21

Development of a Pedigree Analysis Tool for Genetics Counselors

by

Jervis C. Lui

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1998
[June 1998]

Author ______________________
Department of Electrical Engineering and Computer Science
May, 1998

Certified by ______________________
Professor Peter Szolovits
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________
Arthur Smith
Chairman, Department Committee on Graudate Students

Development of a Pedigree Analysis Tool for Genetics Counselors

by

Jervis C. Lui

Submitted to the
Department of Electrical Engineering and Computer Science on

21 May, 1998

In partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis attempts to build several modules of computational procedures and objects that helps a computer program, Geninfer, to perform the computations necessary in pedigree analysis. The main focus will be on the functions of Geninfer that allows it to analyze multiple loci Mendelian disorders.

Thesis Supervisor:	Professor Peter Szolovits	
Title:	Director,	MIT Clinical Decision Making Group
	Professor,	MIT Department of Electrical Engineering and Computer Science

Acknowledgments

I greatly appreciate Professor Peter Szolovits for providing me with the opportunity to participate in this research. This thesis would not exist without his guidance and support. It has been an honor and pleasure to be supervised by Professor Szolovits.

I would also like to express my appreciation to Sean Doyle, Eric Jordan, and Milos Hauskrecht. Their help and suggestions went beyond all expectations, and have greatly contributed to this work. I will surely miss the times that we worked together as a team.

I must also thank the following people for their support and friendship:

To my family: I cannot describe all the feelings that I have for you all. You will always be the most important people in my life. Thanks for everything.

To all my relatives, especially Wan Ping E: Words cannot describe all the feelings. I can only say that my life will not be as wonderful without your help, guidance, and love. Thank you.

To my friends in MacGregor House, especially Ally Ip, Christopher Leung, Kenneth Hon, Levina Ha, and Roland Law: Thanks for all the laughs and tears. I will miss you all.

To my friends in MIT HKSS: See you again in Hong Kong!

To my friends at the daily mass in MIT, especially Father Paul Reynolds, Sister Mary Karen Powers, Marissa Long, Michael Lopez, and Sunil Konath: Thanks for your fellowship.

To my friends at Cornell Catholic Community, especially Father Murphy, Angela, Kelvin, Kenneth, and Peter: Thanks for the great fellowship. I will miss you all.

To my friends at Cornell University, especially Alan Yeung and Thomas Tong: Thanks for the many years of friendship.

To my friends at University of Waterloo Catholic Fellowship, especially David, Gus, Ida, and Petrus: It has been a privilege knowing you all. I will miss you a lot.

To my friends at University of Waterloo, especially Ah Sze, Andy Wong, Gabriel Chow, Hannah Chi, Hazel Wong, Jonathan Leung, Kenneth Yip, Patrick Chung, and the "Computer Engineering Chinese Student Association" friends: I will never forget the many hours of studying we had together.

To my friends at Upper Canada College, especially Christopher Yuen, David Fung, Leon Chan, and Victor Fung: You all have been pivotal to my happiness at UCC. Thanks.

To my teachers at Upper Canada College, especially Dr. Moore, Mr. Procuniak, and Mr. Sumner: Thanks for the many encouragements and inspirations.

Table of Contents

List of Figures

Chapter 1

Introduction

Pedigree analysis in genetics is a field in medicine that requires intensive amount of computation. This thesis attempts to build several modules of computational procedures and objects that helps a computer program, Geninfer, to perform the computations necessary in pedigree analysis.

This section aims to provide background in pedigree analysis, justify the need for automated technologies in pedigree analysis, and then explain why bayes net is an appropriate uncertainty model for pedigree analysis. We will also discuss recursive decomposition as an efficient algorithm to analyze bayes net problems.

1.1 Pedigree Analysis

Figure 1.1 shows a typical family pedigree tree. A square represents a male, and a circle represents a female. A darkened symbol represents a person infected with the disease in question.

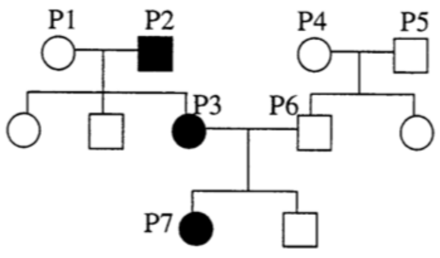

Figure 1.1: A Pedigree Tree

People joined by a horizontal line are married couples, and the people joined directly underneath them are their children. In this example, P1 is married to P2 and give birth to P3; P4 is married to P5 and give birth to P6; P3 is married to P6 and give birth to P7. The probability that a child inherits (or does not inherit) a certain disease from his parents is expressed as P(child | parents), and its value is dependent upon many factors, including the parents' genetic makeup and environmental factors. Gelbart[1] contains a detailed account of the mathematical principles behind pedigree analysis. In general, when finding, say, P(P3=T | P1=T, P2=F), we should also take into account information from other members of the family in order to give an accurate risk analysis.

1.2 The Need for Automated Technologies in Pedigree Analysis

Although the mathematical principles of pedigree analysis are well known and widely advocated, their application in the genetics counseling office is limited because a detailed analysis of the patient's whole family pedigree involves difficult and tedious calculations; therefore, when determining the risk of genetic disorder recurrence, they often solely take into account the information known about the patient's ancestors and ignore possibly valuable information available from knowledge about other members of the family[2]. We anticipate that major research efforts now underway in the Human Genome project will lead to even more information overload, and the genetics counselor of the future will have

[1] J.F. Crow, *Genetics Notes: An Introduction to Genetics* 8th ed, (Macmillan Publishing Co, NY, 1983).

[2] S.P. Pauker, and P. Szolovits. Pedigree Analysis for Genetic Counseling. In Lun, K.C., et al. (eds.), *MEDINFO 92: Proceedings of the Seventh Conference on Medical Informatics,* pages 679-683. Elsevier (North Holland) 1992.

to be able to recognize and offer advice to patients with an even larger variety of disorders – a task almost certain to require assistance from automated technologies. The goal of Geninfer is to develop an intelligent system that could perform such tasks and offer advice that can help Geneticists to carry out accurate pedigree analysis.

1.3 Bayes Net in Pedigree Analysis

The first step in creating an automated tool for pedigree analysis is to find an appropriate computer model to represent pedigree trees. The formalism of Bayesian networks[3] has been successfully employed in the previous version of Geninfer because of its nature in handling uncertainty and conditional probability. As mentioned in section 1.2, the problem of pedigree analysis is essentially an uncertainty evaluation problem. As a result, we can easily transform a pedigree tree into a bayesian network. For example, assume that we are only interested in the phenotype and genotype of a particular family. Then, in creating the bayesian network, we can symbolize each person by two different nodes, each node representing either the phenotype or the genotype of the particular person. Assuming that environmental factor does not exist, then the phenotype of a person is determined solely by the person's genotype, which in turn is determined solely by his parents' genotype. Figure 1.2 is an example of such a transformation.

[3] P. Szolovits, Uncertainty and Decisions in Medical Informatics, *Methods of Information in Medicine* **34** (1995) 111-21.

