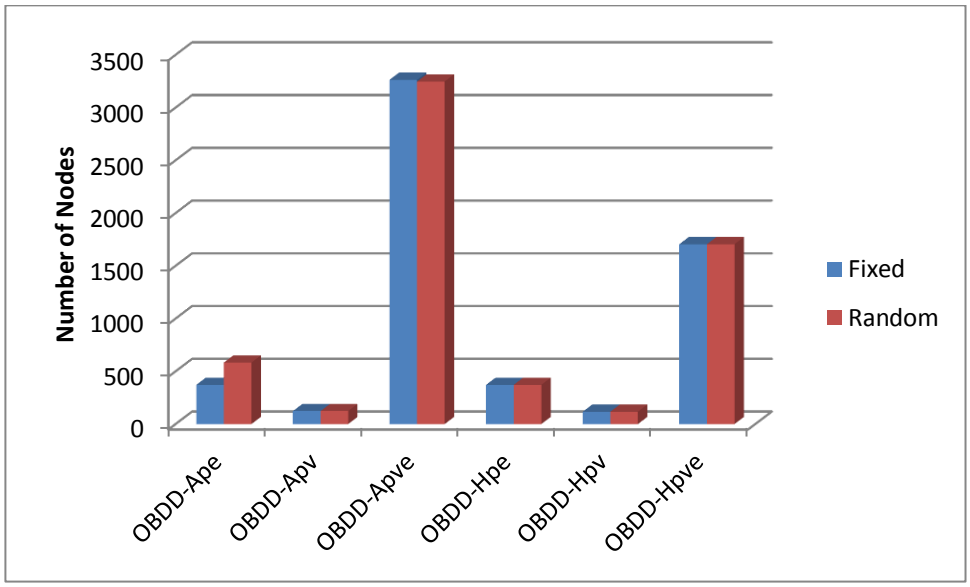


Figure 5.34: Nodes Processed for Fixed and Random Delays

5.6.2.3 Edge and Vertex Delays

The delays of the edges and vertices of a network can affect the performance of the OBDD-Ap and OBDD-Hp. This section compares the 4×4 grid with delays as per the example in Section 5.6.1 to the performance of the 4×4 grid with random delays between 0 and 100 assigned to each edge and vertex.

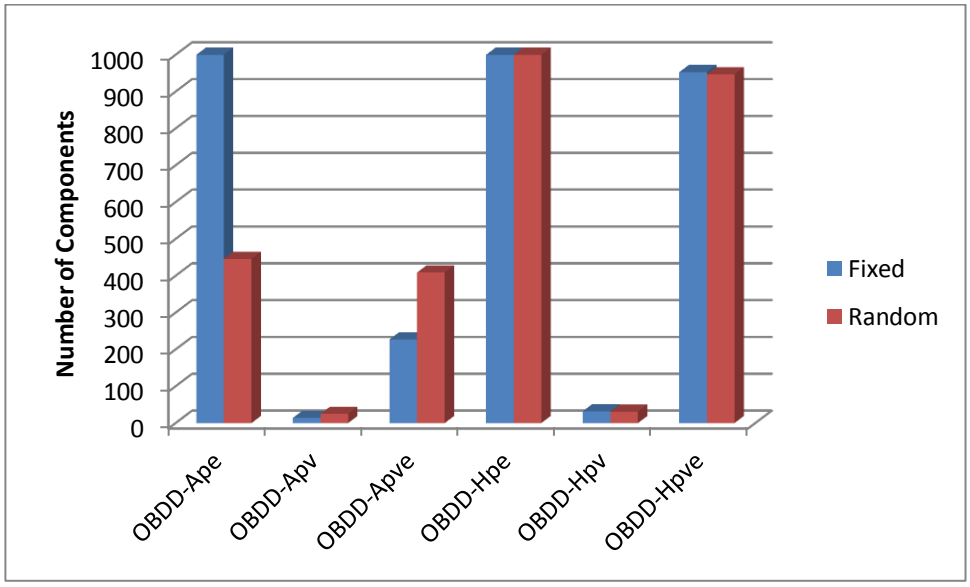


Figure 5.35: Max Components per Node for Fixed and Random Delays

As can be seen from Figure 5.34, the number of nodes processed by each of the algorithms is very similar for the fixed and random delays. This is entirely expected for the OBDD-Hp since the number of nodes processed is not dependent

on delays at all. The minor differences for the OBDD-Ap are that the nodes are actually more for the fixed delay network. For example, the OBDD-Apve processes 3,263 nodes for the fixed delay network and 3,247 nodes for the random delay network.

The maximum number of components per node varies slightly more than the number of nodes, although still not greatly. The graph in Figure 5.35 shows the number of components for each case apart from the OBDD-Hpe, which is 4,193 for the fixed network and 5,472 for the network with random delays; too large to fit the scale of the graph.

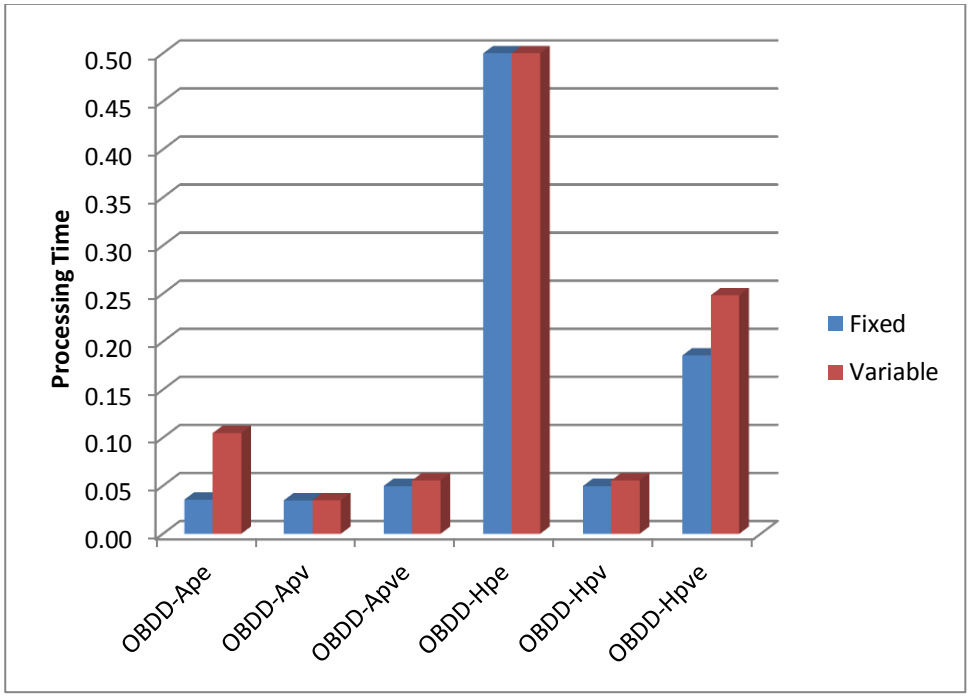


Figure 5.36: Processing Time for Fixed and Random Delays

The processing time shows a similar pattern but more clearly. As shown in Figure 5.36, the processing time for the network with variable delays is noticeably higher for all cases; the OBDD-Hpe requires 2.13s for the fixed network and 3.35s for the random one. Significantly there is no case where the processing time for the fixed network is greater than for the variable one.

While it will generally be the case that a network with a larger variance of delays will take longer to process, this is not always the case. For this reason having a stable set of delays for tests is important. All remaining tests in this chapter follow the delay pattern set out in the example in Section 5.6.1.

5.6.3. Components, Nodes and Performance for EMD

As shown in Section 5.6.2, the number of components for the OBDD-Ae is greater than the number of components for the OBDD-Ave, while the latter has more nodes. This difference is similar to the difference between OBDD-A and OBDD-H. The OBDD-Ae has fewer levels, but also has far fewer nodes per level. Recall that for the OBDD-Ae, a condition is deleted when either one of the endpoints is reached; this reduces the effect of storing delay in each condition.

This is because the varying component failures of the OBDD-Ave present more options in terms of the delays of the conditions associated with each node. The OBDD-Ae has more nodes found isomorphic, compared to the number of nodes processed.

When two nodes are merged, the components of both nodes are merged into the resulting node. In general, if a node is the result of more merge operations it will have the chance of having more components. Thus, while the OBDD-Ave will generate more nodes the OBDD-Ae will have a higher maximum number of components per node.

This issue was verified by a simple experiment. The OBDD-A implementation was amended to not delete a condition when it has been triggered (through the TRIGGERP method shown in Figure 5.2). This does not change the values of REL and EMD computed by the algorithm, but by leaving unnecessary information in nodes it reduces the amount of isomorphism. Doing so increased the number of nodes processed by the OBDD-Ave for the 3×7 grid with random delays from 1684 to 19560 but decreased the maximum number of components per node from 1503 to 522. In this case the increase in the number of nodes overshadowed the decrease in the number of components, causing processing time to increase from 0.1793s to 0.3374s. Clearly the balance between the number of nodes and the number of components strongly affects algorithm performance; this is discussed further in Chapter 6.

A similar argument holds true for the OBDD-He and the OBDD-Hve, except that in this case the number of nodes generated by the two algorithms is identical since delay does not affect isomorphism of the OBDD-H. The difference between the isomorphism definitions of both algorithms is that the OBDD-He immediately merges all blocks with a common vertex while the OBDD-Hve doesn't.

An example of this is computing the 3×5 grid. The OBDD-He processes N_{24}=[{[v_2][v_3][v_4]* }, 0.00810000] C: [0.00810000, { (v_0,v_4,11) }] to give N_{50}=[{[v_2, v_4]*[v_3] }, 0.00729000] C: [0.00729000, { (v_0,v_2,16) (v_0,v_4,11) (v_2,v_4,5) }]. The OBDD-Hve processes the equivalent node N_{234}=[{[v_2][v_3][v_4]* }, 0.00590490] s=1[27] C: [0.00590490, { (v_0,v_4,11) }] to give N_{470}=[{[v_2, v_4][v_3][v_4]* }, 0.00531441] s=1 C: [0.00531441, { (v_0,v_4,11) (v_2,v_4,5) }]. Because vertex v_4 has not yet been decided as active, the OBDD-Hve cannot merge the blocks, while the OBDD-He can due to vertices being perfect.

In this case the OBDD-He node is merged with an isomorphic node, increasing the number of components of that node. The OBDD-Hve node is not merged, and hence increases the number of nodes rather than components. This explains why the OBDD-He has a higher maximum number of components per node. Because component lists are searched repeatedly during the processing of a node, longer lists have a strong impact on algorithm performance. As with the OBDD-A, finding a balance between the number of nodes generated and the number of components would be ideal; this is discussed further in Chapter 6.

5.6.4. Effect of Network Connectivity Model

As with reliability, the OBDD-A and OBDD-H for EMD are affected by the network connectivity model. In general, more target vertices allow more chance of at least one of them being reachable but may affect the number of components. Being required to reach more of these target vertices make the network less likely to be connected and may also increase the delay of paths chosen.

Each network model was tested on the 4×4 grid network whose delays are set up as described in Section 5.6.1; the delay of each vertex other than the source and sinks is equal to its degree and the delay of each edge is equal to the sum of the delays of its endpoints. When additional target vertices are needed, these are chosen from vertices v_{15}, v_{14}, v_{12} and v_9 (in order) as shown in Figure 5.37. For example, if three target vertices are required they will be v_{12}, v_{14} and v_{15}. Each fallible vertex or edge has a probability of 0.9 of being active. Note that the grid shown has a single target vertex; multiple target vertices change the delays of the network as described above. For example if v_{14} is a target, then $D(v_{14})=0$ and hence $D(e_{20})=4$, $D(e_{21})=3$ and $D(e_{23})=0$.

[27] Recall that for the OBDD-Hve (and OBDD-Ave) the Boolean variable s_i tracks whether the last vertex to be decided was available or failed. In this case $s_i = 1$ and hence the vertex was available.

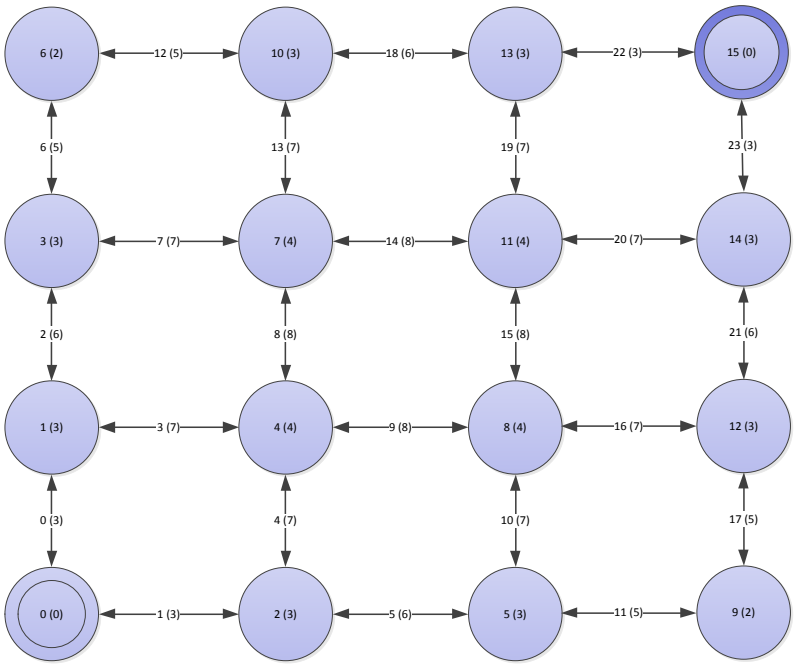


Figure 5.37: 4x4 Grid with Delays

The general patterns shown above continue for the different communication models. The performance of the OBDD-A is better than the OBDD-H, which generates more components and thus requires more processing time. In particular, the performance of the OBDD-He is the worst, greatly exceeding that of the OBDD-Hve. As with Model 1, it is recommended that the OBDD-Hve algorithm be used to solve Model ‘e’, with the probability of vertices set to 1.0.

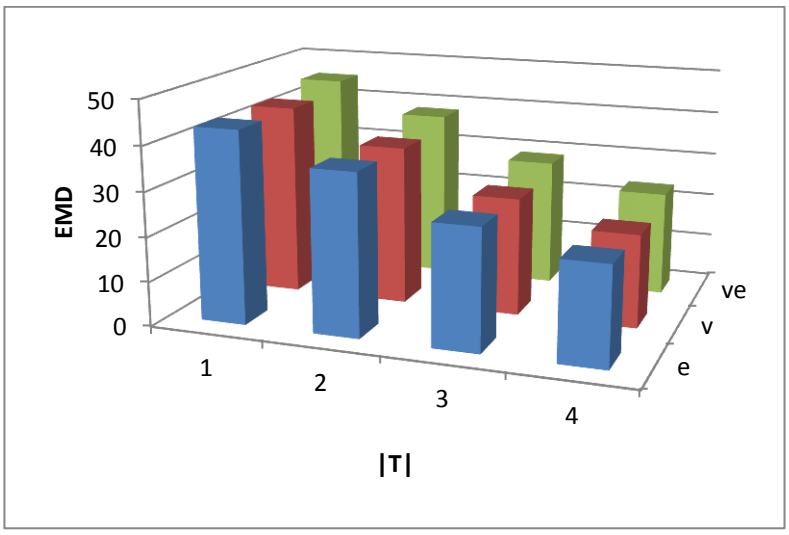


Figure 5.38: EMD for Model 3.1

The change in the EMD is shown for all models. For example, consider the EMD for Model 3.1 as shown in Figure 5.38. It can be seen that the change of EMD is similar for all connectivity models and proceeds in a regular manner.

