Intelligent Reliability Analysis

Jeffrey A. Barnett & Tom Verma*
Automation Sciences Laboratory
The Northrop Corporation
Pico Rivera, CA 90660

Abstract

An ideal automated reliability analysis system would take a CAD model as input, then identify critical components and output failure probabilities. Unfortunately, the practical systems in use today require a fault tree—a kind of and/or graph—as their input. Since the cognitive distance between the CAD model and its associated fault tree is large, the manual translation step is a source of many mistakes. Furthermore, the use of fault trees makes it difficult to handle the cyclical dependencies that naturally occur among system components. The functional dependency graph, a representation midway between CAD models and fault trees, is described below and methodology that automates reliability and diagnostic analyses is developed. When the resultant technology is employed, the knowledge burden is more equally shared between the user and the computer. In addition, more complicated systems can be properly analyzed because cyclical dependencies can be represented.

1 Summary

A reliability analysis produces, from a system design, a list of its critical sets and the probability that the system will fail to perform specified missions. A *critical set* is a set of system components that satisfies two conditions: (1) the system necessarily fails if all elements of the critical set fail and (2) no proper subset of the critical set satisfies the first condition. Small critical sets represent potential vulnerabilities that can easily be triggered by outside influences. Battle damage to a critical weapon system component is an example.

The results of reliability analyses are used throughout the system life cycle. During product engineering, the data is used to improve designs; for example, an engineer may increase redundancy to decrease the number of small critical sets. Later, the analysis is used to determine the appropriate size of the spares inventory and to form diagnostic plans.

The next section is a brief introduction and an overview to the portions of reliability analysis relevant to this article. The interested reader is referred to two classics for additional information: Amstadter [1] and Barlow, et al [2]. Current approaches are described in the *IEEE Transactions on Reliability Analysis* and in the proceedings and tutorial notes of the yearly *Annual Reliability and Maintainability Symposium*. Barnett and Verma [3] develop many of the results that appear below.

Section 3 identifies several problems with current practice: the required input is difficult to prepare and the source of many errors, a system and each of its subsystems require separate analyses, and conventional methods do not adequately handle cyclical dependencies. Section 4 defines a new representation, the functional dependency graph, that addresses these problems and Section 5 discusses issues of dependency cycles in more detail. Section 6 provides computational methods and, finally, Section 7 discusses current status and future directions. This paper combines AI techniques, used in theorem proving and logic programming, with conventional reliability practice in a way that extends our ability to analyze complex systems.

2 Remedial Reliability Analysis

The prototypical reliability analysis proceeds in four steps:

1. Determine component failure probabilities as a function of system missions.
2. Represent component dependencies by a fault tree—a structure defined below.
3. Enumerate critical sets from the fault tree.
4. Calculate system failure probabilities from the component failure probabilities and the list of critical sets.

*Verma's current address is Tudor Investment Corporation, New York, NY 10006. Email addresses of the authors are jbarnett@nrtc.northrop.com and verma@tudor.com.

In Proc. Tenth IEEE Conference on Artificial Intelligence for Applications, San Antonio, 1994.

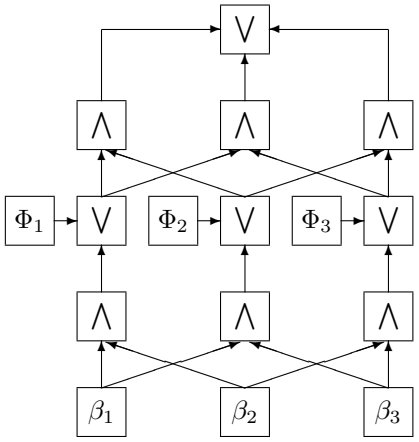

Figure 1: Fault tree for a redundant circuit.

The first step typically uses assumptions such as that missions can be parameterized by their duration and that component failure probabilities have exponential distributions, e.g.,

$$p(c) \quad = \quad 1 - e^{-\lambda_c T},$$

where c is a component, λ_c is its failure rate—the reciprocal of mean time to failure, and T is mission duration. The λ terms are found in engineering handbooks or by testing.

The second step is to represent system dependencies by an acyclic and/or graph such as the one shown in Figure 1. This structure is variously called a fault tree or a series/parallel graph. Leaf nodes are system components and the root represents the system. An *and* node in the tree fails if all of its children fail; similarly, an *or* node fails if any of its children do. Thus, *true* denotes failure. Since the graph is acyclic and interpreted as a boolean expression, it can always be converted to an equivalent tree by replicating nodes that have multiple parents.