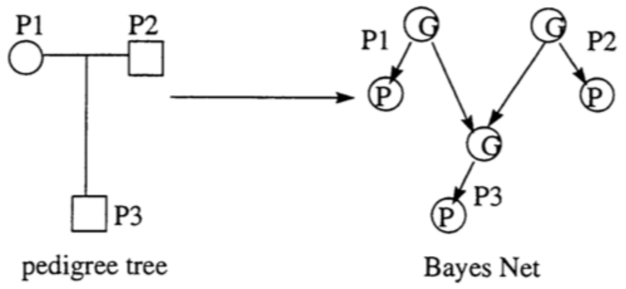

Figure 1.2: Transforming Pedigree Tree to Bayes Net

1.4 Bayes Net Inference Using Recursive Decomposition

Although bayes net provides a reasonable model for pedigree analysis, a straight forward calculation generally requires a running time in the order of 2^n, where n = the number of nodes in a bayes net[4]. For this reason, several algorithms have been developed that significantly reduces the amount of computation to the order of log n. One of these algorithms, which we will employ in our Geninfer program, is Recursive Decomposition.

This section aims to demonstrate the algorithm of recursive decomposition by an example taken from the paper, "*Bayesian Belief-Network Inference Using Recursive Decomposition*", written by Gregory F. Cooper[5]. The essence of the algorithm is the recursive divide-and-conquer nature of the inference method.

Consider figure 1.3, a bayes net with seven variables, namely x1, .., x7.

$$\begin{tikzcd} x7 & x6 \arrow[l] & x5 \arrow[l] & x4 \arrow[l] & x3 \arrow[l] & x2 \arrow[l] & x1 \arrow[l] \end{tikzcd}$$

Figure 1.3: A String Bayes Net

[4] G.F. Cooper, Bayesian Belief-Network Inference Using Recursive Decomposition, KSL-90-05, Stanford University, 1990.
[5] G.F. Cooper, ibid.

Suppose we want to calculate

$$P(x1 = T \mid x7 = T) = P(x1 = T, x7 = T) / P(x7 = T) \quad (1)$$

which can be solved by first calculate

$$P(x7 = T) = \sum_{x} P(x7=T|x6) P(x6|x5) P(x5|x4) P(x4|x3) P(x3|x2) P(x2|x1) P(x1) \quad (2)$$

A straight forward calculation would require 2^6-1 = 63 additions and 6 x 2^6 = 384 multiplication. The recursive decomposition algorithm reduces the amount of computation by separating the network into two sub-networks connected by a d-separator variable[6]. Given the value of a d-separator variable, the variables in each of the disconnected sub-networks are probabilistically independent of the other sub-network. Mathematically, this means we can move the variable outside the summation and thus reduce the amount of computation. More specifically, if we were to choose x4 as the d-separator, then equation 2 would become

$$P(x7 = T) = \sum_{x4} \left[\sum_{\{x5,x6\}} P(x7=T|x6) P(x6|x5) P(x5|x4) \right] \times \left[\sum_{\{x1,x2,x3\}} P(x4|x3) P(x3|x2) P(x2|x1) P(x1) \right] \quad (3)$$

which contains only 20 additions and 66 multiplication. By choosing x4 as the d-separator, we have also divided the network into two sub-networks, {x1, x2, x3} and {x5, x6, x7}, each probabilistically independent of each other. To further simply the calculations, the recursive decomposition algorithm looks for a d-separator variable within each sub-network, and further divide the sub-networks into smaller networks. For

[6] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, (Morgan Kaufmann Publishers Inc, San Mateo, CA, 1988).

example, by choosing x6 and x2 as the d-separator in the sub-networks {x1, x2, x3} and {x5, x6, x7}, we can modify equation 3 to

$$P(x7 = T) = \sum_{x4} [\sum_{x6} [P(x7=T|x6)] [\sum_{x5} P(x6|x5)P(x5|x4)]] \times [\sum_{x2} [\sum_{x3} P(x4|x3)P(x3|x2)] [\sum_{x1} P(x2|x1) P(x1)]] \qquad (4)$$

Equation 4 requires only 15 additions and 30 multiplication, a substantial improvement from equation 2, which requires 63 additions and 384 multiplication. The strategy is to repeatedly apply recursive decomposition algorithm to the network until it is reduced to a network with only one (or two) variables, so that each sub-network contains only a one-step probabilistic calculation.

1.5 Thesis Objectives

This proposed thesis serves as an extension of the previous version of Geninfer written by Professor Peter Szolovits[7]. In that version, bayes net was implemented to carry out pedigree analysis for single-locus two-allele diseases. However, most diseases are multiple-loci with more than two alleles; therefore, one goal of this thesis is to build several modules of computational procedures and objects that helps Geninfer to perform the computations necessary in multiple-loci multiple-allele diseases.

[7] S.P. Pauker, and P. Szolovits. Pedigree Analysis for Genetic Counseling. In Lun, K.C., et al. (eds.), *MEDINFO 92: Proceedings of the Seventh Conference on Medical Informatics,* pages 679-683. Elsevier (North Holland) 1992.

Pedigree analysis involves NP-hard calculations, and extending the scope of diseases from single-locus to multiple-loci causes the computation time to increase exponentially. In addition, the previous implementation of recursive decomposition requires that a new decomposition tree be made whenever a new arc is added or deleted to an existing pedigree tree. This requirement is extremely undesirable because it would cause serious delays between sensitivity analysis, a common practice in genetic counseling. An objective of this thesis is to remedy this problem by extending the recursive decomposition algorithm so that it would alter, instead of destroy, the existing decomposition tree whenever an arc is added or deleted from a pedigree tree. This way, the amount of delay between sensitivity analysis is minimized. This method, known as incremental decomposition, can be extended to cover the case of adding and deleting nodes.

In extending the scope of diseases from single-locus to multiple-loci, we can no longer use probability tables to hold conditional probability values, as we did in the previous version. This is because the required storage space for the probability tables will be too large. For example, consider a two loci disease with two alleles per locus. Each person will have 2^4 possible genotypes, and the probability table that holds the conditional probabilities of child given parents' genotypes will have to hold $2^4 \times 2^4 \times 2^4 = 2^{12}$ entries. Evidently, the amount of required storage space grows exponentially with the increase of number of loci and number of allele. An objective of this thesis is to remedy this problem by developing an equation that will replace the probability table. Replacing the tables with equations would save a lot of memory space, with the inevitable trade-off of speed.

In summary, there are several main objectives in this project:

1. Incorporate incremental decomposition algorithm into existing recursive decomposition algorithm.
2. Replacing probability tables with probability equations.
3. Modify procedures of the previous version of Geninfer that assume one-dimensional values.