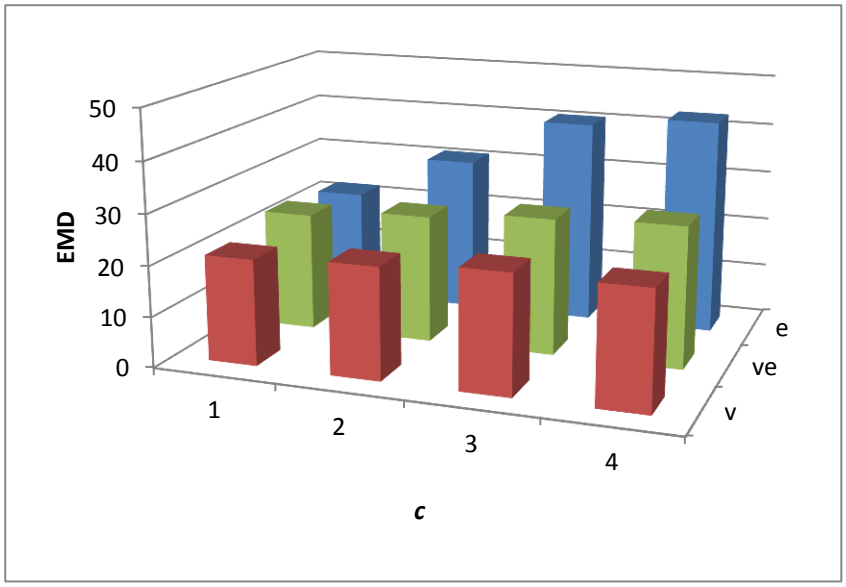


Figure 5.39: EMD for Model 3.2

Similar tests for Model 3.2 show that the EMD increases as *c*, the number of targets needing to be reached, increases. This pattern is somewhat stronger for Model 'e'.

For Model 3.3, the target groups were defined as $T_1=\{v_3,v_6,v_{10}\}$ and $T_2=\{v_{13},v_{14},v_{15}\}$. Due to the smaller target groupings the EMD for $c_i=3$ is the least; recall that the EMD is composed only of successful states and several paths with greater delay cannot be successful when all targets must be reached. This increases the EMD as shown in Figure 5.40.

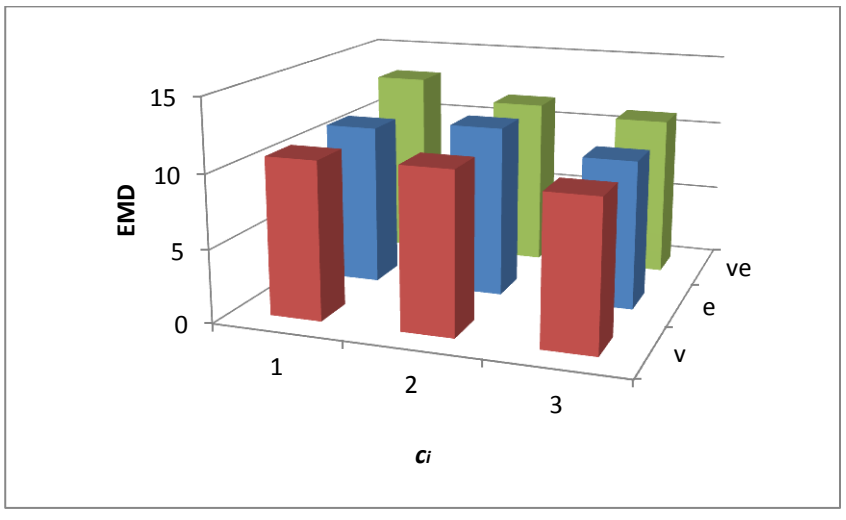


Figure 5.40: EMD for Model 3.3

5.6.5. Expected Hop Count

5.6.5.1 Comparison of EHC and EMD

The OBDD-A and OBDD-H algorithms can compute EHC and it is expected that their performance for EHC is superior to that for EMD as discussed in Section 5.5, at least for larger networks. Both methods make use of largely the same code, so the change in performance for different network and component failure models is identical, but a small number of differences do exist. Each difference is addressed in this section.

The differences in the computation of EHC and EMD lie in the former not requiring memory lookups for the delays on edges and vertices, as well as taking advantage of the fact that a non-reaching path must be extended by at least one unit to reach a target. In contrast, a reaching path for EMD may be extended along an edge of length zero. Hence the EHC implementation is expected to require slightly fewer nodes and slightly less processing time per node.

For Model 1v, nodes are only found successful on the lowest level of the diagram - once the target is decided as available. This means that the number of nodes generated is identical for both EHC and EMD. For example, the 3×100 grid requires 44,350 nodes with a maximum of 296 nodes and 159 components per level for both metrics. Similarly both metrics require the identical 493,975 nodes for the 4×100 grid, with a maximum of 2,621 nodes and 2,255 components per level. Despite nodes and components being identical, the OBDD-A requires slightly less time for EHC (0.5266s and 20.7527s) as compared to EMD (0.5279s and 20.8197s).

For the same network but with the delays fixed as described the example in Section 5.6.1, the OBDD-A also requires 44,350 nodes, with a maximum of 296 nodes per level, to solve Model 1v. However the maximum number of components is 15,719 instead of 159. As a result, the computation requires an average of 248.2715s. The 4×100 grid requires more than an hour of processing time for the fixed delay EMD. It can be seen that processing an EMD problem for even fixed delays can be considerably more involved than the EHC problem for the same network.

For Model ‘e’, components can be found to be successful before the lowest level is reached. This decreases the number of nodes in the diagram, which can in turn increase the number of components per node. For example, the 3×12 grid network requires 424 nodes with a maximum of 559 components per node for EHC, but 613

nodes with a maximum of 322 components for the EMD equivalent. While this is not always the case, this example demonstrates that using the EMD algorithm to compute EHC can be quicker for some networks under Model ‘e’; in this case 0.0425s for EMD compared to 0.0555s for EHC.

For networks with larger F_{max}, such as the 5×5 grid, the additional work performed by the EMD algorithm as compared to the EHC algorithm begins to show. While computing EHC requires 2,552 nodes with at most 537 components per node, EMD requires 3,957 nodes with at most 520 components per node. This leads to the EHC requiring less time; 0.063s compared to 0.0732s.

5.6.5.2 Performance Comparison with Existing Solutions

It is difficult to compare the performance of the OBDD-A and OBDD-H when computing EHC with other solutions. The most recent solutions that do not form part of this work are those by Soh *et al*. [64] and Li *et al*. [34], but neither work gives specific results to compare against. The former gives average results for randomly generated networks in various categories, obtaining good results (0.91s) for networks of 50 vertices for Model 1v. The latter paper [34] only discusses tests on two network families and does not give details of the results, hence comparison with OBDD-A and OBDD-H is impossible.

Comparing the OBDD-A and OBDD-H with the results in [64] is not impossible, but is possibly misleading. It is entirely possible to randomly generate WSN using the BRITE program to randomly generate networks with 50 vertices and 100 edges. Unfortunately the way in which the vertices are connected has a large impact on performance. In addition, the choice of the source and target vertices and the total connectivity of the network (*i.e.*, the network may contain sub-networks that are unreachable even with all edges and vertices available) all affect the performance.

Thus a proper comparison is not possible. However, it is possible to consider some examples. For example, consider a WSN where the sensor nodes are placed in a grid pattern, with each node able to communicate only to its nearest neighbours so that they form a 3×100 network. The OBDD-A and OBDD-H both require 0.53s to compute EHC for this network, despite it containing 300 WSN nodes. We generated a 50 vertex WSN network using the same BRITE settings as described in

[64] and found that it required 0.15s[28] compared to the 0.91s time for that in [64]. Note again that this comparison is not exact since different computers were used and the implementations may have a different focus.

While the results obtained do not form a proper comparison, they at least allow us to state that the OBDD-A and OBDD-H algorithms can compute EHC in time that is at least comparable to that of [64]. Note that the performance of SDP methods is based on the number of paths or cuts of the network, while the performance of OBDD-A is based on the maximum size of the boundary set, F_{max}, and the number of components in a node. This means that one approach may be better for one network while the other may be better for another of similar size but different inter-connectivity.

5.7. Chapter Summary

The OBDD-A and OBDD-H have been shown to be capable of computing performability metrics other than REL through the inclusion of additional, metric-specific, information in each diagram node. This information can grow sufficiently to greatly slow processing time, even when the number of diagram nodes generated is not affected.

The Expected Message Delay (EMD) was introduced and shown to be solvable by both OBDD-A and OBDD-H. While only relatively small networks can be solved for Model 'e', the other models allow EMD to be computed for larger networks. Both algorithms showed different behaviour for EMD as compared to REL, largely relating to the interaction between the number of nodes in the diagram and the number of components per node. This thesis is the first to address the full EMD problem.

The OBDD-A and OBDD-H were shown to also compute the EHC for networks of reasonable size. The OBDD-H generates less diagram nodes than the OBDD-A, but far more components, resulting in greatly increased processing times for larger networks. While a direct comparison is impossible, the OBDD-A and OBDD-H were shown to be at least comparable in performance with existing EHC solutions.

In both cases, the results generated allow for the computation of more metrics than just the EHC/EMD. Metrics such as the probability that the hop count/message

[28] In actuality, five networks were generated. Three of them were found not to have source and target vertices connected, and one had both vertices connected by several paths containing a very small number of hops. The fifth network generated was thus chosen as being more representative.

delay is less than a given amount can be computed without any changes to the algorithm other than the final computations performed on the results. This increases the flexibility of the OBDD-A and OBBD-H as a tool for performability analysis.

The primary concept introduced to allow the OBDD-A and OBDD-H to compute EMD is that of components. Any metric that requires the storage of information other than the node probability requires this mechanism. While the OBDD-A algorithm for REL was shown to exhibit performance based largely on F_{max}, this is not necessarily the case for an augmented algorithm with components.

In particular, increasing the number of components required per node has a marked effect on the processing time of the algorithm. When the information stored is likely to be identical for different reaching paths (such as with EHC) the algorithms remain relatively efficient. However when less information is identical (and hence fewer components are merged) the algorithm's performance decreases. The worst case is when each reaching path has unique information (*e.g.*, if the reaching paths themselves must be stored, as is the case for the maximum bandwidth problem) the OBDD-A algorithms cannot compute answers for even relatively small networks.

Chapter 6
Conclusion and Future Work

6.1. Conclusion

The reliability and performability of computer communication networks are of increasing importance as such networks become more widespread. This thesis has introduced two new formats of the OBDD, the OBDD-A and the OBDD-H, and shown their application to computing the reliability and performability of computer communication networks. Both have been shown to have exceptional memory performance compared to existing solutions as well as competitive processing time complexity. Both algorithms have been shown to have better competitive time performance compared to existing algorithms.

In addition, a number of network connectivity models have been introduced. Both the OBDD-A and OBDD-H are able to compute solutions for most of the models, while existing algorithms generally focus on one model, or at most address two or three. The OBDD-A and OBDD-H are thus very flexible tools, although the OBDD-H can only compute solutions for undirected networks.

Finally, each of the connectivity models has three corresponding component failure models. While both the OBDD-H and OBDD-A can solve all three, the binary versions of these algorithms show poor performance for models with both vertex and edge failure.

In addition to network reliability, it has been shown that both the OBDD-Ap and OBDD-Hp can solve the EHC and EMD performability problems. EHC has been solved using other approaches, but this work is the first to use an OBDD-based approach. EMD has been addressed in the literature, but the proposed solutions only apply to EHC due to restrictions placed on edge and vertex delays. The OBDD-Ap was shown to be competitive with existing approaches while the OBDD-Hp is less efficient for larger networks.

Finally, the solutions produced by the OBDD-Ap and OBDD-Hp are not restricted to only the metrics presented. Because the probability of each event (*e.g.*, the network having a message delay of 6) is tracked separately until the end, more specific metrics can be computed. For example it is possible to compute the

probability that a network has delay less than a given amount or a hop count within a given range.

Thus, the OBDD-A and OBDD-H algorithms have been shown to be flexible but powerful algorithms that are comparable or superior to more specific algorithms. A number of further extensions to these algorithms are possible, although these do not fall within the scope of this thesis.

6.2. Future Work

6.2.1. Heuristics

A number of heuristics have been used in earlier implementations in order to speed up the creation of the diagram. For example, for OBDD-A for Model 'e' and 've' where both endpoints of an edge exist in VS_i, both child nodes will be isomorphic with the parent node and hence vertical isomorphism exists. This can also be applied to Model 'v' on a per-edge instead of per-node basis. This heuristic was included in the implementations described in [21, 45, 104, 106]. Note that this affects the processing time but not the number of nodes processed.

As discussed in Section 3.2.6 the OBDD-A largely ignores vertical isomorphism; that is isomorphism between a parent node and one or more of its child nodes. The same holds for the OBDD-H. Currently the only use of such isomorphism is for the OBDD-Ave (and OBDD-Hve) where nodes on levels deciding an edge for which the last vertex has been decided as failed are automatically detected as isomorphic with both child nodes. This means that instead of generating both child nodes and then merging them, the parent is copied into a single child node without any processing occurring.

As another example; when computing REL(s,t), OBDD-Ae starts with N_0=[({s},1.0),{}]. The next level will always contain a negative child that is either N_1=[({s},0.1),{}] or N_1=[({},0.1),{}] if v_0 becomes redundant immediately, and the positive child will always be N_2=[({v_0,v_1},0.9),{}]. Hence it is possible to use a heuristic to skip the creation of the first level of this diagram and add these nodes directly. Similarly, the negative child on the first level of an OBDD-Av with a single source is always a failure node (N_1=[({},0.1),{}]), and this could be generated using a heuristic.

The benefit of such heuristics is that they reduce the processing required to complete a diagram, both directly and sometimes through reducing the number of

nodes processed. The drawback is that each node must be checked against any heuristics being used, requiring extra processing per node. The heuristics described above are relatively efficient to test for, but this is not necessarily the case. Thus the benefit of using each available heuristic must be investigated to check whether they should be applied, and this can vary depending on the structure of the networks being considered.