The Φ_i nodes are circuit components and the β_i nodes are batteries in the fault tree shown by Figure 1. The root fails when the services of any two of the three Φ_i are not available and each Φ_i denies service if it fails or both batteries that support it fail. The failure condition, f, for the example system is

$$\begin{aligned} f \quad = \quad & (\Phi_1 \vee \beta_1\beta_2)(\Phi_2 \vee \beta_1\beta_3) \\ & \vee (\Phi_1 \vee \beta_1\beta_2)(\Phi_3 \vee \beta_2\beta_3) \\ & \vee (\Phi_2 \vee \beta_1\beta_3)(\Phi_3 \vee \beta_2\beta_3) \end{aligned} \tag{1}$$

where juxtaposition means *and* ($\wedge$) in boolean expressions. The symbols, Φ_i and β_i, represent propositions that components with the same name fail.

The third step of the reliability analysis begins with a fault tree such as the one shown by the figure above. The fault tree is converted to a boolean expression (Eq. 1) then transformed into conjunctive normal form and simplified by subsumption.[1] The result of this process applied to Eq. 1 is

$$\begin{aligned} f \quad = \quad & \Phi_1\Phi_2 \vee \Phi_1\Phi_3 \vee \Phi_2\Phi_3 \vee \beta_1\beta_3\Phi_1 \\ & \vee \beta_2\beta_3\Phi_1 \vee \beta_1\beta_2\Phi_2 \vee \beta_2\beta_3\Phi_2 \\ & \vee \beta_1\beta_2\Phi_3 \vee \beta_1\beta_3\Phi_3 \vee \beta_1\beta_2\beta_3, \end{aligned} \tag{2}$$

a form where each term (conjunction) is a critical set of the system.

The final step of the reliability analysis computes the probability of system failure in terms of the probabilities of individual component failures. Two reasonable assumptions are usually made to simplify and speed up probability calculations. The first is that the probabilities of component failures are marginally independent of each other and the second is that the probabilities are small. When these assumptions are valid, the failure probability of a critical set is the product of the failure probabilities of its elements and the failure probability for the system can be accurately approximated by the sum of the failure probabilities of its critical sets. Thus,

$$p(f) \quad \approx \quad p(\Phi_1)p(\Phi_2) + \cdots + p(\beta_1)p(\beta_2)p(\beta_3) \tag{3}$$

is derived from Eq. 2 as a reasonable approximation for the example.

3 Trouble in Paradise

Critical set enumeration and failure probability calculation are straightforward though computationally expensive procedures. Both are mechanical operations on the fault tree—the latter requires component failure probabilities as well. However, preparing an appropriate fault tree from a system specification is difficult and the source of many mistakes. This step requires deep knowledge of the inner workings of a system and how its components depend on and support each other. In fact, two systems with isomorphic component connection graphs do not necessarily have the same fault tree.

Another problem occurs when dependency loops exist among components because the acyclic nature of the fault tree prohibits a simple representation strategy. Consider the following set of boolean equations that are encountered in Section 5 for an example with

[1] Subsumption is a reduction that replaces the form "$\alpha\beta \vee \alpha$" with the equivalent "α", where α and β are arbitrary boolean expressions.

dependency loops.

$$\begin{array}{lll} a = A \vee bx & b = B \vee ac \vee ay \vee cy & c = C \vee bz \\ x = X & y = Y & z = Z \end{array} \quad (4)$$

The literals A, B, C, X, Y, and Z represent failures of components with the same names and the variables a, b, c, x, y, and z, represents failure of the services associated with them. The general solution to these equations is

$$\begin{array}{rcl} a & = & A \vee BX \vee CXY \vee (C \vee Y \vee Z)X\theta \\ b & = & B \vee AC \vee AY \vee CY \\ & & \quad \vee (AZ \vee CX \vee XY \vee YZ \vee XZ)\theta \\ c & = & C \vee BZ \vee AYZ \vee (A \vee Y \vee X)Z\theta, \end{array} \quad (5)$$

where θ is a free boolean parameter. Thus, the solution is a parameterized family and the critical-set list cannot be formed until θ is specified. An additional problem is that variable failure probabilities are not uniquely determined either. Rewrite these equations more abstractly as $v = \xi_v \vee \mu_v\theta$, where v is either a, b, or c, and it follows that

$$p(v) \quad = \quad p(\xi_v) + p(\overline{\xi}_v\mu_v|\theta) \cdot p(\theta).$$