Chapter 2

Extending the Recursive Decomposition Algorithm – Incremental Decomposition

2.1 Objective

In the previous version of Geninfer, the whole decomposition tree is reconstructed after an arc is added or deleted to the pedigree tree. The objective of this sub-section is to extend the RD algorithm so that it would alter, instead of destroy, the existing decomposition tree whenever an arc is added or deleted between a pair of nodes. This is desirable because constructing a decomposition tree is a very time-consuming process, and adding or deleting an arc is a very common task performed by genetics counselors. The strategy that we use for adding and deleting arcs can be employed for adding and deleting nodes as well.

2.2 Solution

In order to achieve this goal, we must first understand how adding an arc to a pedigree tree alters the decomposition tree. Consider the example in figure 2.1:

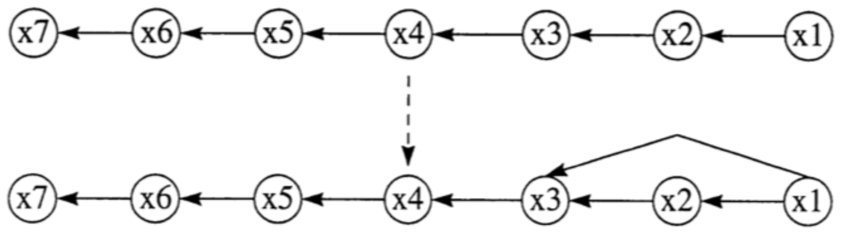

Figure 2.1: Adding an Arc

The original string bayes net has the following decomposition tree:

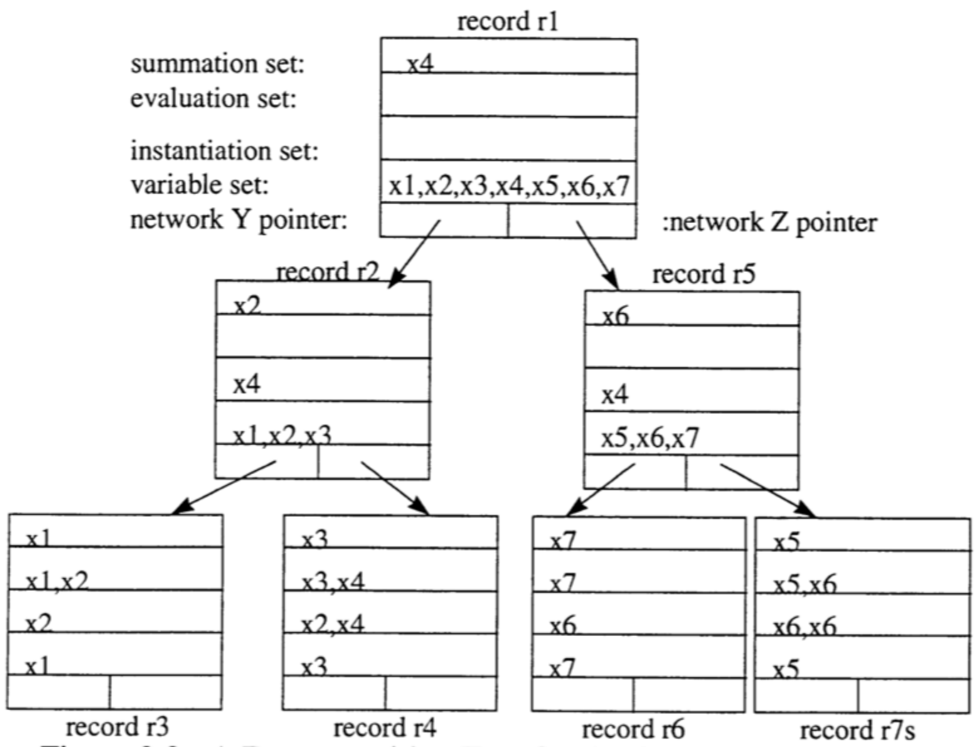

Figure 2.2: A Decomposition Tree for the Original Bayes Net in Figure 2.1

which represents equation 4. After an arc is added between x1 and x3, only a portion of the decomposition tree is affected. According to Cooper[8], that portion is the deepest record in the decomposition tree that contains both x1 and x3 in its variable set, which we shall call rj, as well as other records that are pointed to by rj. In this example, that rj is r2. This is because the record represents the smallest sub-network that contains both x1 and x3, and the arcs between them, if any. Therefore, after an arc is added to x1 and x3, only record r2, and all the records that it points to, should be reconstructed. The sub-network of the decomposition tree that is represented by r1, r5, r6 and r7 should not be affected. This reasoning is true also in the case of deleting an arc between the nodes x1 and x3, and in general, any two nodes.

It should be noted that any record that points to rj also represents a sub-network that contains both x1, x3, and any arc between them. However, since we want to minimize the amount of redissection of the decomposition tree, we should look for the smallest sub-network that contains both nodes, which is record rj.

In order to achieve our goal of minimizing the portion of the decomposition tree that needs to be reconstructed, we should first identify the deepest record in the decomposition tree that contains both nodes in its variable set. We then redissect only the sub-network represented by that record and the records pointed to by it. This method, suggested by Cooper[9], is called incremental decomposition, and its implementation will be discussed in section 2.3 Implementation.

---

[8] G.F. Cooper, Bayesian Belief-Network Inference Using Recursive Decomposition, KSL-90-05, Stanford University, 1990.
[9] G.F. Cooper, Bayesian Belief-Network Inference Using Recursive Decomposition, KSL-90-05, Stanford University, 1990.

The incremental decomposition method can also be used when a node is added or deleted from a bayes net. In this type of problem, we also look for the portion of the decomposition tree that represents the smallest sub-network that contains the part of the bayes net that is affected by the addition or deletion of the node, namely the deepest record (call it rj again) in the decomposition tree that would contain the involving nodes in its variable set. The involving nodes are the added/deleted nodes and the nodes linked with them. We should also modify the records that is above rj in the decomposition tree in the following scheme: if a node is being added to a bayes net, then it should also be added to the variable set of all records above rj; if a node is being deleted, then it should also be deleted from the variable set of all records above rj.

A drawback of incremental decomposition is that the resulting redissected decomposition tree would not necessary be the best tree that would result from redissecting the whole bayes net. This is because a portion of the dissection tree (the ones not affected by the redissection) must remain unchanged; otherwise, the essence of incremental decomposition (saving time) is lost, and we may as well redissect the whole decomposition tree.

Aside from the introduction of incremental decomposition algorithm, more can be done to minimize the amount of computation time involved in adding or deleting an arc or a node to a bayes net. In particular, consider the case when a user adds an arc between two nodes and then decides to delete the arc. If no probability calculation is done between the adding and deleting of the arc, then the structure of the decomposition tree should not be changed at all. For this reason, we have introduced the slots *added-arcs*, *deleted-arcs*, *added-elements*, and *deleted-elements* to the class *Cooper-BNET-equation-mixin*. Their functions will also be discussed in section 2.3 Implementation.