This thesis does not use any heuristic except the one for Model 've' as discussed in Section A.3.1. The use of such heuristics may change both the processing speed and number of nodes generated, and thus affects the fairness of the comparison of the performance of the algorithm on the different models. In particular, the choice of which heuristics are used (and the discovery of new heuristics) could bias performance comparison. For an unbiased comparison between the models on the main algorithm, heuristics should hence be avoided.

For research purposes, however, it is useful for the implementation to be as efficient as possible. While the focus of the implementation (and algorithm) given in this thesis are fair comparisons between the different models computed, the focus of the implementations for general research is generally efficiency. In such cases, heuristics should be investigated and used where feasible.

6.2.2. OBDD-H for Directed Networks

The current OBDD-H uses the partitioning model of the boundary set [15, 18] and hence cannot compute REL or performability for directed networks. It has been considered that it would be possible to extend the partitioning definition by allowing directed linkages between mutually-connected partitions.

For example if an existing partition $[v_0,v_1]*[v_2]$ were to be extended through an available directed edge (v_0,v_3), the new partitioning could be recorded as $[v_0,v_1]*[v_2]$ $[v_3](0,2)$, with the pair of vertices at the end indicating that a directed connection exists between partition 0 (containing v_0) and partition 2 (containing v_3).

This approach has not yet been thoroughly studied, although initial work indicates that it is likely to reduce the OBDD-H to the time efficiency of the OBDD-A, or perhaps worse. A detailed study of the feasibility of this approach could show otherwise, but since the OBDD-H has been shown to be less efficient than the OBDD-A for performability such a study has yet to be undertaken.

6.2.3. Multi-valued Diagrams for Vertex Failure

A multi-variate version of the OBDD-A (the Augmented Ordered Multi-Variate Decision Diagram, OMDD-A) has been introduced by the author for the computation of REL and EHC for Model 've' [21, 93]. This algorithm has been extended to the computation of EMD [35] and a similar hybrid version produced (OMDD-H). These algorithms have not been addressed in this thesis for reasons of space, and because they require a number of heuristics in order to function effectively.

The grouping of variables (as discussed in 2.8.3) for the OMDD-A follows from the ordering of variables; each edge is with the endpoint that has the lowest index. For example the edge (v_3,v_7) is grouped together with v_3. This ordering functions well for Models 'e' and 've', but not for 'v'. For Model 'v', this grouping would result in a binary diagram, and hence is not appropriate. Thus a 'good' grouping of vertices must be found if the OMDD-A and OMDD-H are to be applied to Model 'v'.

Possible groupings include cliques [107], constant size groupings and random. The first involves identifying all maximal cliques in the diagram, and using this as the grouping. The advantage of this is that since each vertex in a grouping is connected to each other grouping, a maximum number of edges can be dealt with at one time. Unfortunately finding maximal cliques is itself a NP-Complete problem [107-109], and hence this would add a possibly significant amount of pre-processing.

An alternative is simply to choose a constant, c, and choose group $g_i = \{v_{ic}, \ldots v_{(i+1)c-1}\}$, giving each grouping size c. For example if c=3 then $g_0=\{v_0, v_1, v_2\}$, $g_1=\{v_3,v_4, v_5\}$ and so on. These groupings could also be set to have random size at each level. In any case, choosing a grouping that keeps the vertices in sequential ordering takes advantage of the edge ordering of the network.

6.2.4. Extension to Network Communication Model 5

While the OBDD-A cannot easily be extended to compute REL or EMD for Model 5, doing so for the OBDD-H is feasible. For Models 1 – 4 a block in a partition is marked if that block is connected to a source vertex. For Model 5 it is important to track which source vertex group it is connected to, and possibly how many connections exist. This can be done by replacing the Boolean field MARKED by a list of numbers that indicates which source vertices or groups it is connected to. For

example a block connected to source vertices v_0 and v_2 might be displayed as $[v_4, v_6]^{0,2}$.

This modification would require a change in the definition of node isomorphism; two blocks are equal only if (a) they contain the same vertices and (b) the vertex lists of connected sources (or source groupings) are the same. This would cause fewer nodes to be isomorphic, which may reduce the maximum number of components if computing EMD.

While this modification is theoretically possible, this thesis focuses on Models 1-3, in order to reduce the combinations of different algorithm options under consideration. Currently the implementation of both OBDD-A and OBDD-H algorithms must be tested for three metrics (REL, EMD and EHC), six communication Models (1, 2.1, 2.2, 3.1, 3.2 and 3.3) and three component failure Models ('e', 'v', and 've') for a total of 108 combinations. Adding Model 5 would increase this to 162. For this reason a solution to Model 5 is not addressed in this thesis.

6.2.5. Improvements to Node Failure Testing

The current test for node failure is designed to be extremely quick to compute and to give no false negatives. As mentioned in Section 3.3.3, however, it is possible for a node to have no success nodes in any sub-tree without being detected as a failure node. It would be possible to add additional criteria for node failure, in order to reduce the number of nodes in the diagram.

For example, when computing ALL-REL a node is a failure node if any vertex becomes redundant without having been reached by a minpath. Similarly, for K-REL a node is failed if any of the *K* target vertices become redundant without having been reached. For REL(*s*,*c*-of-T) a node is failed if more than |T|-*c* target vertices become redundant without having been reached.

The implementation in this thesis is designed to be general, to make comparisons between network models as fair as possible. For this reason such optimizations have not been considered. Furthermore the latter two optimizations are more useful for models with relatively high |K| or *c*; this thesis has not performed tests for |K| or *c* larger than 5, except for the special case of ALL-REL.

6.2.6. Investigating Isomorphism

The current definitions of node isomorphism for the OBDD-A and OBDD-H strike a balance between efficient isomorphism checks and the number of nodes declared isomorphic. As shown in Section 5.6.3, on occasions the algorithms will actually have lower processing time if they generate more nodes and fewer components per node.

A good example of this is the comparison between the OBDD-A and OBDD-H for EMD in Section 5.6. The OBDD-H consistently processes far fewer nodes than the OBDD-A, but also has a much higher maximum number of components per node. The processing time performance was shown to be better for the OBDD-A than the OBDD-H.

Experiments can be carried out on the modification of what information in components is considered for isomorphism. No obvious solution has yet been found, and it is possible that this would require an entirely new algorithm, with a node structure distinct from the OBDD-A and OBDD-H. For this reason the work goes beyond the bounds of this thesis.

6.2.7. Metrics for OBDD-A and OBDD-H

Currently, the two algorithms presented in this thesis have been applied to the computation of network reliability, expected hop count, expected message delay and capacity. The first three of these were successful, but the application to capacity [45] was not promising since the nature of the problem required storing information that greatly reduced node isomorphism.

It may be possible to change the information stored in order to compute capacity, or to optimize the computation in another way. Further, the algorithms could possibly be extended for solving other problems such as coverage, fault coverage and common cause failure, as discussed in Section 2.6.3.

Coverage is able to be computed by the algorithms introduced in this thesis. If a region can be sensed by a particular group G of sensor nodes, that group can be defined as a target (or source) grouping. Thus coverage is equivalent to computing the reliability of a network under Model 3.1 (or 4.1), and K-coverage is equivalent to Model 3.2 (or 4.2).

If computing fault coverage instead of reliability, the diagram can compute both the system reliability and the probability that an uncovered fault exists. A multi-

valued diagram can be used to allow each component to have three possible states; available, unavailable but covered and unavailable and uncovered. The reliability is normally stored in each node, but the chance of an uncovered fault must be stored in addition.

For Common Cause Failure (CCF), groups of components that may fail due to a common cause can either be decided as one variable (if they are not likely to fail independently) or can be given a separate level of the diagram. This requires the state of the common causes to be stored in each diagram node, in addition to the information already stored.

For example if vertices v_2, v_4 and v_5 can fail independently or due to common cause they can be grouped together into CCF_1. When the first of these (v_2) is about to be decided the diagram instead inserts a new level for CCF_1. The positive children of this level represent the trigger for CCF_1 not occurring (*i.e.*, the components may fail independently but the common cause has not happened) and the negative children represent CCF_1 occurring. Positive children are treated as normal, with vertices v_2, v_4 and v_5 being able to fail independently based on their component failure probability. For negative children, the probability of these vertices failing is 1; hence on levels where these vertices are decided each such node has only one child which is created with the appropriate CREATE-NEG-CHILD method.

While these metrics are of interest, this thesis focuses on reliability and message delay, and thus they are not addressed.

6.2.8. Parallel Computation

The main loop of the algorithm has a queue of nodes that must be processed to form the next level of the diagram. The algorithm currently processes one node at a time, since it is sequential. The processing of each of these nodes could be done in parallel, however, in order to decrease the processing time required.

Every part of the processing, from the creation of all child nodes to termination testing can be carried out in parallel. The isomorphism testing would be challenging; each parallel process attempting to store a node would have to lock each existing node being compared to, and release that node when the nodes is found to be non-isomorphic or merging has been completed. The parallel algorithm would have to wait until all nodes on one level have been processed and queued before starting the next level.

While there has been work undertaken [110] to parallelize reliability algorithms, the algorithms presented in this thesis have not yet been parallelized. However this is seen as more of an implementation issue that will not affect the main algorithm, and hence it does not form part of this work.

6.2.9. Applications

While this thesis has focused on computer communication networks, a general network model has deliberately been used in order to allow the algorithms presented to apply to other types of networks. Such networks include road transport and power distribution networks. Further work will seek applications of the OBDD-A and OBDD-H to such networks.

Applications in other areas will also be sought. For example existing research by Berber, Kovacevic and Temerinac [111, 112] on digital channel modelling and capacity calculation for WSN is able to compute packet error probability for very small networks. Work is currently in progress on applying the OBDD-A and OBDD-H to computing packet error probability for large networks.

References

[1] P. Baronti, *et al.*, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards," *Computer Communications*, vol. 30, pp. 1655-1695, May 2007.

[2] W. S. Akyildiz, *et al.*, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, pp. 393-422, March 2002.

[3] H. M. AboElFotoh and C. J. Colbourn, "Computing 2-terminal reliability for radio-broadcast networks," *IEEE Transactions on Reliability*, vol. 38, 1989.

[4] S.-Y. Kuo, *et al.*, "Determining Terminal-Pair Reliability Based on Edge Expansion Diagrams using OBDD," *IEEE Transactions on Reliability*, vol. 48, pp. 234-246, 1999.

[5] M. B. Javanbarg, *et al.*, "Reliability Analysis of Infrastructure Networks Using OBDD," presented at the ICOSSAR Osaka, Japan, 2009.

[6] Y.-R. Chang, *et al.*, "A cut-based algorithm for reliability analysis of terminal-pair network using OBDD," in *27th Annual International Computer Software and Applications Conference*, 2003, pp. 368 - 373.

[7] C. Tanguy, "Exact two-terminal reliability of some directed networks," presented at the 6th International Workshop on Design and Reliable Communication Networks, 2007.

[8] S. Rai and S. Soh, "A computer approach for reliability evaluation of telecommunication networks with heterogeneous link-capacities," *IEEE Transactions on Reliability*, vol. 40, pp. 441-451, Oct. 1991.

[9] C. Tanguy. (2006, Exact solutions for the two- and all-terminal reliabilities of the Brecht-Colbourn ladder and the generalized fan. *arXiv:cs/0701005*. Available: http://arxiv.org/PS_cache/cs/pdf/0701/0701005v1.pdf

[10] S. Soh and S. Rai, "CAREL: Computer Aided Reliability Evaluation for Distributed Computing Networks," *IEEE Transactions on Reliability*, vol. 2, pp. 199-213, 1991.

[11] S. Kharbash and W. Wang, "Computing Two-Terminal Reliability in Mobile Ad Hoc Networks," presented at the Wireless Communications and Networking Conference, 2007.

[12] N. K. Goyal, *et al.*, "SNEM: a new approach to evaluate terminal pair reliability of communication networks," *Journal of Quality in Maintenance Engineering*, vol. 11, pp. 239-53, 2005.

[13] Y. Xiao, *et al.*, "An Enhanced Factoring Algorithm for Reliability Evaluation of Wireless Sensor Networks," presented at the 9th International Conference for Young Computer Scientists, Hunan, China, 2008.

[14] S. K. Chaturvedi and K. B. Misra, "A hybrid method to evaluate reliability of complex networks," *International Journal of Quality & Reliability Engineering*, vol. 19, pp. 1098-1012, 2002.

[15] G. Hardy, *et al.*, "K-Terminal Network Reliability Measures With Binary Decision Diagrams," *IEEE Transactions on Reliability*, vol. 56, pp. 506 - 515, Sept. 2007.

[16] J. Carlier and C. Lucet, "A Decomposition Algorithm for Network Reliability Evaluation," *Discrete Applied Mathematics*, vol. 65, pp. 141-156, 1996.

[17] F.-M. Yeh, *et al.*, "OBDD-Based Evaluation of k-Terminal Network Reliability," *IEEE Transactions on Reliability*, vol. 51, pp. 443-451, 2002.

[18] G. Hardy, *et al.*, "Computing all-terminal reliability of stochastic networks with Binary Decision Diagrams," in *11th International Symposium on Applied Stochastic Models*, 2005, pp. 1469-1474.

[19] A. R. Sharafat and O. R. Ma'rouzi, "All-Terminal Network Reliability Using Recursive Truncation Algorithm," *IEEE Transactions on Reliability*, vol. 58, pp. 338 - 347, June 2009.

[20] D. Chen, *et al.*, "Network Survivability Performance Evaluation: A Quantitative Approach with Applications in Wireless Adhoc Networks," 2002.

[21] J. U. Herrmann, *et al.*, "Using Multi-valued Decision Diagrams to Solve the Expected Hop Count Problem," in *IEEE 23rd International Conference on Advanced Information Networking and Applications Workshops (AINA 2009)*, Bradford, UK, 2009, pp. 419-424.

[22] J. U. Herrmann, *et al.*, "On Augmented OBDD and Performability for Sensor Networks," *International Journal of Performability Engineering*, vol. 6, pp. 331-342, Jul. 2010.

[23] H. M. F. AboElFotoh, *et al.*, "On The Reliability of Wireless Sensor Networks," in *IEEE International Conference on Communications*, Istanbul, 2006, pp. 3455-3460.

[24] R. Kirby and R. Schwartz. (2009) Microprocessor-based protective relays *IEEE Industry Applications Magazine*. 43-50. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?reload=true&arnumber=5200923

[25] H.-Y. Lin, *et al.*, "Minimal cutset enumeration and network reliability evaluation by recursive merge and BDD," in *ISCC*, 2003, pp. 1341-1346.

[26] F.-M. Yeh, *et al.*, "Analyzing network reliability with imperfect nodes using OBDD," in *Pacific Rim International Symposium on Dependable Computing*, 2002, pp. 89-96.

[27] S. Soh and S. Rai, "A cutset approach to survivability evaluation of large telecommunication-networks with heterogeneous link-capacities," in *IEEE International Symposium on Circuits and Systems*, 1991, pp. 896-899.