Clearly, $p(v)$ cannot be calculated without $p(\theta)$. Further, unless θ and the literals are marginally independent, some conditional probabilities will be needed as well. All that can be inferred at this point is that

$$p(\xi_v) \leq p(v) \leq p(\xi_v \vee \mu_v), \quad (6)$$

a range that may be too large to be informative. The left and right end points of this interval correspond, respectively, to $\theta = \mathit{false}$ and $\theta = \mathit{true}$.

Another problem with current practice is that a separate fault tree must be generated for each subsystem. The technology proposed in the following sections address the issues discussed here. The proposal provides a representation—the functional dependency graph—that is closer to the original system specification, handles dependency loops, and combines the analysis of a system with analyses of its subsystems.

4 Functional Dependency Graphs

Working system components provide functionalities that are needed by other components and subsystems to function properly. The dependencies are on component *functionalities*, not the components per se. These dependencies result from and are the bases of the system architecture: they are the underlying reasons that systems should work and they tell us how and why systems can fail, i.e., dependencies determine fault trees.

A *functional dependency graph* (FDG) is a data structure that encodes dependency knowledge. An FDG is a directed graph; its nodes represent functionalities and its edges represent dependencies. The functionality at an edge's head depends on the one at its tail. The name of the functionality represented by a node labels all edges that originate at that node.

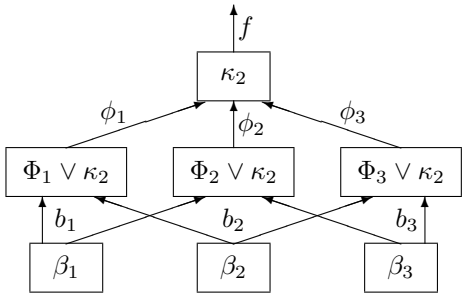

Figure 2: FDG for a redundant circuit.

Figure 2 is a functional dependency graph. It depicts the same system as Figure 1. Each node is labeled with a boolean expression that describes how that node depends on its inputs and the mechanisms in the system. The κ_i symbols are shorthand for expressions that evaluate *true* when i or more arguments (inputs to the node) are *true*. Thus, $\kappa_1 = x_1 \vee x_2$ and $\kappa_2 = x_1x_2$ for a node with inputs x_1 and x_2, while

$$\begin{array}{rcl} \kappa_1 & = & x_1 \vee x_2 \vee x_3 \\ \kappa_2 & = & x_1x_2 \vee x_1x_3 \vee x_2x_3 \\ \kappa_3 & = & x_1x_2x_3 \end{array}$$

for a node with inputs x_1, x_2, and x_3.

In this example, the b_i are three functionalities associated with providing electric power. In another system, they might be lumped to provide a single functionality but not here. Though these three may be similar, they are not identical—usage distinguishes them. The ϕ_i are the non-redundant computational functionalities provided by the Φ_i, and f is the redundant functionality that is this system's design goal.

A boolean equation, representing the failure condition for a functionality, is associated with each node. The equations for this example are

$$\begin{array}{rcl} f & = & \kappa_2(\phi_1, \phi_2, \phi_3) \\ \phi_1 & = & \Phi_1 \vee \kappa_2(b_1, b_2) \\ \phi_2 & = & \Phi_2 \vee \kappa_2(b_1, b_3) \\ \phi_3 & = & \Phi_3 \vee \kappa_2(b_2, b_3) \\ b_1 & = & \beta_1 \\ b_2 & = & \beta_2 \\ b_3 & = & \beta_3 \end{array}$$

and the solution for f is found by simple substitution. It coincides with that displayed by Eqs. 1 and 2.

The FDG for a simple system is similar to a compact version of its fault tree. However, two advantages of the FDG are immediately apparent. The first and most important is that the FDG is easier to generate than a fault tree because it more closely corresponds to the original design intent. In the example, the reason for multiple copies of similar components is to increase reliability via redundancy. This fact is clear because the non-redundant capabilities (the ϕ_i) are labeled and distinguished from the more fault-tolerant capability, f. Using an FDG should, because functionalities and design intent are more explicit, decrease mistakes in reliability analyses.