2.3 Implementation

Here is the new definition of Cooper-BNET-equation-mixin:

```
(defclass Cooper-BNET-equation-mixin ()
  ((dissection-tree :accessor BNET-dissection-tree :initform nil)
   (compiled-evaluation-function :accessor BNET-compiled-evaluation-function
                         :initform nil)
   (added-elements :accessor added-elements :initform nil)
   (deleted-elements :accessor deleted-elements :initform nil)
   (added-arcs :accessor added-arcs :initform nil)
   (deleted-arcs :accessor deleted-arcs :initform nil))
  (:default-initargs :element-type 'Cooper-RV-equ))
```

Whenever a user adds or deletes an arc, the functions add-arc and delete-arc will be called. Similarly, add-element and delete-element will be called when a node is added or deleted from a bayes net. We now add the functions *add-arc :after*, *delete-arc :after*, *add-element :after*, and *delete-element :after* as defined below:

```
defmethod add-arc :after ((CBNET Cooper-BNET) (source set-index) (target set-index))
  ;; modifies:   (added-arcs CBNET), (deleted-arcs CBNET)
  ;; effects:    If (null-dissection-tree CBNET), then don't do anything; otherwise, let
  ;;             arc=arc from source to target.
  ;;             #1. If arc is in (deleted-arcs CBNET), then remove arc from (deleted-arcs
  ;;             CBNET).
  ;;             #2. If arc is not in (deleted-arcs CBNET), then push arc into (added-arcs
  ;;             CBNET).

defmethod delete-arc :after ((CBNET Cooper-BNET) (source set-index)
                              (target set-index))
  ;; modifies:   (added-arcs CBNET), (deleted-arcs CBNET)
  ;; effects:    If (null-dissection-tree CBNET), then don't do anything; otherwise, let
  ;;             arc=arc from source to target.
  ;;             #1. If arc is in (added-arcs CBNET), then remove
  ;;             arc from (added-arcs CBNET).
  ;;             #2. If arc is not in (added-arcs CBNET), then push arc into (deleted-arc
 ;;               CBNET).

defmethod add-element :after (element (CBNET Cooper-BNET))
 ;; modifies:   (added-elements CBNET), (deleted-elements CBNET)
 ;; effects:    If (null-dissection-tree CBNET), then don't do anything; otherwise,
 ;;             #1. If element is in (deleted-elements CBNET), then remove element from
 ;;             (deleted-elements CBNET).
 ;;             #2. If element is not in (deleted-elements CBNET), then push element into
 ;;             (added-element CBNET).

defmethod delete-element :after (element (CBNET Cooper-BNET))
 ;; modifies:   (added-elements CBNET), (deleted-elements CBNET)
 ;; effects:    If (null-dissection-tree CBNET), then don't do anything; otherwise,
 ;;             #1. If element is in (added-elements CBNET), then remove element from
 ;;             (added-elements CBNET).
 ;;             #2. If element is not in (added-elements CBNET), then push element into
 ;;             (deleted-element CBNET).
```

With the introduction of these new functions, whenever an arc is being added to a bayes net (call it CBNET), the program first check if the same arc can be found in (deleted-arcs CBNET). If not, then an arc will be placed in the slot (added-arcs CBNET); otherwise, the arc will be taken out of (deleted-arcs CBNET). Notice that the structure of the CBNET decomposition tree is not being changed yet; only an additional arc is being placed in the (added-arcs CBNET) slot. Similarly, when the user deletes an arc, and the same arc could not be found in (added-arcs CBNET), then the arc will be stored in (deleted-arcs CBNET); otherwise, the arc will be taken out of (added-arcs CBNET). Again, the decomposition tree of CBNET is not altered yet. When the user has completed all the additions and deletions of arcs and wants to perform a calculation, he would call the function *infer*, defined in Cooper.Lisp. The program will look at the added-arcs and deleted-arcs slots, determine the changes that need to be made according to those slots, and the decomposition tree will be altered according to the incremental decomposition algorithm, as described in section 2.2. The slots (added-arcs CBNET) and (deleted-arcs

CBNET) will then be emptied. The usefulness of this scheme is demonstrated in the following example: if a user adds an arc between x1 and x3, and decides to delete the arc before performing any calculations, then effectively no arc will be added to the slots (added-arcs CBNET) and (deleted-arcs CBNET). When the program performs its next calculation, it will notice that effectively no modification is made on the bayes net, and thus would not redissect the decomposition tree at all.

The functions and usefulness of *added-elements* and *deleted-elements* are identical to added-arcs and deleted-arcs, except that they deal with the addition and deletion of nodes. In essence, their presence prevent the decomposition tree from being altered if an element is added and then deleted, and vice versa.

If the procedure infer is called, and any of the added-arcs, deleted-arcs, added-elements, or deleted-elements slots is not empty, then procedures *mark-records* and *incremental-decompositions* will be called. The specifications of these procedures are shown below:

```
 (defun mark-records (BNET &optional (make-cache-array-now? t))
  ;; requires: (delete-arc x y BNET) or (delete-arc z x BNET) where y = all
  ;;           children of x and z = all parents of x must be called before
  ;;           (delete-element x BNET) is called.  This can easily be
  ;;           accomplished by having the caller of delete-element checks all
  ;;           connections of x before calling delete-element, and this
  ;;           requirements saves a lot of computation time and allows simpler
  ;;           algorithms to be used in this function.
  ;; modifies: BNET
  ;; effects:  Marks the records.
  ;;             Clears the record of added- and deleted-arcs and -elements.

(defun incremental-decompositions (BNET)
  ;; modifies: BNET
  ;; effects:  Sets up the markov blanket and neighborhood size of every nodes.
  ;;           Dissects the sub-networks corresponding to the records in
  ;;           the decomposition tree of BNET that are marked, according
;;        to the incremental decompositions method.
;;        Assign records to nodes in their variable sets.
```

As can be seen from the descriptions, procedure mark-records and incremental-decompositions carry out the incremental decomposition algorithm. It should be noted that, in the previous version of Geninfer, callers of (add-arc CBNET source target) and (delete-arc CBNET source target) often need to call (dissect CBNET) as well. With the introduction of incremental decomposition and the changes mentioned in this section, it is no longer necessary to call (dissect CBNET) after any addition or deletion of arcs.

2.4 Testing

According to Guttag and Liskov[10], many errors can be discovered by a path-complete black box and glass box test. However, due to the large size of this program, it is not feasible to perform this kind of testing. Instead, we will perform a path-complete test only for the procedures described in section 2.3 because of their significance in the incremental decomposition algorithm. These test cases involve adding and deleting an arc, and adding and deleting a node.

In the first test case, we built a string bayes net with seven variables x1 .. x7. A decomposition tree identical to the one shown in figure 2.2 was built by the program. We then added an arc between x1 and x3, just like the scenario depicted in figure 2.1. When prompted to perform a calculation, the program first marked r2 as the record that needed to be redissected, and then modified the decomposition tree to the one shown in figure

[10] J. Guttag, and B. Liskov, *Abstraction and Specification in Program Development*, (The Massachusetts Institute of Technology, MA, 1995)

2.3. A comparison of figure 2.2 and 2.3 showed that only the sub-network represented by r2, r3 and r4 was being affected. The sub-network represented by r1, r5, r6 and r7 remained unaffected, showing that the incremental decomposition algorithm was being implemented correctly. The calculation results were also accurate. The actual code for this test case was included in Appendix A.