[28] M. O. Ball, "Computational Complexity of Network Reliability Analysis: An Overview," *IEEE Transactions on Reliability*, vol. 35, pp. 230 - 239, 1986 1986.

[29] K. Aggarwal, *et al.*, "A Simple Method for Reliability Evaluation of a Communication System," *IEEE Transactions on Communications [legacy, pre - 1988]*, vol. 23, pp. 563--566, 1975.

[30] F.-M. Yeh and S.-Y. Kuo, "OBDD-Based Network Reliability Calculation," *Electronics Letters*, vol. 33, pp. 759 - 760, Apr. 1997.

[31] X. Zang, *et al.*, "A bdd-based algorithm for reliability graph analysis," Department of Electrical Engineering, Duke University, Technical report2000.

[32] H. M. F. AboElFotoh, *et al.*, "Computing Reliability and Message Delay for Cooperative Wireless Distributed Sensor Networks Subject to Random Failures," *IEEE Transactions on Reliability*, vol. 54, pp. 145-155, 2005.

[33] H. M. F. AboElFotoh, "Algorithms for computing message delay for wireless networks," *Networks*, vol. 29, pp. 117--124, 1997.

[34] Y.-K. Li, *et al.*, "Reliability Computation for Multipath Transmission in Wireless Sensor Networks," in *International Joint Conference on Computational Sciences and Optimization*, Sanya, Hainan 2009, pp. 694 - 697.

[35] J. U. Herrmann, *et al.*, "Computing Performability for Wireless Sensor Networks," *International Journal of Performability Engineering*, vol. 8, pp. 131-140, March 2012.
[36] R. K. Ahuja, *et al.*, *Network Flows: Theory, Algorithms and Applications*. Upper Saddle River, New Jersey: Prentice Hall, 1993.
[37] T. H. Cormen, *et al.*, *Introduction to Algorithms*, 3rd ed.: MIT Press, 2009.
[38] J. Edmonds and R. M. Karp, "Theoretical improvements in the algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, pp. 248-264, 1972.
[39] J. Lestor R. Ford and D. R. Fulkerson, *Flows in Networks*: Princeton University Press, 1962.
[40] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Mathematics Doklady*, vol. 11, pp. 1277-1280, 1970.
[41] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Mathematics Doklady*, vol. 15, pp. 434-437, 1974.
[42] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, pp. 652-686, July 1985.
[43] A. V. Goldberg and S. Rao, "Beyond the flow decomposition barrier," *Journal of the ACM*, vol. 45, pp. 783-797, 1998.
[44] S. Soh and S. Rai, "An efficient cutset approach for evaluating communication-network reliability with heterogeneous link-capacities," *IEEE Transactions on Reliability*, vol. 54, pp. 133--144, 2005.
[45] J. U. Herrmann, *et al.*, "Communication Network Analysis Using Augmented OBDDs," in *9th Postgraduate Electrical Engineering and Computing Symposium (PEECS 2008)*, Perth, Western Australia, 2008, pp. 93-98.
[46] A. Shrestha, *et al.*, "Modeling and Evaluating the Reliability of Wireless Sensor Networks," presented at the Reliability and Maintainability Symposium, 2007.
[47] Y.-F. Xiao, *et al.*, "Reliability evaluation of wireless sensor networks using an enhanced OBDD algorithm," *Journal of China Universities Posts and Telecommunications*, vol. 16, pp. 62-70 Oct 2009.
[48] L. Xing, "Fault-tolerant network reliability and importance analysis using binary decision diagrams," in *Reliability and Maintainability Symposium (RAMS 04)*, 2004, pp. 122--128.
[49] A. Shrestha, *et al.*, "Infrastructure Communication Reliability of Wireless Sensor Networks," in *2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing* 2006, pp. 250-257.
[50] S. A. Doyle and J. B. Dugan, "Dependability assessment using binary decision diagrams (BDDs)," in *25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, USA, 1995, pp. 249-258.
[51] S. A. Doyle, *et al.*, "Combinatorial-models and coverage: a binary decision diagram (BDD) approach," in *Reliability and Maintainability Symposium*, Washington, DC, USA, 1995, pp. 82-89.
[52] Y.-R. Chang, *et al.*, "Reliability evaluation of multi-state systems subject to imperfect coverage using OBDD," in *Pacific Rim International Symposium on Dependable Computing*, 2002, pp. 193 - 200.
[53] Y.-R. Chang, *et al.*, "OBDD-Based Evaluation of Reliability and Importance Measures for Multistate Systems Subject to Imperfect Fault Coverage," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, pp. 336-347, Oct.-Dec. 2005.

[54] L. Xing and J. B. Dugan, "Dependability analysis using multiplevalued decision diagrams," in *6th International Conference on Problems in Safety Assessment and Management*, San Juan, Puerto Rico, 2002.

[55] R. K. Wood, "Factoring Algorithms for Computing K-Terminal Network Reliability," *IEEE Transactions on Reliability,* vol. R-35, Aug. 1986.

[56] K. B. Misra, "An Algorithm for the Reliability Evaluation of Redundant Networks," *IEEE Transactions on Reliability,* vol. R-19, pp. 146 - 151, Nov. 1970.

[57] H. Nakazawa, "Bayesian Decomposition Method for Computing the Reliability of an Oriented Network," *IEEE Transactions on Reliability,* vol. R-25, pp. 77 - 80, June 1976.

[58] D. Bienstock, "Some Lattice-Theoretic Tools for Network Reliability Analysis," *Mathematics of Operations Research,* vol. 13, pp. 467-478, Aug 1988.

[59] R.-H. Jan, *et al.*, "Topological Optimization of a Communication Network Subject to a Reliability Constraint," *IEEE Transactions on Reliability,* vol. 42, pp. 63-70, Mar. 1993.

[60] Z. Deng and C. Singh, "A new approach to reliability evaluation of interconnected power systems including planned outages and frequency calculations," *IEEE Transactions on Power Systems,* vol. 7, pp. 734-743, May 1992 1992.

[61] K. Dohmen, "Inclusion-Exclusion and Network Reliability," *Electronic Journal of Combinatorics,* vol. 5, 1998.

[62] S. K. Chaturvedi and K. B. Misra, "An Efficient Multi-Variable Inversion Algorithm for Reliability Evaluation of Complex Systems Using Path Sets," *International Journal of Reliability, Quality & Safety Engineering,* vol. 9, pp. 237-49, Sept. 2002.

[63] K. D. Heidtmann, "Smaller sums of disjoint products by subproduct inversion " *IEEE Transactions on Reliability,* vol. 38, pp. 305-311, 1989.

[64] S. Soh, *et al.*, "On Computing Reliability and Expected Hop Count of Wireless Communication Networks," *International Journal of Performability Engineering* vol. 3, pp. 267-279, April 2007.

[65] A. Satyanarayana and R. K. Wood, "A Linear-Time Algorithm for Computing K-Terminal Reliability in Series-Parallel networks," *SIAM J. Computing,* vol. 14, pp. 818-832, 1985.

[66] A. Satyanarayana and M. K. Chang, "Network reliability and the factoring theorem," *Networks,* vol. 13, pp. 107 - 120, 1983.

[67] R. K. Wood, "A factoring algorithm using polygon-to-chain reductions for computing K-terminal network reliability," *Networks,* vol. 15, pp. 173 - 190, 1985.

[68] O. R. Theologou and J. G. Carlier, "Factoring and reductions for networks with imperfect vertices," *IEEE Trans. Reliability,* vol. 40, Jun. 1991.

[69] H. Noltemeier and H.-C. Wirth, "Network Design and Improvement," 1999.

[70] J. E. Ramirez-Marquez, *et al.*, "Title," unpublished|.

[71] M. Marseguerra, *et al.*, "Optimal Design of Reliable Network Systems in Presence of Uncertainty," *IEEE Transactions on Reliability,* vol. 54, pp. 243-253, Jun. 2005.

[72] E. S. Elmallah and H. M. F. AboElFotoh, "Circular Layout Cutsets: An Approach for Improving Consecutive Cutset Bounds for Network Reliability," *IEEE Transactions on Reliability,* vol. 55, pp. 602-612, Dec 2006 2006.

[73] V. Li and J. Silvester, "Performance Analysis of Networks with Unreliable Components," *IEEE Transactions on Communications,* vol. 32, pp. 1105 - 1110, Oct. 1984.

[74] R. R. Brooks, *et al.*, "Mobile Network Analysis Using Probabilistic Connectivity Matrices," *IEEE Transactions on Systems, Man, and Cybernetics —PART C: Applications and Reviews,* vol. 37, pp. 1-9, 2007.

[75] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers,* vol. C-35, pp. 677 - 691, Aug. 1986.

[76] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys,* vol. 24, pp. 293-318, 1992.

[77] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal,* vol. XXXVIII, pp. 985–999, 1959.

[78] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, pp. 509 - 516, June 1978

[79] M. Fujita, *et al.*, "Automatic and semi-automatic verification of switch-level circuits with temporal logic and binary decision diagrams," presented at the IEEE International Conference on Computer-Aided Design, Santa Clara, CA 1990.

[80] K. S. Brace, *et al.*, "Efficient implementation of a BDD package," in *27th ACM/IEEE Design Automation Conference*, 1990, pp. 40--45.

[81] R. Drechsler, *et al.*, "Genetic algorithm for variable ordering of OBDDs," *IEE Proceedings on Computers and Digital Techniques,* vol. 143, pp. 364--368, 1996.

[82] B. Bollig and I. Wegener, "Improving the Variable Ordering of OBDDs is NP-Complete," *IEEE Transactions on Computers,* vol. 45, pp. 993-1002, 1996.

[83] F. M. Yeh and S. Y. Kuo, "Variable ordering for ordered binary decision diagrams by a divide-and-conquer approach," *IEE Proceedings Computers and Digital Techniques,* vol. 144, pp. 261--266, 1997.

[84] E. Cerny, *et al.*, "Synthesis of Minimal Binary Decision Trees," *IEEE Transactions on Computers,* vol. C-28, July 1979.

[85] E. Cerny and J. Gecsei, "Simulation of MOS Circuits by Decision Diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 4, pp. 685-693, Oct. 1985.

[86] A. Srinivasan, *et al.*, "Algorithms for discrete function manipulation," in *IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, 1990, pp. 92-95.

[87] T. Y.-K. Kam, "State Minimization of Finite State Machines using Implicit Techniques," Ph.D., University of California at Berkeley, 1996.

[88] P. C. McGeer, *et al.*, "Fast discrete function evaluation using decision diagrams," in *1995 IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, 1995, pp. 402-407.

[89] S. Nagayama and T. Sasao, "On the optimization of heterogeneous MDDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 24, pp. 1645-1659, Nov. 2005.

[90] A. S. Miner and G. Ciardo, "Efficient reachability set generation and storage using decision diagrams," in *20th International Conference on Application and Theory of Petri Nets*, 1999, pp. 6-25.

[91] D. M. Miller and R. Drechsler, "Implementing a multiple-valued decision diagram package," in *28th IEEE International Symposium on Multiple-Valued Logic*, Fukuoka, 1998, pp. 52-57.

[92] E. Zaitseva and V. Levashenko, "Decision Diagrams for Reliability Analysis of Multi-State System," presented at the 3rd International Conference on Dependability of Computer Systems, 2008.

[93] J. Herrmann and S. Soh, "Comparison of Binary and Multi-Variate Hybrid Decision Diagram Algorithms for K-Terminal Reliability," in *34th Australasian Computer Science Conference (ACSC2011)*, Perth, Australia, 2011.

[94] B. Becker, *et al.*, "On the Relation Between BDDs and FDDs," *Information and Computation*, vol. 123, pp. 185--197, December 1995.

[95] B. Becker and R. Drechsler, "How Many Decomposition Types Do We Need?," in *Proceeding of the European Design and Test Conference*, Paris, 1995, pp. 438--443.

[96] R. Drechsler and B. Becker, "Ordered Kronecker functional decision diagrams-a data structure for representation and manipulation of Boolean functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 965-973, 1998.

[97] R. Drechsler and B. Becker, "Dynamic minimization of OKFDDs," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1995, pp. 602-607.

[98] S. Chakravarty, "A characterization of binary decision diagrams," *IEEE Transactions on Computers*, vol. 42, pp. 129--137, 1993.

[99] S. Soh and S. Rai, "Experimental results on preprocessing of path/cut term in the sum of disjoint products technique," *IEEE Transactions on Reliability*, vol. 42, pp. 24 - 33, Mar 1993.

[100] Y.-R. Chang, *et al.*, "Computing system failure frequencies and reliability importance measures using OBDD," *IEEE Transactions on Computers*, vol. 53, pp. 54-68, Jan. 2004.

[101] L. Xing, "An Efficient Binary-Decision-Diagram-Based Approach for Network Reliability and Sensitivity Analysis," *IEEE Transactions on Systems, Man, and Cybernetics —PART A: Systems and Humans*, vol. 38, pp. 105-, Jan. 2008.

[102] J. U. Herrmann and S. Soh, "A Memory Efficient Algorithm for Network Reliability," in *15th Asia-Pacific Conference on Communications (APCC2009)*, 2009.

[103] P. Tittmann, "Partitions and network reliability," *Discrete Applied Mathematics*, vol. 95, pp. 445-453, 30 July 1999.

[104] J. Herrmann, *et al.*, "An OBDD Approach for Computing Expected Hop Count of Communication Networks," in *8th Postgraduate Electrical Engineering and Computing Symposium (PEECS 2007)*, Perth, Australia, 2007, pp. 79-84.

[105] D. Callan. (2007, Cesaro's integral formula for the Bell numbers (corrected). *arXiv:0708.3301v1*. Available: http://arxiv.org/abs/0708.3301

[106] J. Herrmann, "Improving Reliability Calculation with Augmented Binary Decision Diagrams," in *IEEE 24th International Conference on Advanced Information Networking and Applications (AINA 2010)*, Perth, Australia, 2010, pp. 328-333

[107] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, pp. 575-577, Sept. 1973.

[108] Q. Ouyang, *et al.* (1997, Oct) DNA Solution of the Maximal Clique Problem. *Science*. 446-449.

[109] R. M. Karp, "Reducibility Among Combinatorial Problems," in *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, M. Junger, *et al.*, Eds., ed: Springer, 2010, pp. 219-241.

[110] T. Tsuchiya, *et al.*, "Parallelizing SDP (Sum of Disjoint Products) Algorithms for Fast Reliability Analysis," *IEICE Transactions on Information and Systems*, vol. E83-D, pp. 1183-1186, May 2000.

[111] S. Berber, *et al.*, "Digital channel modeling and capacity calculation for multi-relay multi-hop wireless sensor networks," in *5th WSEAS International Conference on Communications and Information Technology* 2011.

[112] S. Berber, *et al.*, "Digital channel modeling for multi-relay wireless sensor networks," in *5th WSEAS International Conference on Communications and Information Technology* 2011.