The second advantage is that fault conditions for the ϕ_i and b_i, considered as subsystems, are immediately available from the same representation. This is not true of the example fault tree shown in Figure 1 since the functionalities are not identified.

A corollary to the above is that it should be easier to incorporate design changes in a reliability analysis when FDGs are used. Only that portion of an FDG describing how a modified functionality is provided needs to be changed. Further, those changes need only be made once. More must be done when using a fault tree because there is no clear demarcation between service providers and service consumers. Using a fault tree formalism, a change must be made to the fault tree for each subsystem that uses the modified service.

A third advantage of FDGs for more complicated systems is discussed in the next section. The case where dependency cycles exist is considered and incorporated in the formalism.

5 Dependency Cycles

System functionalities are inherently co-dependent. For example, power from an automobile engine is not available without oil pressure, but oil pressure depends on the oil pump which needs engine power to work. Therefore, it is necessary and natural that an FDG should permit dependency cycles.

Figure 3 is an example of an FDG for a system with dependency cycles: X, Y, and Z are generators and the associated functionalities, x, y, and z, are raw power. A and C are regulators so the functionalities a and c are smoothed power. The functionality a (respectively c) is available when the regulator A (respectively C), along with at least one input power source, is available. The functionality b is mission power. It depends on the correct functioning of combiner B and the availability of two of its three input power sources (at least one must be smoothed). The boolean failure

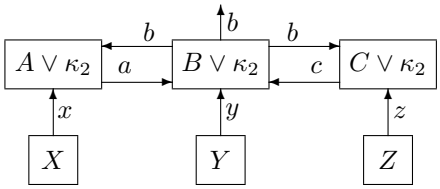

Figure 3: An FDG with cycles.

conditions for the functionalities are given by Eq. 4 and the general solution to those equations is the family, with free parameter θ, displayed by Eq. 5.

As in this example, the general solution to the boolean equations associated with an FDG containing cycles may involve one or more free parameters. The particular solutions are found by substituting arbitrary boolean expressions for each. Thus, the FDG does not uniquely identify critical sets or determine failure probabilities. However, if we stipulate the closed-world assumption that

> *A functional dependency graph explicitly encodes all relevant dependencies.*

the problems of non-determinism can be eliminated.

Our stipulation would be violated if the expression substituted for a free parameter contained literals or variables. Therefore, the value assigned to a free parameter must be a constant, i.e., either *true* or *false*. When all free parameters are simultaneously *true* or simultaneously *false*, the solutions correspond, respectively, to the maximum and minimum fixed points of the original equations. The solution when the free parameters are *true* is the union of the failure modes of all models consistent with the FDG, while the solution when the free parameters are *false* is the intersection, i.e., the latter is the set of failure modes that are *structurally implied* by the FDG.

Critical sets and, hence, failure probabilities will be completely determined by the FDG if we can agree on a mechanical method to assign values, *true* or *false*, to each free parameter. We propose that all assignments be *true*. The disadvantage of this choice is that it generally requires more computational resources to find the maximal solution than the minimal one. However, there are several theoretical and practical advantages that offset this fact.

Our experience indicates that it is easier to encode a correct model when it is known that the semantics are the maximal solution because the FDG more resembles the original system architecture. Consider the example shown in Figure 3 and the critical-set list for b in Eq. 5 with $\theta = true$. AZ is a critical set as it

should be. Only one power source, Y, supports b: Z has failed and X is masked because A has failed. In the minimal solution ($\theta = \mathit{false}$), AZ is not a critical set, and this is tantamount to assuming that power from Y and from the path Y-B-C-B can support b without X or Z. Clearly, this is not the design intent of the example system.

A second advantage of the maximal solution is that the associated model is better for *diagnostic* reasoning. Typically, diagnostic plans use critical-set analyses to determine test sequences. If the maximal solution is used, the test plan considers critical sets that cause system failures when dependency loops might not be self-supporting. In the example system, AZ is a possible critical set because the B-C-B cycle, supported by Y alone, is not considered a second source of useful power distinct from Y.

The third advantage of the maximal solution is that it leads to more realistic estimates of failure probabilities. Note, the minimum and maximum solutions provide, respectively, lower and upper bounds on these probabilities (Eq. 6). The lower bound is too optimistic—systems fail for more reasons than are structurally implied. Control theory teaches us that feedback loops can have more than one limit-cycle, a quasi-stable state that can be entered from a given set of initial conditions. A system or subsystem typically fails when any but the intended limit-cycle is entered. The maximal solution accounts for these extra possibilities; the minimum solution does not.