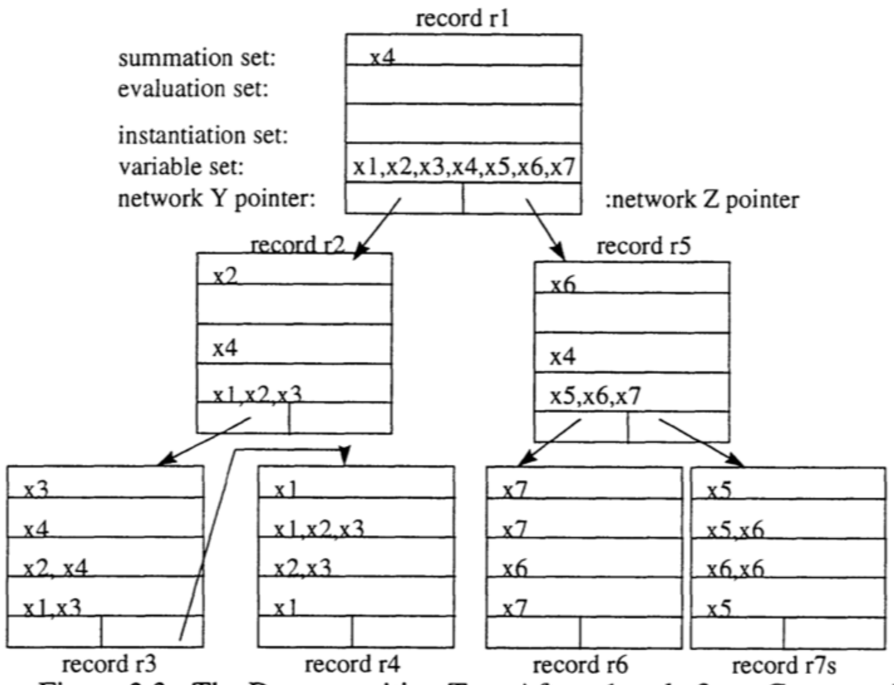

Figure 2.3: The Decomposition Tree After x1 and x3 are Connected

Another example involved deleting an arc from a bayes net. We again used the same scenario as in figure 2.1, except that the direction of change was reversed i.e. at first an arc existed between x1 and x3, then it was being deleted. The actual testing code was a slight variation from that of the first example. A decomposition tree was modified from that in figure 2.3 to that in figure 2.2, once again showing that only the relevant part of the decomposition tree was redissected. The calculation result was also correct.

In the third example, a node was being deleted from a bayes net, as shown in figure 2.4:

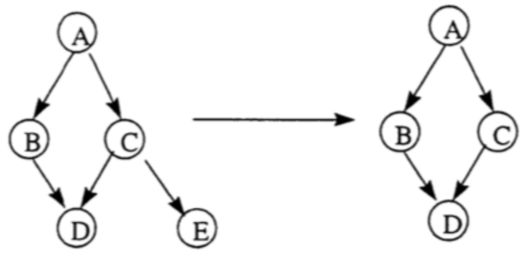

Figure 2.4: Deleting a Node from a Bayes Net

The calculation results performed before and after the node deletion were accurate. In the fourth example, node E was added to the bayes net, which was essentially the reverse of the scenario depicted by figure 2.4. The calculations performed before and after the node addition were also accurate, reassuring us that the program worked well with the addition and deletion of nodes as well as with the addition and deletion of arcs. The actual testing codes for the third example can be found in Appendix A. The actual codes for the forth example was a slight variation from that of the third example.

Chapter 3

Replacing Probability Tables with Probability Equations

3.1 Objective

In the previous version of Geninfer, probability values were first calculated and then stored in probability tables that were attached to the nodes. With the expansion of this project to multiple dimensional values, this scheme is no longer feasible, because the size of the table needed would be too large and requires too much disk space (see section 1.5). The main goal of this section is to replace the probability tables with probability equations. We will clearly lose the advantage of using tables, which is the ability to look up a value quickly, but it is a necessary tradeoff.

When considering multiple loci diseases, we should realize that not all genes behave independently, as predicted by Mendel's Second Law. The law applies when the genes are on nonhomologous chromosomes, but when two genes are on the same chromosome they tend to stay together in inheritance, a phenomenon known as linkage. Linkage can be illustrated in the following example, taken from Crow[11].

Consider two pairs of alleles on the same chromosome that were involved in a crossing:

C: creeper
c: normal length legs
R: rose comb
r: single comb

A test-cross between a homozygous rose-combed, normal-legged strain with a single-combed, creeper strain gave the following result:

Figure 3.1: Progeny from Rc/rC x rc/rc

	$\frac{R\ c}{r\ C}$ x $\frac{r\ c}{r\ c}$	
$\frac{R\ c}{r\ c}$	1069	99.5% nonrecombinants
$\frac{R\ c}{r\ C}$	1104	
$\frac{R\ c}{r\ C}$	6	0.5% recombinants
$\frac{R\ c}{r\ C}$	4	

This example demonstrated that genes located on the same chromosome tend to remain together in inheritance; they are linked. However, they were not completely linked, and could be separated by the process of crossing over. This process occurred during meoisis, when homologous chromosomes lined up side by side during synapsis. The whole process of crossing over could be diagramed as follows:

[11] J.F. Crow, *Genetics Notes: An Introduction to Genetics* 8th ed, (Macmillan Publishing

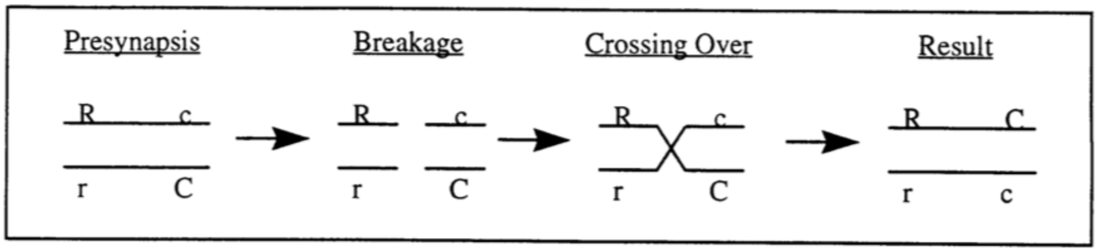

Figure 3.2: The four stages of Crossing Over

After crossing over, the R and c genes, and the r and C genes, which were on the same strand, are now uncoupled and two types of recombinant strands, RC and rc, have been produced. By considering the number of recombinants, a genetic map can be constructed by a process known as chromosome mapping. The map shows the distance between different genes by their recombination frequency. In the above example, the distance between genes R and C is 0.05m.u., where m.u. stands for map units.