Appendix A
OBDD-A for Models ‘v’ and ‘ve’

The OBDD-A presented in Chapter 3 only applies to networks with fallible communication links and devices that do not fail. However one of the strengths of the OBDD-A approach is that it allows the computation of network reliability for each of the different component failure models. This appendix presents OBDD-A algorithms for the remaining models and then combines all of the algorithms into a general OBDD-A algorithm that can compute network reliability for any of these models. The OBDD-A referred to in Chapters 4 to 6 is the general one. Section A1 details the OBDD-Av, while A2 details the OBDD-Ave. Both have been implemented, with the results of testing the implementations given in Section 4.8.

A.1. OBDD-A For Vertex Failure

A.1.1. Introduction

When dealing with models that allow vertices to fail, a number of changes must be made to the OBDD-Ae algorithm, described in Section 3.2, to accommodate this, although not every part of the algorithm is affected. The model used to describe OBDD-A nodes does not change, and hence node isomorphism also does not change. The specification of networks and the variable ordering also doesn’t change. The division of nodes into success, failure and non-terminal does not change, but the process of checking for the node type does.

When vertices are fallible and edges are perfect (Models 1v and 4.1v) a number of changes must be made to the algorithm. The first issue is that each level of the diagram decides a vertex instead of an edge, affecting CREATE-POS-CHILD. When a vertex is decided as available, all edges leaving that vertex can be followed; before this decision is made the edges may not be reachable. This also affects conditions; if we have conditions (v_a, v_b) and (v_b, v_c) in a node, it does not necessarily follow that condition (v_a,v_c) exists, since v_b is fallible.

While the possibility of fallible vertices has been addressed in the literature [13, 23, 33, 64], this is generally when addressing the EHC metric as well as REL [33, 64]. However Model ‘v’ is also appropriate for computing REL for wireless networks [23, 33] and in a number of other cases.

Many of the methods from the OBDD-Ae introduced previously are modified for the OBDD-Av. Such methods have the suffix 'v' appended to their names to make it clear which version the method is created for. Methods without the 'v' suffix are identical to those from Section 3.3.1 and are not presented again. Similarly, methods specifically designed for network models with both edge and vertex failure have the 've' suffix appended when they are introduced in later sections.

A.1.2. Conditions and Child Nodes

Conditions in the vertex failure version of the OBDD-A are handled in much the same way as for the edge failure version. When vertices are perfect and a condition (v_f,v_t) is added to node N_i which already has $v_f \in VS_i$, vertex v_t is immediately added to v_f. When vertices are fallible this does not happen, since v_f may be unavailable, preventing the condition from forming a path to v_t.

Similarly, when condition (v_f,v_t) is added to node N_i which has $(v_t,v_x) \in CI_i$ or $(v_a,v_f) \in CI_i$, the extended conditions (v_f,v_x) or (v_a,v_t) are not added to CI_i for Model 'v' for the same reason; until the vertices v_t or v_f respectively are known to be available it is not known whether information can pass through them to form new conditional paths.

The main changes to the algorithm for the creation of child nodes are that ADD-COND (now called ADD-CONDV) must now be called for every edge adjacent to the decision variable, v_k, and that parts that were in ADD-COND for models with perfect vertices have been separated out into the RESOLVEV method called from CREATE-POS-CHILDV.

```
CREATE-POS-CHILDv (N_i, k):        // e_k = (v_f, v_t) or {v_f, v_t}
1)  Create N_{2i+2} as a copy of N_i;
2)  P_{2i+2} = P_{2i+2} × Pr(v_k);
3)  foreach (e_x = (v_k, v_y) or (v_y, v_k) or {v_k, v_y}) and ( y > k ) do
4)      ADD-CONDv(N_{2i+2}, e_x );      // Version for v failure
5)      if ( e_x is undirected ) then
6)          ADD-CONDv(N_{2i+2}, (v_y, v_k) );
7)  if v_k ∈ VS_i then
8)      TRIGGERv(N_{2i+2}, v_k );       // Add vertices to VS
9)  else
10)     RESOLVEv(N_{2i+2}, v_k );       // Combine conditions through v_k
11) return N_{2i+2};
```

Figure A.1: CREATE-POS-CHILDv

The main changes to CREATE-POS-CHILDV, shown in Figure A.1, are the repeated applications of ADD-CONDV to each edge, and the calling of RESOLVEV. The ADD-CONDV method shown in Figure A.2 is simpler than the

corresponding method for edge failure. This is because the expanding of conditions through a vertex has been moved from ADD-COND to RESOLVEv, which is called directly from CREATE-POS-CHILDv (lines 7-10).

```
ADD-CONDv (Ni, (vf, vt)):   // For vertex failure
1)  if ( (vf, vt)∈CIi ) then   // If exists do nothing
2)      return Ni;
3)  if ( vt ∈VSi ) then        // If exists do nothing
4)      return Ni;
5)  Add (vf, vt) to CIi;
6)  return Ni;
```

Figure A.2: ADD-CONDv

The modified TRIGGERv is shown in Figure A.3. This is similar to the TRIGGER method (Figure 3.9, p53), but does not recursively call itself because TRIGGERv is only called on vertices decided as available, which isn't the case for v_x ($x>k$). Also TRIGGERv does not add v_k to VS_i because it is only called when v_k is already present, although it adds each v_x found.

```
TRIGGERv (Ni, vk):
1)  foreach C = (vk, vx)∈CIi do
2)      Add vx to VSi;
3)      Delete C from CIi;
4)      foreach (C2 = (vf, vx)∈CIi ) do
5)          Delete C2 from CIi;
6)  return Ni;
```

Figure A.3: TRIGGERv

The new method, RESOLVEv, is shown in Figure A.4; this matches up pairs of conditions with either endpoint of the decision variable and forms them into new conditions passing through this variable. This preserves conditional paths through v_k when it becomes redundant. In the OBDD-Ae, this is done directly in ADD-COND for each new condition, but for Model 'v' this matching is only done when the vertex is decided as available.

```
RESOLVEv (Ni, vk):
1)  foreach Cf = (vf, vk) ∈CIi do
2)      foreach Ct = (vk, vf) ∈CIi with f≠t do   // Avoid self loops
3)          if (vf, vt) ∉ CIi then
4)              Add (vf, vt) to CIi;
5)  return Ni;
```

Figure A.4: RESOLVEv

If TRIGGERv is needed ($v_k \in VS_i$) then RESOLVEv is not needed because all conditions leading to v_k will already have been deleted; conversely when $v_k \notin VS_i$, all paths leading to and from v_k are conditional and thus RESOLVEv is needed

instead of TRIGGERv. Hence CREATE-POS-CHILDv calls only one of these two methods, but not both.

The CREATE-NEG-CHILDv method shown in Figure A.5 requires only one change compared to the version for Model 'e'; the probability of the negative child is modified by multiplying by the failure probability of a vertex instead of an edge.

```
CREATE-NEG-CHILDv (N_i, k):
1) Relabel N_i as N_{2i+1};        // N_i no longer needed
2) P_{2i+1} = P_i × (1 - Pr(v_k));
3) return N_{2i+1};
```

Figure A.5: CREATE-NEG-CHILDv

A.1.3. The OBDD-Av Algorithm

For Model 'v' it is not necessary to detect when vertices become redundant because exactly one vertex is decided at every level. Once the vertex is decided, it is redundant and all information regarding it can be deleted using DEL-REDUNDANT. Hence CHECK-REDUNDANT is itself redundant for the OBDD-Av.

Note that if target vertices are not being deleted, then the OBDD-Av algorithm must check for this manually. Previously CHECK-REDUNDANT tested for this case, and this test must now be moved to either DEL-REDUNDANT or the body of the main algorithm. Since vertices can fail, however, we do want to delete a target when it has failed, but not delete it when it has succeeded. Since this is implementation dependent, as discussed in Section 3.3.2, this check is not reflected in the pseudo-code.

The TEST-NODE method from Model 'e' requires only a minor change. Because vertices may fail, a node cannot be labelled successful merely by reaching the target vertex; the target vertex itself must also be available. The TEST-NODEv method is shown in Figure A.6.

```
TEST-NODEv (N_i):
1) if ( k≥t and v_t ∈ VS_i ) then
2)       REL += Prob( N_i);
3)       return 1;  // a success node
4) else if ( VS_i == { } ) then
5)       return 2;  // a failure node
6) else
7)       return 3; //a non-terminal node
```

Figure A.6: TEST-NODEv

Note that the test to see whether the target vertex reached has already been decided means that a node that has reached an undecided target would be declared non-terminal. The node and its children will continue to be processed until the level $k=t$ is reached, causing unnecessary processing

A better way is to immediately decide the target vertex[29]; doing so require little additional processing since the negative child must be failed and the positive child must be a success. Hence the probability of the node is multiplied by the probability that the target vertex is active (this probability is 0.9 for this work) and the resulting probability can then be added to REL as normal.

While this method work well for models with a single target (such as Models 1v and 4.1v), it becomes complicated for models with multiple targets. For such models, the chance that another target is reachable must also be considered. Since the implementation for this thesis is designed for accurate comparison between different models, this heuristic is not used. Heuristics are discussed in Section 6.2.1.

The OBDD-Av algorithm is shown in Figure A.7. As can be seen it is mostly identical to that for edge failure, except that the 'v' versions of CREATE-POS-CHILD and CREATE-NEG-CHILD are called and that there are no calls to CHECK-REDUNDANT and associated checks of **REDF** and **REDT**.

```
OBDD-Av (G):                    // Algorithm for Models 1v and 4.1v
1)  Initialize the root node N0;
2)  Initialize k=0, QC = { N0 }, QN = { }, reliability = 0;
3)  Remove the first node, Ni, from QC;
4)  N2i+2 = CREATE-POS-CHILDv(Ni, k );
5)  N2i+1 = CREATE-NEG-CHILDv(Ni, k );
6)  for each child node Nfi do
7)      DEL-REDUNDANT( Nfi,vk );
8)      if ( TEST-NODE(Nfi) == 3 ) then
9)          Check each node on QN for isomorphism with Nfi;
10)         if ( an isomorphic node Nx was found ) then
11)             Merge Nfi into Nx;
12)         else
13)             Add Nfi onto QN;
14) if ( QC == { } ) then
15)     if ( QN == { } ) then
16)         return reliability;
17)     else
18)         k++;
19)         Swap QC and QN;
20) goto 3)
```

Figure A.7: OBDD-Av Algorithm

[29] Strictly speaking, this means that the OBDD-A is no longer an *ordered* diagram, since the target vertex may be decided at almost any level. However since the resulting nodes are never included in the diagram itself it may be considered that they are simply being transferred directly to the lowest level of the diagram and then decided.

A.1.4. Example

For the first example, again consider the network shown in Figure A.8 with $s = v_0$, $t = v_3$, each edge infallible and each vertex having a 0.9 chance of being available. This is an example of Model 1e.

The resulting OBDD-Av is shown in Figure A.9. It can be seen that it has four levels that include non-terminal nodes, as compared to five levels for the corresponding OBDD-Ae. This is because there are only four decisions variable (vertices) for the OBDD-Av as compared to the five decision variable (edges) for the OBDD-Ae.

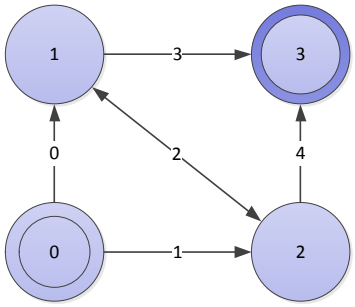


Figure A.8: Simple Network

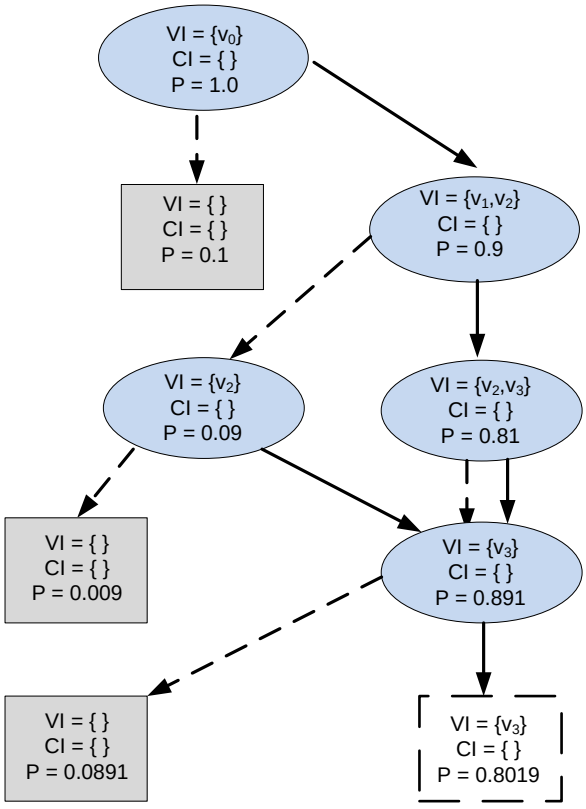


Figure A.9: OBDD-Av for Simple Network

The root node is N_0={ ({v_0}, 1.0), { } }, as given in Section 3.4, with S = {v_0}. Note that this is identical to the root node for the OBDD-Ae because the information stored at this stage is identical; in both cases network traffic is known to be able to reach the source vertex with a probability of 1.0. Again, the other initializations set reliability=0, Q_C = { N_0 }, Q_N = { } and k = 0, however in this case the latter indicates that the decision variable is v_0. Since this example considers a network under Model 'v', the CHECK-REDUNDANT method is not called; it is known that the decision vertex v_0 will become redundant so there is no need to test for it.

The main body of the loop (lines 3 to 13) begins and the root node, N_0 is removed from Q_C, which becomes empty. N_0={ ({v_0}, 1.0), { } } is copied to N_2 and P_2 is multiplied by 0.9, giving N_2={ ({v_0}, 0.9), { } }. Each edge adjacent to v_0 (*i.e.*, e_0=(v_0,v_1) and e_1=(v_0,v_2)) is then dealt with in turn via a call to ADD-CONDV. First e_0 adds the condition (v_0,v_1) to CI_2 and then e_1 adds the condition (v_0,v_2) to CI_2. Hence N_2={ ({v_0}, 0.9), {(v_0,v_1), (v_0,v_2)} }. Since $v_0 \in VS_2$ the TRIGGERV method is called instead of the RESOLVEV method.

TRIGGERV is called on N_2 and v_0. For each condition (v_0,v_t) in CI_2, TRIGGERV will add v_t to VS_2. Since CI_2={(v_0,v_1), (v_0,v_2)} this results in VS_2={v_0,v_1,v_2}. Hence N_2={ ({v_0,v_1,v_2}, 0.9), {(v_0,v_1), (v_0,v_2)} }.