Thus, the recommendation is to use the maximum solution to model systems via the FDG formalism. However, the analysis procedures discussed in the next section can, generate either solution.

6 Analysis Procedures

This section briefly describes mechanical techniques to do reliability analyses given an FDG and the component failure probabilities. It is assumed that negation is not used in boolean expressions, a standard assumption in the reliability field. (How can the correct operation of one part of a system depend on another part failing?) However, de Kleer, et al [5] argue convincingly that negation makes *diagnostic* reasoning more accurate when explicit failure modes are modeled. For example, the output of a two-inverter pipe can consistently be correct if both inverters fail to invert.

Figure 4 depicts the databases and computations necessary to do a reliability analysis. The FDG and the component failure probabilities (CFP) are the inputs. A structure equivalent to a fault tree (FT) is generated for each functionality by a substitution process (`sub`) and critical sets (CS) are found by subsumption and simplification (`ss`). Finally, the functional failure probabilities (FFP) are found by a calculation ($\Sigma\Pi$) suggested by Eq. 3. The remainder of this section describes `sub` and `ss` in more detail.

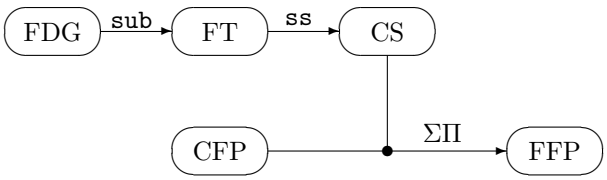

Figure 4: Reliability analysis data flow.

The `sub` procedure uses substitution to make the boolean equation for each functionality variable-free. (Recall, a variable is a symbol that denotes a functionality.) This procedure is demonstrated on the equation-set (Eq. 4) for the example in Figure 3. To remove variables, first substitute the definitions of x, y, and z, respectively, into the equations for a, b, and c. The result is

$$a = A \vee bX \quad b = B \vee ac \vee aY \vee cY \quad c = C \vee bZ.$$

Next, substitute a into b to get

$$b \;=\; B \vee (A \vee bX)c \vee (A \vee bX)Y \vee cY,$$

a self-referential definition—b is directly defined in terms of itself. This form can be written more abstractly as $b = \alpha \vee \beta b$, a form whose most general solution is $b = \alpha \vee \beta\theta$, where θ is a free parameter.

> Proof: Any solution of $b = \alpha \vee \beta b$ must have the form $b = \alpha \vee \beta\theta$ for some θ. Now let θ be arbitrary and note that $\alpha \vee \beta(\alpha \vee \beta\theta) = \alpha \vee \beta\theta$; thus, $\alpha \vee \beta\theta$ is a solution for *every* θ.

The maximal (minimal) solution is generated by replacing each θ with *true* (*false*) whenever necessary to break self references. Thus, the maximal form for b at this point is

$$b \;=\; B \vee (A \vee X)c \vee (A \vee X)Y \vee cY.$$

Similarly, substituting c into b and choosing the maximal solution yields

$$b \;=\; B \vee (A \vee X)(C \vee Z) \vee (A \vee X)Y \vee (C \vee Z)Y,$$

a result that is equivalent to a fault-tree representation of the maximal solution for b. The expression for b is

now variable-free and can be substituted into a and c to complete the `sub` procedure. To wit,

$$\begin{aligned} a &= A \vee \Big(B \vee (A \vee X)(C \vee Z) \\ &\qquad \vee (A \vee X)Y \vee (C \vee Z)Y\Big)X \quad (7) \\ c &= C \vee \Big(B \vee (A \vee X)(C \vee Z) \\ &\qquad \vee (A \vee X)Y \vee (C \vee Z)Y\Big)Z. \end{aligned}$$

It is worth noting that the result of `sub` is independent of the order of substitutions as long as *true* (*false*) is substituted at every opportunity to break self references. This procedure is similar to the one described by Dionne, et al [6] to handle cycles in terminological definitions.