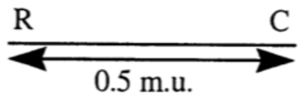

Figure 3.3: Genetic map of R and C genes

Crossing over is of great significance in genetics for several reasons. If crossing over did not occur, genes on the same chromosome would be inseparably linked. If a beneficial gene were linked to a deleterious one there would be no way to uncouple them. Likewise, if two beneficial genes were on homologous chromosomes, there would be no way for them to get into the same chromosome. A breeder who wanted to produce an animal or plant homozygous for both desirable genes would be out of luck. He would have to wait for mutation, a very slow way of effecting change. In the evolutionary history of a species, crossing over is important because it permits genes on the same or hologous chromosomes to be shuffled in the same way that genes on independent

Co. NY, 1983)

chromosomes are. It extends the benfit of sexual reproduction, or recombination, to genes on the same chromosome pair.

3.2 Solution

The mathematical principles of pedigree analysis are well versed and thoroughly discussed by scientists such as Crow[12] and Gelbart[13]. However, no formal mathematical equation has been established. In this section, we attempt to derive an equation that can be implemented in a computer program. Our goal is to calculate the probability that a child has genotype cg given his parents' genotypes i.e. P(child = cg | father, mother). The child inherits his paternal genotype from his father and his maternal genotype from his mother. Those genotypes are determined by the genetic makeup of his parents' sex cells that, when fused, created the child. Assume that the formation of the father's and mother's sex cells are independent processes, then:

$$\begin{aligned} & P(\text{child} = cg \mid \text{father, mother}) \\ = \quad & P(\text{father's sex cell's genotype} = \text{paternal genotype of } cg) \times \\ & P(\text{mother's sex cell's genotype} = \text{maternal genotype of } cg) \end{aligned} \qquad (5)$$

The next step is to determine P(father's sex cell's genotype = paternal genotype of cg). P(mother's sex cell's genotype = maternal genotype of cg) can be derived in a similar way. When the father's sex cell is formed, it is equally likely to contain his paternal or maternal genotype. Therefore,

$$P(\text{father's sex cell's genotype} = \text{paternal genotype of } cg)$$

[12] J.F. Crow, *Genetics Notes: An Introduction to Genetics* 8th ed, (Macmillan Publishing Co, NY, 1983).
[13] W.M. Gelbart, A. Griffiths, R.C. Lewontin, J.H. Miller, and D.T. Suzuki, *An Introduction to Genetic Analysis* 5th edition, (W.H. Freeman & Company, NY, 1993)

= P(child inherit father's paternal genotype) x P(father's paternal genotype = paternal genotype of cg) +
P(child inhierit father's maternal genotype) x P(father's maternal genotype = paternal genotype of cg)
= 0.5 x P(father's paternal genotype = paternal genotype of cg) +
0.5 x P(father's maternal genotype = paternal genotype of cg) (6)

The next step is to derive P(father's paternal genotype = paternal genotype of cg. P(father's maternal genotype = paternal genotype of cg) can be derived in a similar way.

The following tree shows all the possible genotypes of resulting paternal genotype (designed by the bold line) of the father after counting all possible crossing overs.

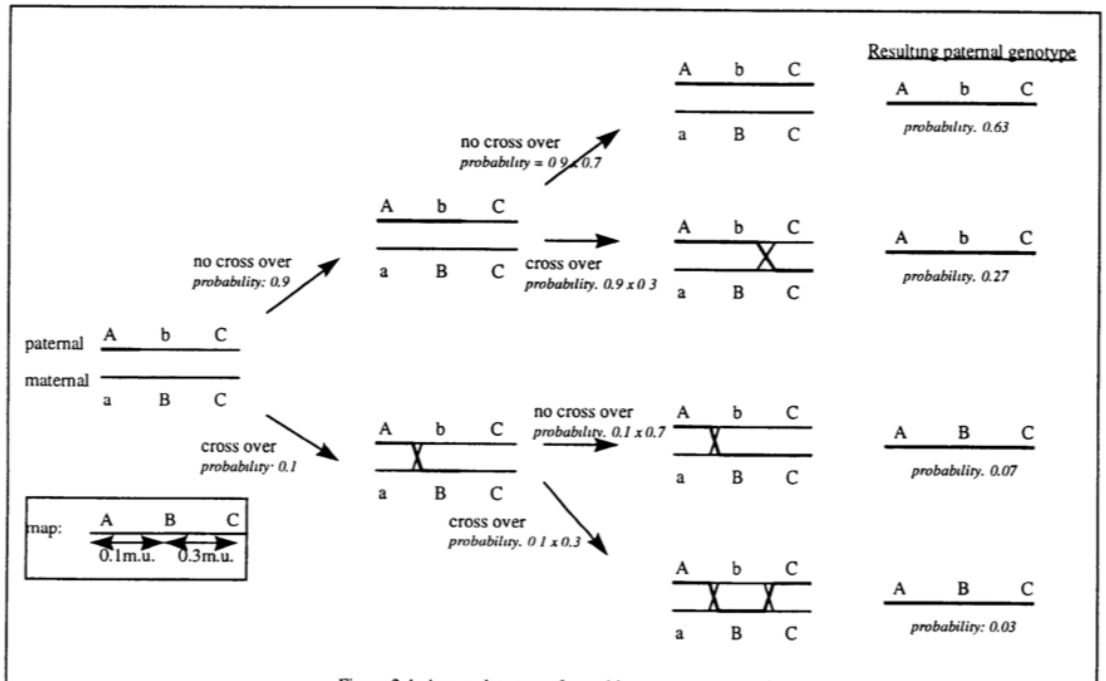
Resulting paternal genotype
paternal
maternal
A b C
a B C
no cross over probability: 0.9
cross over probability: 0.1
no cross over probability = 0.9 x 0.7
cross over probability. 0.9 x 0.3
no cross over probability. 0.1 x 0.7
cross over probability. 0.1 x 0.3
probability. 0.63
probability. 0.27
probability. 0.07
probability: 0.03
map: A B C
0.1m.u. 0.3m.u.

Figure 3.4: A complete tree of possible crossing over results

From the tree, it can be inferred that,

P(father's paternal genotype = paternal genotype of cg)
= P(1st gene of father's paternal = 1st gene of cg's paternal) x
P(all other genes of father's paternal = all other genes of cg's paternal)
= P(1st gene of father's paternal = 1st gene of cg's paternal) x
{ P(cross over) x P(all other genes of father's paternal = all other genes of cg's paternal) +
P(no cross over) x P(all other genes of father's paternal = all other genes

of cg's paternal) }

= P(1st gene of father's paternal = 1st gene of cg's paternal) x
{ P(cross over between 1st and 2nd gene) x
P(2nd gene of father's paternal = 2nd gene of cg's paternal) x
[P(cross over between 2nd and 3rd gene) x
P(all other genes of father's paternal = all other genes of cg's paternal) +
P(no cross over between 2nd and 3rd gene) x
P(all other genes of father's paternal = all other genes of cg's paternal)] +
P(no cross over between 1st and 2nd gene) x
P(2nd gene of father's paternal = 2nd gene of cg's paternal) x
[P(cross over between 2nd and 3rd gene) x
P(all other genes of father's paternal = all other genes of cg's paternal) +
P(no cross over between 2nd and 3rd gene) x
P(all other genes of father's paternal = all other genes of cg's paternal)]} (7)

The equation further expands to 3rd, 4th, 5th, etc, gene, until all genes have been covered.