The negative child, N_1, is created by renaming N_0 and multiplying the node probability by the chance that v_0 is failed (0.1). Hence N_1={ ({v_0}, 0.1), { } }. Both children are now passed to DEL-REDUNDANT with a second argument of v_0. Note that we're calling the original version of DEL-REDUNDANT, not one modified for vertex failure as can be seen from the lack of the 'v' suffix. All information on v_0 is removed from both child nodes giving N_1={ ({ }, 0.1), { } } and N_2={ ({v_1,v_2}, 0.9), { } }.

The TEST-NODEV method determines that N_1 is failed, and that N_2 is non-terminal. Hence N_1 is discarded and N_2 is added to Q_N. Hence Q_C = { } and Q_N = { N_2 }. Since Q_C is empty, the queues are swapped and k is incremented to 1. The second level of the diagram decides vertex v_1. Execution returns to the top of the loop (line 3).

Starting the main body of the loop again, N_2={ ({v_1,v_2}, 0.9), { } } is removed from Q_C, which becomes empty. N_2 is processed as above to create children N_5={ ({v_2}, 0.09), { } } and N_6={ ({v_2,v_3}, 0.81), { } }. Once again the positive child has

the conditions deleted after they have been used to update VS. TEST-NODEv finds that both N_5 and N_6 are non-terminal; hence both are added to Q_N. While N_6 has reached v_3, it is not yet known whether v_3 is itself available and hence the node is unfinished.

We now have Q_C={ } and Q_N={ N_5,N_6 } as the main body of the loop completes. Hence the next level of the OBDD-Av is started, with *k* being incremented to 2 and the queues swapped. This level decides the last non-target vertex, v_2.

The first vertex, N_5={ ({v_2}, 0.09), { } }, is removed from Q_C and processed to give N_{11}={ ({}, 0.009), { } } and N_{12}={ ({v_3}, 0.081), { } }. The negative child is a failure node and is discarded, while the positive is non-terminal Similarly N_6={ ({v_2,v_3}, 0.81), { } } is processed to give N_{13}={ ({v_3}, 0.081), { } } and N_{14}={ ({v_3}, 0.729), { } }. Both are non-terminal and isomorphic with N_{12}, and are thus merged to give N_{12}={ ({v_3}, 0.891), { } }.

The level *k*=2 is completed and *k* is incremented to three. We now have Q_C={ N_{12} } and Q_N={ }. The only node, N_{12}, is removed from Q_C and processed to give N_{25}={ ({ }, 0.0891), { } } and N_{25}={ ({v_3}, 0.8091), { } }. The former is a failure node and the latter is a success node. The success node is processed to give *reliability* = 0.8019. This reliability is lower than for the edge-failure model because allowing the source and target vertices to fail introduces two points where the connectivity will fail with a vertex failing. For this reason it is not uncommon to assume that source and/or target vertices are perfect (*i.e.*, Prob = 1.0).

A.2. OBDD-A For Vertex and Edge Failure

A.2.1. Introduction

When both vertices and edges are fallible (*i.e.*, when solving the Model 've') many of the changes to the OBDD-A from Section A.1 above hold; since it cannot be assumed that vertices are available conditions cannot be followed until it is known that the first endpoint is available. While this indicates that implementing Model 've' would mean deciding a vertex after all adjacent edges are decided, this is not the most efficient way.

When a vertex fails, the availability of the adjacent edges does not change the state of the network. Hence, when deciding an edge adjacent to a failed vertex, both child nodes will be identical and will be isomorphic. Because this is known before they are generated, it isn't necessary to actually generate them; the parent node can

simply be moved from Q_C to Q_N to give the merged child node, without any further processing other than checks for redundancy and isomorphism. Note that since these checks must still occur, the process still counts as processing the parent node and creating the child node for the purpose of counting the number of nodes in an OBDD-A diagram. This is a heuristic, and it would be possible to create the child nodes through processing. However this thesis uses this heuristic for the implementation for all algorithms dealing with Model 've'.

The disadvantage of deciding a vertex before the adjacent edges is that each node must contain information on the state of this vertex until the next vertex is decided. For example, if v_1 in the simple network (Figure 3.15 on p.58) is decided, then the implementation must know whether it is active or inactive when deciding edges $e_2=\{v_1,v_2\}$ and $e_3=(v_1,v_3)$. Currently the state of previously decided components is not explicitly stored in the table; if vertices are decided before edges that must change. This work uses the vertex-before-edge form of the algorithm due to the decrease in the number of nodes that are fully processed.

The decision variable *k* used by the OBDD-A is used as the subscript for e_k and v_k for Models 'e' and 'v' respectively. However for Model 've', *k* is not a suitable subscript for either vertices or edges being decided. Clearly some mechanism must be introduced to track which vertex or edge is currently being decided, and hence which will be decided next.

This indicates that it is necessary to track the decision vertex and edge separately using two new variables for the algorithm, *ke* and *kv*, both initialized to 0. Variable *ke* is incremented by one when an edge has just been decided, but when a vertex has been decided *kv* is not incremented; *kv* is only incremented when the decided vertex has become redundant. Because of this, when $ke+kv=k$ the algorithm is deciding a vertex and when $ke+kv<k$ the algorithm is deciding an edge. The algorithm also needs to know when it is deciding the last edge for this vertex; this occurs when the vertex that has just been decided becomes redundant (see Section A.3.4 below). Both *ke* and *kv* are assumed to be either global variables or inheritable from the parent function; they are not explicitly passed to called methods.

A.2.2. The Mathematical Model and Node Isomorphism

The mathematical model of the OBDD-A node changes slightly for the vertex-before-edge Model 've'. Recall that each OBDD-A node, $N_i \in N$, is defined as an

information pair [VI_i, CI_i]. For the ve models this changes to N_i = pair [VI_i, CI_i, s_i] where s_i is a Boolean that is set to **TRUE** in nodes when the last vertex decided is available and **FALSE** otherwise. The state of s_i determines how child nodes are generated on levels where edges are being decided.

Because the model of a node is changed, isomorphism must be reconsidered. Because the state of s_i determines the sub-tree of a node, two isomorphic nodes must have identical s_i. Hence the definition becomes "Two nodes N_i and N_j at the same level of an OBDD-A are *isomorphic* if $VS_i = VS_j$, $CI_i = CI_j$ and $s_i = s_j$".

Adding a requirement to isomorphism will decrease the number of nodes found to be isomorphic per level, however this change allows a limited form of isomorphism between diagram levels. It does not allow testing for isomorphism between unrelated nodes on different levels however; since parent nodes are deleted this is still irrelevant.

A.2.3. Conditions and Child Nodes

The CREATE-POS-CHILD method for Model 've' (CREATE-POS-CHILDVE, shown in Figure A.10) draws mainly on the methods from the OBDD-A for Model 'v'. When an edge is being decided, the ADD-CONDV from Model 'v' (see Figure A.2) is called but when a vertex is being decided the only part of the node modified is s. The TRIGGERV and RESOLVEV methods (see Figure A.3 and Figure A.4) are only called once all edges adjacent to the current vertex have been called, and don't change from the versions used in Model 'v'.

Rather than checking for redundancy in CREATE-POS-CHILDVE (*i.e.*, performing the check for every node being processed on the level), the check occurs once per level, when the level is updated. The CHECK-REDUNDANTVE method (see Figure A.12) sets *redf* and *redt* to be **TRUE** when v_f or v_t of edge (v_f,v_t) are redundant. Note that only the endpoint that has most recently been decided (v_{kv}) can be redundant.

```
CREATE-POS-CHILDve (N_i, k):
1)  Create N_{2i+2} as a copy of N_i;
2)  if kv + ke == k then        // Deciding a vertex
3)      P_{2i+2} = P_i × Pr(v_kv);
4)      s_{2i+2} = true;        // Mark the vertex as active
5)  else                        // e_ke = (v_f, v_t) or {v_f, v_t}
6)      P_{2i+2} = P_i × Pr(e_ke);
7)      ADD-CONDv(N_{2i+2}, (v_f, v_t) );        // v version
8)      if ( e_ke is undirected ) then
9)          ADD-CONDv(N_{2i+2}, (v_t, v_f) );
10) if (redf) then // v_kv redundant
11)     if v_kv∈VS_i then       // s must be true or this wouldn't be called
12)         TRIGGERv(N_{2i+2}, v_kv ); // Add vertices to VS
13)     else
14)         RESOLVEv(N_{2i+2}, v_kv ); // Combine conditions through v_kv
15) return N_{2i+2};
```

Figure A.10: CREATE-POS-CHILDve

The CREATE-NEG-CHILDVE method shown in Figure A.11 is similar to the CREATE-POS-CHILDVE method, in that its action varies depending on whether a vertex of edge is being decided. Note that when CREATE-NEG-CHILDVE is called on a level when an edge is being decided, s_i must be **TRUE**; if it were **FALSE** no calls to the CREATE-POS-CHILDVE and CREATE-NEG-CHILDVE methods are necessary to relabel the parent node. If CREATE-NEG-CHILDVE is called on a level deciding a vertex, then s_i must be set to **FALSE**.

```
CREATE-NEG-CHILDve (N_i, k):
1)  Relabel N_i as N_{2i+1};      // N_i no longer needed
2)  if kv + ke == k then          // Deciding a vertex
3)      P_{2i+1} = P_i × (1 - Pr(v_kv));
4)      s_{2i+1} = false;
5)  else        // Deciding v_ke with s_i==true
6)      P_{2i+1} = P_i × (1 - Pr(e_ke));
7)  if (redf) then // v_kv redundant
8)      if v_kv∈VS_i and s_{2i+1}==true then
9)          TRIGGERv(N_{2i+2}, v_kv );  // Add vertices to VS
10)     else
11)         RESOLVEv(N_{2i+2}, v_kv ); // Combine conditions through v_kv
12) return N_{2i+1};
```

Figure A.11: CREATE-NEG-CHILDve

When NEG-UPDATEVE is called on the last edge to be decided before a vertex, it must call the TRIGGERV or RESOLVEV methods. This is because v_{kv} is available (since s_{2i+1}==**TRUE**) and hence messages can pass through the vertex through other edges (where possible), even when the last edge is unavailable. The only exception is (once again) the case when two or more vertices are decided on successive levels. In this case neither TRIGGERV nor RESOLVEV must be called; hence s_{2i+1} must be checked by the method.

A.2.4. Redundant Vertices

Unlike the OBDD-A for Model 'v', under Model 've' a vertex does not become redundant immediately after it has been decided. It could be argued that a vertex is redundant for those nodes for which it has been decided as failed; this would mean calling DEL-REDUNDANT immediately on the negative child on levels deciding vertices. However for nodes where s_i is **TRUE** a vertex doesn't become redundant until all adjacent edges have been decided. Since those vertices with s_i **FALSE** are not processed, there is no loss to delaying the call to DEL-REDUNDANT until it can also be called on those vertices with s_i **TRUE**. Hence a vertex under Model 've' becomes redundant much as for Model 'e'; when the vertex and all edges adjacent to it have been decided.

```
CHECK-REDUNDANTve ( ): // e_{ke+1}=(v_a,v_b)
1)  if ( a > b ) then
2)        redf = (kv != b)
3)  else
4)        redf = (kv != a)
5)  return;
```

Figure A.12: CHECK-REDUNDANTve

The DEL-REDUNDANTVE method itself is identical to that of the other models, however it is more efficient to use a modified version of the CHECK-REDUNDANT method since only one vertex can become redundant at one time. This should be checked only on levels where edges are being decided. The CHECK-REDUNDANTVE method is shown in Figure A.12. This method simply compares the value of the lower of the two endpoint subscripts of the next edge to be decided to the subscript of the last vertex decided and assigns the truth value of the comparison to the *redf* variable. The *redt* variable is not required for Model 've'.

```
UPDATE-LEVELve ( k ):
1)  if ( k > kv + ke ) then
2)        ke++;
3)  if ( redf ) then
4)        kv++;
5)  k++;
6)  Swap Q_C and Q_N;
7)  CHECK-REDUNDANTve( );
8)  return;
```

Figure A.13: UPDATE-LEVELve

A.2.5. The OBDD-A Algorithm for Vertex and Edge Failure

Because two new level variables, *kv* and *ke*, are introduced, the initialization of a new diagram level becomes more complex. For this reason the level changing code is separated out into a method, UPDATE-LEVELVE, which is shown in Figure A.13. This makes use of a number of variables from the main OBDD-Ave algorithm[30] to appropriately increment *kv* and *ke*.

The updated OBDD-A algorithm for Model 've' is shown in Figure A.14. The main changes are the consolidation of level updating into the UPDATE-LEVELVE method (line 7), the check on the newly added s_i to see if child generation can be simplified (lines 10-11) and calls to the new versions of the methods described above.

```
OBDD-A (G):                    // Algorithm for ve Models
1)  Initialize the root node N0;
2)  Initialize k=0, ke=0, kv=0, QC = { N0 }, QN = { }, redf=false, reliability = 0;
3)  Remove the first node, Ni, from QC;
4)  if ( (k > kv + ke) and si == false )
5)      Relabel Ni as N2i+1;          // No changes if vertex unavailable
6)  else
7)      N2i+2 = CREATE-POS-CHILDve(Ni, k);
8)      N2i+1 = CREATE-NEG-CHILDve(Ni, k);
9)  for each child node Nfi do
10)     if redf then
11)         DEL-REDUNDANT( Nfi, vkv );
13)     if ( TEST-NODE(Nfi) == 3 ) then
14)         Check each node on QN for isomorphism with Nfi;
15)         if ( an isomorphic node Nx was found ) then
16)             Merge Nfi into Nx;
17)         else
18)             Add Nfi onto QN;
19) if ( QC == { } ) then
20)     if ( QN == { } ) then
21)         return reliability;
22)     else
23)         UPDATE-LEVELve( k );
24) goto 3)
```

Figure A.14: OBDD-Ave

While the depth of the OBDD-Ave for models 1ve and 4.1ve is greater than for the previous two versions (|E| + |V| levels, including the terminal nodes) the downwards isomorphism for nodes with s_i = **FALSE** (lines 10-11) helps reduce the amount of processing required.

[30] These variables can be passed as arguments, accessed directly from the previous level or declared as globals, depending on the implementation.

A.2.6. Example of Building an OBDD-Ave

Since the condition handling for Model 've' is identical to Model 'v', this section considers only one example of building an OBDD-Ave to solve a network with fallible vertices and edges.

For the first example, again consider the network shown in Figure A.15 (a repeat of Figure 3.1) with s = v_0, t = v_3, and each edge and each vertex having a 0.9 chance of being available. This is an example of model 1ve.