The next step of the reliability analysis is to generate critical sets from the variable-free expressions. The `ss` procedure transforms the expressions to equivalent ones in conjunctive normal form—a disjunction of conjunctions—by simplification and subsumption. The strategy is to multiply out complex conjuncts, use associativity (replace $\alpha \vee (\beta \vee \gamma)$ with $\alpha \vee \beta \vee \gamma$), and do subsumption (replace $\alpha \vee \alpha\beta$ with α) whenever possible. When no more of these operations can be done, the expression is in the desired form. As an example, consider Eq. 7. After multiplying and using associativity, it is

$$\begin{aligned} a &= A \vee BX \vee ACX \vee AXZ \vee CX \vee XZ \\ &\qquad \vee AXY \vee XY \vee CXY \vee XYZ, \end{aligned}$$

and after subsumption it is,

$$a = A \vee BX \vee CX \vee XZ \vee XY,$$

which agrees with Eq. 5 when $\theta = true$.

The `ss` process is by far the most computationally costly part of reliability analysis: since the number of critical sets can grow exponentially in the size of the fault tree, enumerating them is inherently exponential in the worst case. Therefore, methods that reduce the associated computational costs are vital. We are currently investigating Trie structures (named by Fredkin [7], analyzed by Knuth [8], and related to the present application by de Kleer [4]) to reduce the complexity of the subsumption task.

7 Current Status & Future Directions

The Northrop Automation Sciences Laboratory has implemented the technology described above. One version, written in C using X-windows, is in production use on a variety of real-world systems. The users of this system report favorably on the differential capabilities offered from past tools and approaches. In particular, the analysts make fewer mistakes and they find FDGs easier to generate and manipulate than fault trees.

Several experimental versions are implemented on Symbolics Lisp Machines. One experiment in particular is worth mentioning—the use of hashing to speed up the generation of critical sets. When the `ss` procedure is applied recursively to a part of the fault tree, a hash table is used to determine if this calculation has already been done. If so, the conjunctive normal form equivalent is returned. If not, that form is calculated and put in the table using the fault subtree as the hash key. The `sub` process canonicalizes the boolean forms that it produces to make this procedure more efficient.

Experience has shown that there is a modest speed up when finding the critical sets for a single functionality. The major gain is when the technique is applied to finding critical sets for several functionalities in the same FDG. Since many functionality equations contain the same subexpressions, the speed up is often an order of magnitude or more, particularly for large systems. A future experiment is planned to compare the speed up available from hashing with the one available from Trie structures and to see if there is a way to combine the advantages of both.

Another possible avenue for further investigation is suggested because functional dependency graphs are similar to Bayesian networks [9] with boolean variables. The difference is that cycles are permitted in FDGs but not in Bayesian networks since cycles would compromise the semantics of independence assertions with the latter. The issue is whether insights gained in the work reported here can indicate methods that will properly incorporate dependency cycles into a general Bayesian network formalism.

Acknowledgements

The authors wish to thank Christine Bainbridge, Ted Johnson, and Tim Verma for their timely and convincing demonstrations of the value of fixed points in a variety of endeavors.

References

[1] B. L. Amstadter. *Reliability Mathematics*. McGraw-Hill. 1971.

[2] R. E. Barlow, J. B. Fussell, and N. D. Singpurwalla (eds). *Reliability and Fault Tree Analysis*. SIAM. 1975.

[3] J. A. Barnett and T. Verma. Functional block diagrams: logic and probability. The Northrop Corporation. Palos Verdes Peninsula, CA. 1992.

[4] J. de Kleer. An improved incremental algorithm for generating prime implicants. *Proceedings of the Tenth National Conference on Artificial Intelligence*. pp. 780–785. 1992.

[5] J. de Kleer, A. K. Mackworth, and B. Reiter. Characterizing diagnoses. *Proceedings of the Eight National Conference on Artificial Intelligence*. pp. 324–329. 1990.

[6] R. Dionne, E. Mays, and F. J. Oles. A Non-well-founded approach to terminological cycles. *Proceedings of the Tenth National Conference on Artificial Intelligence*. pp. 761–766. 1992.

[7] E. Fredkin. Trie memory. *Communications of the Association for Computer Machinery* 3. pp. 490–500. 1960.

[8] D. E. Knuth. *Sorting and Searching: The Art of Computer Programming*, Vol. 3. Addison Wesley. 1973.

[9] J. Pearl. *Probabilistic Reasoning in IntelligentSystems: Networks of Plausible Inference*. Morgan Kaufmann. 1988.