In other words,

P(father's paternal genotype = paternal genotype of cg)
= P(i^{th} to n^{th} gene of father's paternal = i^{th} to n^{th} gene of cg's paternal) (8)

for i = 1, and n = number of genes in question. (8) can also be written as:

P(i^{th} to n^{th} gene of father's paternal = i^{th} to n^{th} gene of cg's paternal)
= P(i^{th} gene of father's paternal = i^{th} gene of cg's paternal) x
{ P(cross over between i^{th} and $(i+1)^{th}$ gene) x
P($[i+1]^{th}$ to n^{th} gene of father's paternal = $[i+1]^{th}$ to n^{th} gene of cg's paternal) +
P(no cross over between i^{th} and $(i+1)^{th}$ gene) x
P($[i+1]^{th}$ to n^{th} gene of father's paternal = $[i+1]^{th}$ to n^{th} gene of cg's paternal) } (9)

where P($[i+1]^{th}$ to n^{th} gene of father's paternal = $[i+1]^{th}$ to n^{th} gene of cg's paternal) is simply the same as (8) with i replaced by [i+1]. Clearly, this is a recursive function, and can be best implemented by a recursive procedure. Note that P(i^{th} gene of father's paternal

= i^{ith} gene of cg's paternal) is not simply a 1 or 0 probability if we taken into account mutation rates. To be more specific,

```
P(i^th gene of father's paternal = i^ith gene of cg's paternal)
=       if i^th gene of father's paternal = i^ith gene of cg's paternal
        then answer  = probability of no mutation at i^th gene
                     = 1 - all mutation rates at i^th gene
        else answer  = mutation rate from father's allele to cg's allele          (10)
```

Combining equation (5)-(10), we get:

```
P(child = cg | father = fg, mother = mg)
=    P(father's sex cell's genotype =   paternal genotype of cg) x
       P(mother's sex cell's genotype = maternal genotype of cg)
=    { 0.5 x P(father's paternal genotype = paternal genotype of cg) +
       0.5 x P(father's maternal genotype = paternal genotype of cg) } x
     { 0.5 x P(mother's paternal genotype = maternal genotype of cg) +
       0.5 x P(mother's maternal genotype = maternal genotype of cg) }
=    { 0.5 x
       P(1^th to n^th gene of father's paternal = 1^th to n^th gene of cg's paternal) +
       0.5 x
       P(1^th to n^th gene of father's maternal = 1^th to n^th gene of cg's paternal)} x
     { 0.5 x
       P(1^th to n^th gene of mother's paternal = 1^th to n^th gene of cg's maternal) +
       0.5 x
       P(1^th to n^th gene of mother's maternal = 1^th to n^th gene of cg's maternal)}
                                                                                   (11)
```

where P(1th to nth gene of father's/mother's paternal/maternal = 1th to nth gene of cg's paternal/maternal) can be determined from equation (9), and n = number of genes in question.

A special case arises when the person is at the top of the pedigree tree. In that case, we cannot compute P(child | father, mother), because no information about his parents' genotype is available. There are two ways to handle this problem. The first requires that we have information about the population genetics of all the genes involved.

Assuming that having one particular allele at a gene has no effect on the probability of having a certain allele at another gene (i.e. independent events), then:

$$P(\text{child} = \text{cg}) = \prod p_{ij} \qquad (12)$$

where p_{ij} = probability of having allele i at gene j. Another method is to assume that all alleles are equally likely to occur in the population. We again make the assumption that having one particular allele at a gene has no effect on the probability of having a certain allele at another gene. (12) then becomes:

$$P(\text{child} = \text{cg}) = \prod p_J \qquad (13)$$

where p_j = (number of possible alleles at gene j)$^{-1}$. In my work, I will adopt (13) because currently there is no data representation that holds allele frequency.

The implementation of an equation to calculate the probability of having a certain genotype is possible and straight forward, since the same methodology applies to any person, regardless of sex and ethnicity. The implementation of an equation to calculate the probability of a certain phenotype is not so simple, because the relationship among genotype and phenotype varies for different diseases. Therefore, at this stage of the project, it is not possible to write a universal equation that would work for phenotypes of all kinds of diseases.

3.3 Implementation

The following classes are defined to allow the use of probability equations in place of probabilities tables:

```
(defclass conditional-probability-equation-mixin ()
        ((conditional-probability-equation :accessor conditional-probability-equation
                              :initarg :conditional-probability-equation
                              :initform nil))
        (:documentation "Equation for conditional probability for discrete-RV"))
(defclass equation-DRV (conditional-probability-equation-mixin discrete-RV)
        ())
(defmethod conditional-probability ((node conditional-probability-equation-mixin)
                              (bayesnet Discrete-Bayes-Net))
        (funcall (conditional-probability-equation node) node bayesnet))

(defclass Cooper-RV-equ (Cooper-mixin equation-DRV))
(defclass Cooper-BNET-equation-mixin ()
(defclass Cooper-BNET-equation (Cooper-BNET-equation-mixin Discrete-Bayes-Net))
```

The purpose of these definitions is to introduce a conditional-probability-equation slot in every node to hold the probability equation. Each of these classes and methods are analogies of classes or methods in previous version of Geninfer that use probability tables. For example, *conditional-probability-equation-mixin* resembles conditional-probability-table-mixin; *equation-drv* resembles table-drv; *cooper-rv-equation* resembles cooper-rv; *cooper-bnet-equation-mixin* resembles cooper-bnet-mixin; *cooper-bnet-equation* resembles cooper-bnet.

The next step is to introduce the objects *genotype*, *geneticmaptype*, *genetype*, *mutationtable*, and *mrates* as defined below. These objects represent the input to the probability functions to be described later.

<u>OBJECT: GENOTYPE</u>

```
;; this is the object for genotype.  In a pedigree tree, (known-vals child bayesnet)
;; will return the child's genotype.  The purpose of this object is to represent the actual
;; genotype that the child inherits from his parents.  The rep is simple-vector.
;; Recall that a child's genotype consists of his paternal and maternal genotypes, which are
;; the genes that he inherited from his father and mother respectively.  If the genotype
;; describes n loci/genes, then the length of the structure, say g, will be 2n.  g[0]..g[n-1]
;; represent the paternal genotype, while g[n]..g[2n] represent the maternal genotype.
;;
;; We have chosen simple-vector as the rep of genotype because of its simplicity and the
;; relatively quick speed in referencing its content.  When making a genotype object, one
;; should call (setf g (make-genotype ....)).
```

```
(defun make-genotype (dimensions initial-contents element-type)
  ;; requires: initial-contents is passed as a list of initial contents, with
  ;;          the first content as the first element of initial-contents, etc.
  ;; effects:  Returns a genotype object with dimension dimensions, initial
  ;;          contents initial-contents, and element type element-type.

(defun num_genes (g)
  ;; effects:  Returns the number of loci/genes that g describes.

(defun ith-gene (g i origin)
  ;; effects:  If origin='paternal, then returns the paternal allele of the gene indexed by i in g
i.e. i-th gene in g.  Similar
  ;;            for origin ='maternal.

(defun half-genotype (g origin)
  ;; effects: if origin=paternal, then returns paternal genotype of g.
  ;;           if origin=maternal, then returns maternal genotype of g.

(defun maternal (g)
  ;; effects: Returns the maternal genotype of g

 (defun paternal (g)
  ;; effects: Returns the paternal genotype of g
```