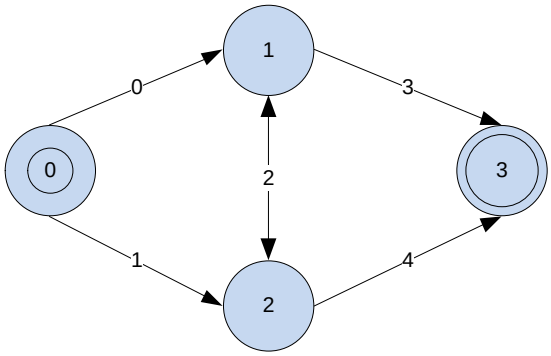


Figure A.15: Simple Network

The resulting OBDD-Ave is shown in Figure A.16. This is far larger than the models that had perfect vertices or perfect edges, since the number of variables roughly doubles. The diagram has a total of 8 levels of non-terminal nodes, since each of the 5 edges has a level and all of the 3 non-target vertices also generate one diagram level.

The root node is N_0={ ({v_0}, 1.0), { }, **TRUE** }, as with the previous two iterations of this example, since the network failure model does not affect the root node; only the source vertex definition affects the root node. However note the additional field for s_0; this is set to **TRUE** although its initial value actually has no bearing on the computation. Other initializations include Q_C={ N_0 }, Q_N = { }, *k*=0, *kv*=0 and *ke*=0. Note that *k* = *kv* + *ke*, which indicates that a vertex is being decided. Since *kv*=0 it is v_0 that is being decided. In addition the redundancy variable *redf* is set to **FALSE**.

The main loop (lines 3-18) starts by removing the first (and the only) node from Q_C and then checking to see whether the special isomorphism comes into play; that if an edge is being decided (*k* < *kv* + *ke*) and the last vertex is failed (s_i = **FALSE**). In this case a vertex is being decided, so two child nodes are created (lines 7-8).

The first child is created by CREATE-POS-CHILDVE, which created N_2 as a copy of N_0 and then checks whether a vertex or edge is being decided. Since this level decides a vertex, the probability is updated by multiplication by Prob(v_0) and s_2 is set to **TRUE**. Hence N_2={ ({v_0}, 0.9), { }, **TRUE** }. Lastly it checks to see if this is the last level before a vertex is to be decided by checking for redundancy (line 10); the last level of the diagram dealing with vertex v_{kv} needs to call TRIGGERVE or RESOLVEVE before redundant information is removed. The CREATE-NEG-CHILDVE method is relatively similar, except that is sets s_1 to be **FALSE**. Hence N_1={ ({v_0}, 0.1), { }, **FALSE** }.

Control passes back to the main loop, which first checks for redundancy (line 10) and then tests both child nodes; in this case both N_1 and N_2 are non-terminal and hence are added to Q_N in turn, with N_2 being tested for isomorphism with N_1 beforehand. Q_N is now { N_1, N_2 }.

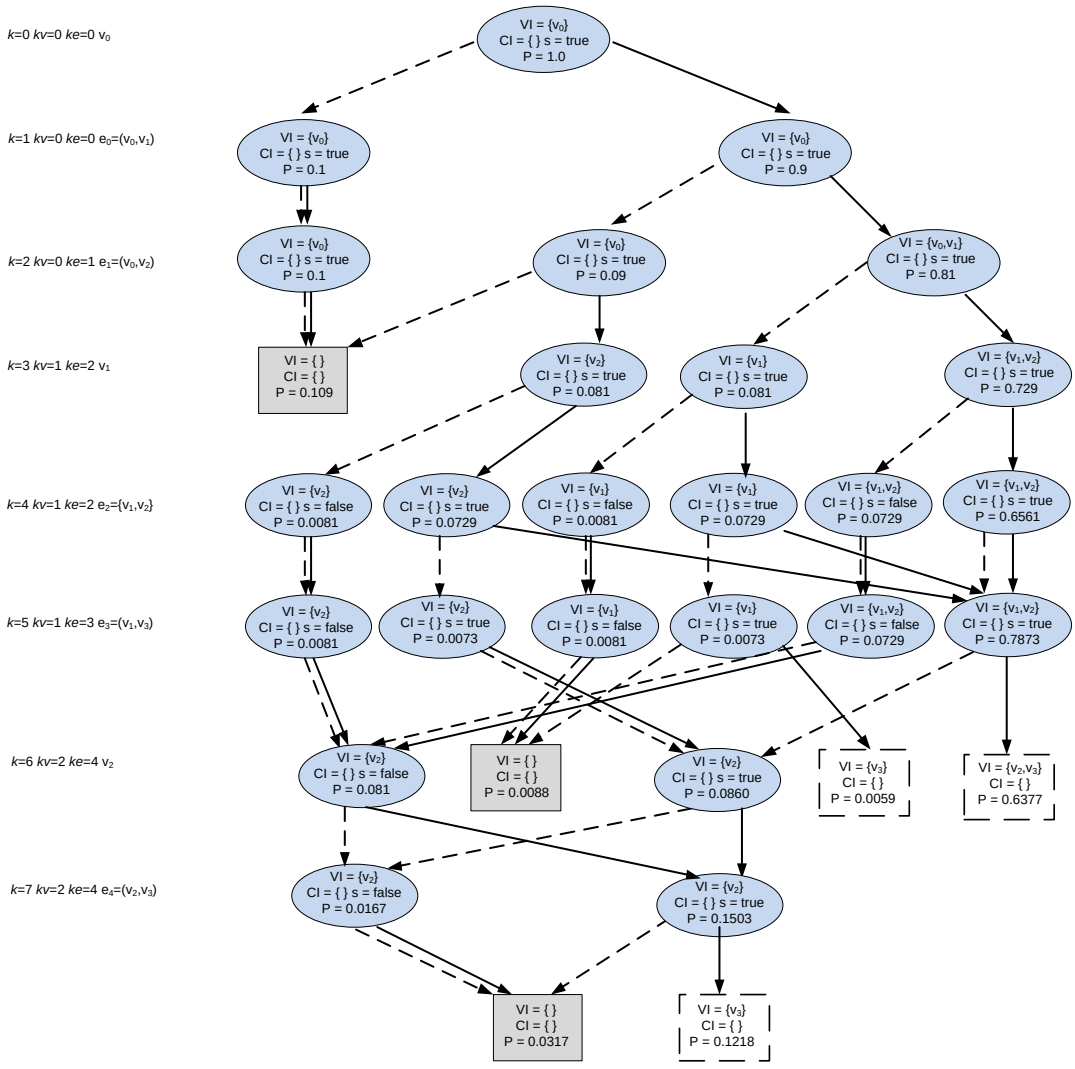


Figure A.16: OBDD-Ave for Simple Network

With the main loop over, the level testing is carried out. Indeed Q_C is empty (line 19) but Q_N is not (line 20) and hence UPDATE-LEVELVE is called. Neither of the first conditions in UPDATE-LEVELVE are **TRUE**, so only *k* is incremented to 1. The contents of Q_C and Q_N are swapped, and CHECK-REDUNDANTVE is called. The next edge is $e_1 = (v_0,v_2)$, and the decision vertex is v_0 so the condition on line 1 fails and *redf* is assigned the value of the negated comparison between the index of v_0 and the first endpoint of e_1; both are zero so *redf* is set to be **FALSE**.

Execution returns to the start of the main loop, and N_1={ ({v_0}, 0.1), { }, **FALSE** } is removed from Q_C. Because *k*=1 and *kv* + *ke* = 0 and s_1 is **FALSE**, only a single child is created by relabeling N_1 to be N_3. This child goes through the usual processes (test for redundancy, termination and isomorphism) and is then added to Q_N. Hence Q_C = { N_2 } and Q_N = { N_3 }. Since Q_C is non-empty the current diagram level is not yet complete, and execution returns to the start of the main loop without updating the level-based variables.

Node N_2={ ({v_0}, 0.9), { }, **TRUE** } is removed from Q_C (leaving Q_C empty) and since s_2 is **TRUE**, two children are created. In this case the first condition in CREATE-POS-CHILDVE fails since this level decides an edge, and lines 6-9 are executed. This gives N_6={ ({v_0}, 0.81), {(v_0,v_1)}, **TRUE** }. The negative child is N_5={ ({v_0}, 0.09), { }, **TRUE** }. Neither child requires a call to TRIGGERV or RESOLVEV since v_0 is not yet redundant. Both are tested and found to be non-terminal. N_5 is then compared to the only node on Q_N, N_3, but is not isomorphic since $s_3 \neq s_5$, and hence N_5 is added to Q_N. N_6 is compared to both N_3 and N_5 but has a different CI, and hence is also added to Q_N. This gives Q_C = { } and Q_N = { N_3, N_5, N_6}.

Since Q_C is empty, the level is updated with a call to UPDATE-LEVELVE. Since *k* > *ke* + *kv* the edge count, *ke*, is incremented to 1. This means that the next level is deciding e_1; since *kv* is still 0 the decision vertex is still v_0. The contents of Q_C and Q_N are swapped and CHECK-REDUNDANTVE is called. We have $e_{ke+1} = e_2 = \{v_1,v_2\}$ and $v_{kv} = v_0$, so *redf* is set to **TRUE**; after the level that is about to be created vertex v_0 will no longer be needed so information regarding v_0 can be removed on this level.

Execution returns to line 3 and N_3={ ({v_0}, 0.1), { }, **FALSE** } is removed from Q_C and processed to give the single child N_7={ ({v_0}, 0.1), { }, **FALSE** }. Since *redf* is

TRUE, information on v_0 is removed giving N_7={ ({ }, 0.1), { }, **FALSE** }, which is a failure node and is discarded.

Since Q_C is non-empty execution returns to the top of the loop and the next node, N_5={ ({v_0}, 0.09), { }, **TRUE** }, is removed from Q_C and processed to give two children. The positive child is N_{12}={ ({v_0}, 0.081), {(v_0,v_2)}, **TRUE** } after the call to ADD-CONDV. Since $v_0 \in VS_{12}$ TRIGGERV is called to give N_{12}={ ({v_0,v_2}, 0.081), {}, **TRUE** } and after the call to DEL-REDUNDANT this becomes N_{12}={ ({v_2}, 0.081), {}, **TRUE** }. The negative child is N_{11}={ ({v_0}, 0.009), { }, **TRUE** }, which doesn't change after the call to TRIGGERV and becomes N_{11}={ ({ }, 0.009), { }, **TRUE** } after the call to DEL-REDUNDANT. N_{11} is a failure node, and hence discarded, while N_{12} is non-terminal and is added to Q_N.

The last node on Q_C, N_6={ ({v_0}, 0.81), {(v_0,v_1)}, **TRUE** }, is similarly processed to give the positive child N_{14}={ ({v_0}, 0.729), {(v_0,v_1),(v_0,v_2)}, **TRUE** }. The calls to TRIGGERV and DEL-REDUNDANT result in N_{14}={ ({v_1,v_2}, 0.729), { }, **TRUE** }. The negative child is N_{13}={ ({v_0}, 0.081), {(v_0,v_1)}, **TRUE** }, and becomes N_{13}={ ({v_1}, 0.081), { }, **TRUE** }. Note that since s_{13} is **TRUE**, v_0 is available to N_{13}, and hence the path is extended through v_0 to v_1, despite the current edge being unavailable. Hence TRIGGERV is required to be called even in the negative child.

Both children are non-terminal, and are hence added to Q_N after isomorphism checks. This means that Q_C = { } and Q_N = { N_{12}, N_{13}, N_{14} }. Note that these nodes are identical to the nodes on the second (*k*=1) level of the OBDD-Av and the third (*k*=2) level of the OBDD-Ae except for the node probabilities.

Since Q_C is empty, UPDATE-LEVELVE is called. On this occasion both of the conditions (lines 1 and 3) are **TRUE** and hence both *ke* and *kv* are incremented, as is *k*. The diagram has finished with v_0 and is about to decide v_1, hence *kv*=1. Similarly the next edge to be decided will be e_2, and hence *ke*=2. The decision variable *k* is now 3, which is equal to *ke*+*kv*. CHECK-REDUNDANT resets *redf* to **FALSE**.

Node N_{12}={ ({v_2}, 0.081), {}, **TRUE** } is processed to give N_{25}={ ({v_2}, 0.0081), {}, **FALSE** } and N_{26}={ ({v_2}, 0.0729), {}, **TRUE** }. Similarly node N_{13}={ ({v_1}, 0.081), { }, **TRUE** } is processed to give N_{27}={ ({v_1}, 0.0081), { }, **FALSE** } and N_{28}={ ({v_1}, 0.0729), { }, **TRUE** } and N_{14}={ ({v_1,v_2}, 0.729), { }, **TRUE** } is processed to give N_{29}={ ({v_1,v_2}, 0.0729), { }, **FALSE** } and N_{30}={ ({v_1,v_2}, 0.6561), { }, **TRUE** }. All are non-terminal and added to Q_N since none of these nodes are isomorphic. Hence Q_N now contains all six nodes. Compare this to the

OBDD-Ae and the OBDD-Av, both of which have at most three nodes on any one level. It can be seen that not only is the OBDD-Ave deeper (*i.e.*, has more levels) than the other diagrams but it may also have considerably more nodes per level.

The next level is initialized, with neither *ke* nor *kv* incremented, although *k* is incremented to 4. The queues are swapped and *redf* remains **FALSE**. Each of the nodes on Q_C is then removed in turn and processed.

The rest of the OBDD-A creation proceeds in a similar manner, and is not detailed here. The full diagram showing all nodes can be seen in Figure A.16 above.

A.2.7. Summary

Yeh, Lin and Kuo [26] propose using the incident edge method by Aggarwal, Misra and Gupta [29]. However the resulting OBDD grows in size to a depth of |V|+|E|+1, with a corresponding exponential increase in the number of nodes generated. Furthermore it is not clear how the algorithm avoids repeating vertex decisions when two edges have the same vertex as an endpoint.

Other algorithms may be able to be modified directly to generate SDP terms or an OBDD in order to include both vertex and edge failure. However in each case the corresponding increase in complexity is exponential. The same has been shown to occur for the OBDD-A. The use of these methods to compute REL for networks with both vertex and edge failure is hence impractical even for relatively small networks.

Appendix B
OBDD-Ap Nodes from Example 5.3.3

Level 0:

N_0=[({ 0 }, 1.00000000), { }, C: [1.00000000, { v0=0 }]

Level 1:

N_1=[({ }, 0.10000000), { }, C: [0.10000000, { }] - Failure node

N_2=[({ 1 2 3 }, 0.90000000), { }, C: [0.90000000, { v1=1 v2=2 v3=3 }]

Level 2:

N_5=[({ 2 3 }, 0.09000000), { }, C: [0.09000000, { v2=2 v3=3 }]

N_6=[({ 2 3 4 }, 0.81000000), { }, C: [0.81000000, { v2=2 v3=3 v4=3 }]

Level 3:

N_{11}=[({ 3 }, 0.00900000), { }, C: [0.00900000, { v3=3 }]

N_{12}=[({ 3 5 }, 0.08100000), { }, C: [0.08100000, { v3=3 v5=5 }]

N_{13}=[({ 3 4 }, 0.08100000), { }, C: [0.08100000, { v3=3 v4=3 }]

N_{14}=[({ 3 4 5 }, 0.72900000), { }, C: [0.72900000, { v3=3 v4=3 v5=5 }]

Level 4:

N_{23}=[({ }, 0.00090000), { }, C: [0.00090000, { }] - Failure node

N_{24}=[({ 6 }, 0.00810000), { }, C: [0.00810000, { v6=7 }]

N_{25}=[({ 5 }, 0.00810000), { }, C: [0.00810000, { v5=5 }]

N_{26}=[({ 5 6 }, 0.07290000), { }, C: [0.07290000, { v5=5 v6=7 }]

N_{27}=[({ 4 }, 0.00810000), { }, C: [0.00810000, { v4=3 }]

N_{28}=[({ 4 6 }, 0.07290000), { }, C: [0.07290000, { v4=3 v6=7 }]

N_{29}=[({ 4 5 }, 0.07290000), { }, C: [0.07290000, { v4=3 v5=5 }]

N_{30}=[({ 4 5 6 }, 0.65610000), { }, C: [0.65610000, { v4=3 v5=5 v6=7 }]

Level 5:

N_{49}=[({ 6 }, 0.00810000), { }, C: [0.00810000, { v6=7 }]

N_{50}=[({ 6 }, 0.00729000), { (5,7,4),(7,5,4)}, C: [0.00729000, { v6=7 }]

N_{51}=[({ 5 }, 0.00810000), { }, C: [0.00810000, { v5=5 }]

N_{52}=[({ 5 }, 0.00729000), { (5,7,4),(7,5,4)}, C: [0.00729000, { v5=5 }]

N_{53}=[({ 5 6 }, 0.07290000), { }, C: [0.07290000, { v5=5 v6=7 }]

N_{54}=[({ 5 6 }, 0.06561000), { (5,7,4),(7,5,4)}, C: [0.06561000, { v5=5 v6=7 }]

N_{55}=[({ }, 0.00081000), { }, C: [0.00081000, { }] - Failure node

N_{56}=[({ 5 7 }, 0.07290000), { }, C: [0.00729000, { v5=6 v7=5 }] [0.06561000, { v5=5 v7=5 }]

N_{58}=[({ 5 6 7 }, 0.65610000), { }, C: [0.06561000, { v5=6 v6=7 v7=5 }] [0.59049000, { v5=5 v6=7 v7=5 }]

Level 6:

N_{99}=[({ 6 }, 0.09558000), { }, C: [0.09558000, { v6=7 }]

N_{102}=[({ 6 }, 0.00656100), { (6,7,6),(7,6,6) }, C: [0.00656100, { v6=7 }]

N_{103}=[({ }, 0.00081000), { }, C: [0.00081000, { }] - Failure node

N_{105}=[({ }, 0.00072900), { }, C: [0.00072900, { }] - Failure node

N_{106}=[({ 6 7 }, 0.78732000), { (7,6,6)}, C: [0.00656100, { v6=7 v7=10 }] [0.00656100, { v6=8 v7=5 }] [0.65610000, { v6=7 v7=5 }]

N_{113}=[({ 7 }, 0.00729000), { }, C: [0.00729000, { v7=5 }]

Level 7:

N_{199}=[({ }, 0.00955800), { }, C: [0. 00955800, { }] - Failure node

N_{200}=[({ 8 }, 0.08602200), { }, C: [0.08602200, { v8=10 }]

N_{205}=[({ }, 0.000656100), { }, C: [0.000656100, { }] - Failure node

N_{206}=[({ 7 8 }, 0.71449290), { (7,8,9)}, C: [0.00590490, { v7=14 v8=10 }] [0.05904900, { v7=10 v8=10 }] [0.00590490, { v7=5 v8=11 }] [0.6436341, { v7=5 v8=10 }]

N_{213}=[({ 7 }, 0.08602200), { }, C: [0.00656100, { v7=10 }] [0.07946100, { v7=5 }]

Level 8:

N_{401}=[({ 8 }, 0.15747129), { }, C: [0.15688080, { v8=10 }] [0.00059049, { v8=11 }]

N_{414}=[{ 8 9 }, 0.64304361, { }, C: [0.00531441, { v8=10 v9=17 }] [0.05314410, { v8=10 v9=13 }] [0.00531441, { v8=11 v9=8 }] [0.57927069, { v8=10 v9=8 }]

N_{427}=[{ }, 0.00860220, { }, C: [0.00860220, { }] - Failure node

N_{428}=[{ 9 }, 0.07741980, { }, C: [0.00590490, { v9=13 }] [0.07151490, { v9=8 }]

Level 9:

N_{803}=[{ }, 0.01574713, { }, C: [0.01574713, { }] - Failure node

N_{804}=[{ 9 }, 0.86218757, { }, C: [0.20502469, { v9=13 }] [0.00053144, { v9=14 }] [0.00053144, { v9=17 }] [0.65610000, { v9=8 }]

Level 10:

N_{1609}=[{ }, 0.08621876, { }, C: [0.02050247, { }] [0.00005314, { }] [0.00005314, { }] [0.06561000, { }] - Failure node

N_{1610}=[{ 9 }, 0.77596881, { }, C: [0.18452222, { v9=13 }] [0.00047830, { v9=14 }] [0.00047830, { v9=17 }] [0.59049000, { v9=8 }] - Success node

Appendix C
OBDD-Hp Nodes from Example 5.4.3

Level 0:

N_0=[{[0]* }, 1.00000000, R: C: [1.00000000, { (v0,v0,0) }]

Level 1:

N_1=[{[1][2][3] }, 0.10000000, R: C: [0.10000000, { }] - Failure node

N_2=[{[1 2 3]* }, 0.90000000, R: C: [0.90000000, { (v0,v1,1) (v0,v2,2) (v0,v3,3) (v1,v2,3) (v1,v3,4) (v2,v3,5) }]

Level 2:

N_5=[{[2 3]*[4] }, 0.09000000, R: C: [0.09000000, { (v0,v2,2) (v0,v3,3) (v2,v3,5) }]

N_6=[{[2 3 4]* }, 0.81000000, R: C: [0.81000000, { (v0,v2,2) (v0,v3,3) (v0,v4,3) (v2,v3,5) (v2,v4,5) (v3,v4,6) }]

Level 3:

N_{11}=[{[3]*[4][5] }, 0.00900000, R: C: [0.00900000, { (v0,v3,3) }]

N_{12}=[{[3 5]*[4] }, 0.08100000, R: C: [0.08100000, { (v0,v3,3) (v0,v5,5) (v3,v5,8) }]

N_{13}=[{[3 4]*[5] }, 0.08100000, R: C: [0.08100000, { (v0,v3,3) (v0,v4,3) (v3,v4,6) }]

N_{14}=[{[3 4 5]* }, 0.72900000, R: C: [0.72900000, { (v0,v3,3) (v0,v4,3) (v0,v5,5) (v3,v4,6) (v3,v5,8) (v4,v5,8) }]

Level 4:

N_{23}=[{[4][5][6] }, 0.00090000, R: C: [0.00090000, { }] - Failure node

N_{24}=[{[4][5][6]* }, 0.00810000, R: C: [0.00810000, { (v0,v6,7) }]

N_{25}=[{[4][5]*[6] }, 0.00810000, R: C: [0.00810000, { (v0,v5,5) }]

N_{26}=[{[4][5 6]* }, 0.07290000, R: C: [0.07290000, { (v0,v5,5) (v0,v6,7) (v5,v6,12) }]

N_{27}=[{[4]*[5][6] }, 0.00810000, R: C: [0.00810000, { (v0,v4,3) }]

N_{28}=[{[4 6]*[5] }, 0.07290000, R: C: [0.07290000, { (v0,v4,3) (v0,v6,7) (v4,v6,10) }]

N_{29}=[{[4 5]*[6] }, 0.07290000, R: C: [0.07290000, { (v0,v4,3) (v0,v5,5) (v4,v5,8) }]

N_{30}=[{[4 5 6]* }, 0.65610000, R: C: [0.65610000, { (v0,v4,3) (v0,v5,5) (v0,v6,7) (v4,v5,8) (v4,v6,10) (v5,v6,12) }]

Level 5:

N_{49}=[{[5][6]*[7] }, 0.00810000, R: C: [0.00810000, { (v0,v6,7) }]

N_{50}=[{[5 7][6]* }, 0.00729000, R: C: [0.00729000, { (v0,v6,7) (v5,v7,4) }]

N_{51}=[{[5]*[6][7] }, 0.00810000, R: C: [0.00810000, { (v0,v5,5) }]

N_{52}=[{[5]*[5 7][6] }, 0.00729000, R: C: [0.00729000, { (v0,v5,5) (v5,v7,4) }]

N_{53}=[{[5 6]*[7] }, 0.07290000, R: C: [0.07290000, { (v0,v5,5) (v0,v6,7) (v5,v6,12) }]

N_{54}=[{[5 6]*[5 7] }, 0.06561000, R: C: [0.06561000, { (v0,v5,5) (v0,v6,7) (v5,v6,12) (v5,v7,4) }]

N_{55}=[{[5][6][7] }, 0.00081000, R: C: [0.00081000, { }] - Failure node

N_{56}=[{[5 7]*[6] }, 0.07290000, R: C: [0.00729000, { (v0,v5,6) (v0,v7,5) (v5,v7,4) }] [0.06561000, { (v0,v5,5) (v0,v7,5) (v5,v7,4) }]

N_{58}=[{[5 6 7]* }, 0.65610000, R: C: [0.06561000, { (v0,v5,6) (v0,v6,7) (v0,v7,5) (v5,v6,13) (v5,v7,4) (v6,v7,12) }] [0.59049000, { (v0,v5,5) (v0,v6,7) (v0,v7,5) (v5,v6,12) (v5,v7,4) (v6,v7,12) }]

Level 6:

N_{99}=[{[6]*[7] }, 0.09558000, R: C: [0.09558000, { (v0,v6,7) }]

N_{102}=[{[6]*[6 7] }, 0.00656100, R: C: [0.00656100, { (v0,v6,7) (v6,v7,6) }]

N_{103}=[{[6][7] }, 0.00081000, R: C: [0.00081000, { }] - Failure node

N_{105}=[{[6][7] }, 0.00072900, R: C: [0.00072900, { }] - Failure node

N_{106}=[{[6 7]* }, 0.78732000, R: C: [0.06561000, { (v0,v6,7) (v0,v7,10) (v6,v7,6) }] [0.00656100, { (v0,v6,8) (v0,v7,5) (v6,v7,6) }] [0.64953900, { (v0,v6,7) (v0,v7,5) (v6,v7,6) }] [0.06561000, { (v0,v6,7) (v0,v7,5) (v6,v7,12) }]

N_{113}=[{[6][7]* }, 0.00729000, R: C: [0.00729000, { (v0,v7,5) }]

Level 7:

N_{199}=[{[7][8] }, 0.00955800, R: C: [0.00955800, { }] - Failure node

N_{200}=[{[7][8]* }, 0.08602200, R: C: [0.08602200, { (v0,v8,10) }]

N_{205}=[{[7][8] }, 0.00065610, R: C: [0.00065610, { }] - Failure node

N_{206}=[{[7 8]* }, 0.71449290, R: C: [0.00590490, { (v0,v7,14) (v0,v8,10) (v7,v8,9) }] [0.05904900, { (v0,v7,10) (v0,v8,10) (v7,v8,9) }] [0.00590490, { (v0,v7,5) (v0,v8,11) (v7,v8,9) }] [0.58458510, { (v0,v7,5) (v0,v8,10) (v7,v8,9) }] [0.05904900, { (v0,v7,5) (v0,v8,10) (v7,v8,15) }]

N_{213}=[{[7]*[8] }, 0.08602200, R: C: [0.00656100, { (v0,v7,10) }] [0.07946100, { (v0,v7,5) }]

Level 8:

N_{401}=[{[8]*[9] }, 0.15747129, R: C: [0.15688080, { (v0,v8,10) }] [0.00059049, { (v0,v8,11) }]

N_{414}=[{[8 9]* }, 0.64304361, R: C: [0.00531441, { (v0,v8,10) (v0,v9,17) (v8,v9,12) }] [0.05314410, { (v0,v8,10) (v0,v9,13) (v8,v9,12) }] [0.00531441, { (v0,v8,11) (v0,v9,8) (v8,v9,12) }] [0.52612659, { (v0,v8,10) (v0,v9,8) (v8,v9,12) }] [0.05314410, { (v0,v8,10) (v0,v9,8) (v8,v9,18) }]

N_{427}=[{[8][9] }, 0.00860220, R: C: [0.00860220, { }] - Failure node

N_{428}=[{[8][9]* }, 0.07741980, R: C: [0.00590490, { (v0,v9,13) }] [0.07151490, { (v0,v9,8) }]

Level 9:

N_{803}=[{[9] }, 0.01574713, R: C: [0.01574713, { }] - Failure node

N_{804}=[{[9]* }, 0.86218757, R: C: [0.20502469, { (v0,v9,13) }] [0.00053144, { (v0,v9,14) }] [0.00053144, { (v0,v9,17) }] [0.65610000, { (v0,v9,8) }]

Level 10:

N_{1609}=[{ }, 0.08621876, R: C: [0.02050247, { }] [0.00005314, { }] [0.00005314, { }] [0.06561000, { }] - Failure node

N_{1610}=[{ [9]* }, 0.77596881, R: C: [0.18452222, { (v0,v9,13) }] [0.00047830, { (v0,v9,14) }] [0.00047830, { (v0,v9,17) }] [0.59049000, { (v0,v9,8) }] - Success node

Appendix D
OBDD-H Pseudo-Code for Model 3

```
NON-TERMINAL-NODEh (N_i):
1)  if (there are sufficient available targets from each group are in the marked block) then
2)      reliability += P_i *Prob(T);
3)      return false;          //a success node
4)  else if ( Part_i == { } ) then
5)      return false;          // a failure node
6)  else
7)      return true;           //a non-terminal node
```

Figure D.1: NON-TERMINAL-NODEh for Model 3

```
      EDGE-CONTRACTh (N_i, k):                  // e_k = {v_f, v_t}
e     1)  Locate block b_y containing v_t in VI_i;  // Set y=B if no block contains v_t
e     2)  if ( y == B ) then                    // No block contains v_t
e     3)      Add v_t to b_x;
e     4)  else if ( y > 0 ) then                // v_f and v_t in different blocks – merge
e     5)      for each v_a∈b_y do
e     6)          Delete v_a from b_y;
e     7)          Add v_a to b_0;
e     8)      if ( b_y is marked ) then
e     9)          Mark b_0;
e     10)     Delete b_y;
v,ve  11) Create block b=[v_f, v_t];
v,ve  12) Add b to Part_i;
      13) return N_i;
```

Figure D.2: General EDGE-CONTRACTh for Model 3

```
EDGE-DELETEh (N_i, k):
1)  if ( no partition contains v_t ) then
2)      Add new block b_j =[v_t];
3)      if ( v_t is a target ) then
4)          Mark b_j;
5)  return N_i;
```

Figure D.3: General EDGE-DELETEh for Model 3