OBJECT: GENETYPE

```
;; The object for a gene.  Recall that a genotype contains many genes.  The object
;; GENOTYPE therefore is a simple vector of type GENETYPE.
;; The rep of gene is integer i.e. 0, 1, 2, etc, represent different alleles of
;; of the same gene.  alleleunknown is the allele that stands for unknown.
;; allelenull is the allele that stands for null i.e. a point deletion.
;;
;; We have chosen to use integer to represent genes because of the frequent need to
;; compare genes.  When calculating probabilities in pedigree analysis, we often need to
;; consider whether a certain gene of the child has the same allele as the child's parents.  In
;; that case, we can accomplish the task by a simple equal (=) operation.

(defconstant alleleunknown kunknownval "Unknown allele/gene value")
(defconstant allelenull    -2          "Null allele/gene value")
(deftype genetype () '(integer))
(defun same-allele? (allele1 allele2) (= allele1 allele2))
```

OBJECT: GENETICMAPTYPE

```
;; The rep of geneticmaptype is simple-vector.  It is chosen to be an array for its simplicity.
;; This is the object for the map that stores distances between genes.
;; For example, when making a map, one should call (setf map (make-geneticmaptype
;; ......)
;;
;; The information in GENETICMAPTYPE is important not only because it provides the
;; recombination frequencies, but also because it tells which genes are on separate
;; chromosomes.  A map distance of 0.5 m.u. implies that the two genes are either on
;; different chromosomes (likely), or that they are very far apart on the same chromosome.
;; Note that it does not make sense to have recombination frequency higher than 0.5.

(defclass geneticmaptype ()
  ((geneticmap :accessor geneticmap :initarg :geneticmap)))

(defun make-geneticmaptype (length initial-contents element-type)
  (make-instance 'geneticmaptype
    :geneticmap (make-array length :initial-contents initial-contents
                            :element-type     element-type)))

 (defmethod mapdistance ((map geneticmaptype) i j))
  ;; effects:  Returns the genetic map distance between gene i and j according
  ;;           to map.

(defmethod set-mapdistance ((map geneticmaptype) i distance))
  ;; modifies: map
  ;; effects:  set map distance between locus i and i+1 on map to be distance.
```

OBJECT: MUTATIONTABLE

```
;; it is a table that holds the mutation rates from the parents' alleles to the child's alleles
;; Figure 3.5 shows a graphical representation of this object.
;; (paternal-rates mutationtable) and (maternal-rates mutationtable) are MRATES objects.
;; The former contains mutation rates for the paternal genotype, while the later the rates
;; for the maternal genotype.

(defclass mutationtable ()
 ((paternal-rates :accessor paternal-rates :initarg :paternal-rates)
  (maternal-rates :accessor maternal-rates :initarg :maternal-rates)))
```

OBJECT: MRATES

```
;; MRATES is a list of (i ( (allele1 . mutation rate from allele1 to child's allele)
;;                          (allele2 . mutation rate from allele2 to child's allele))
;; where  i = the index that identified a gene e.g. the index in the object GENOTYPE
;;        and allele1, allele2 = the parental, maternal alleles of the corresponding parent's
;;        gene in question.
;; For example, if the gene in question has the index i = 3 in the GENOTYPE simple
;; vector representation, and the corresponding parent is the father, then a possible
;; MRATES is:
;;        (3     ( (2 . 0.003)  (4 . 0.004) ) )
;;  This means that for the gene in question, the father's paternal allele is represented by 2,
;;  and the father's maternal allele is 4.  The mutation rate from the father's paternal allele
;;  to the child's allele is 0.003, while mutation rate from the father's maternal allele to the
;;  child's allele is 0.004.


(defclass mrates ()
  ((mrates-aux :accessor mrates-aux :initform :initarg :mrates-aux)))

(defmethod add-mrates-aux ((mutationrates mrates) genenumber rates)
  ;; effects: add the mutation rates from child's allele to parents' allele to mrates-var
  ;; mrates-var is a mrates object
  ;; genenumber = the index that is used to identify the gene in question
                      the "i" in ith-gene
  ;; rates is created by make-mrates.

(defun make-rates (ppair mpair)
  ;; requires: sum of all the rates less than or equal to 1.
  ;; effects: returns an alist of two pairs.
  ;;          The first pair is (allele1 . mutation rate from allele1 to child's allele)
  ;;          The second pair is    (allele2 . mutation rate from allele2 to child's allele)
  ;;          where allele1, allele2 = the paternal, maternal alleles of the parent in question.

(defmethod mutationrate ((mutationrates mrates) genenumber allele)
  ;; effects: return the mutation rate that the gene identified by genenumber mutated from
  ;;          allele "allele" to the child's allele.
```

For purpose of clarity, the diagram below explains the structure of MUTATIONTABLE:

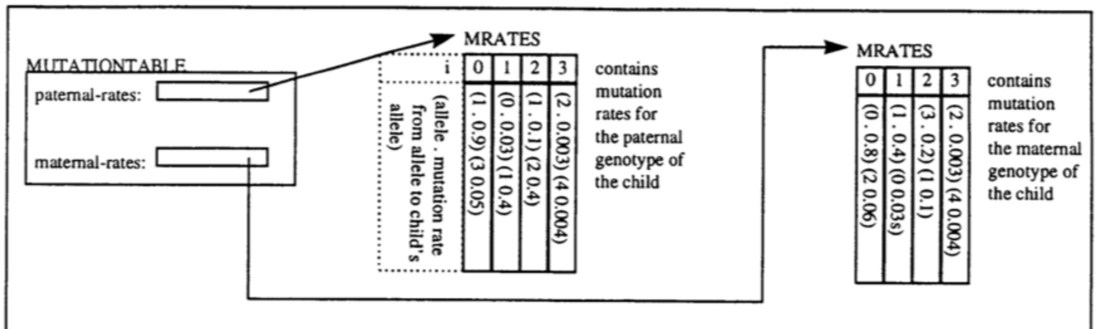

Figure 3.5: A sample MUTATIONTABLE structure

After defining the necessary objects, we are now ready to introduce the probability equations that will be used to replace the probability tables:

EQUATIONS

```
(defun root-genotype-probability (node))
  ;; effect: returns the probability of node having its value given that it is the root of the
  ;;           pedigree tree.

(defun genotype-probability (fgenotype mgenotype cgenotype map mutationtable))
  ;; fgenotype,mgenotype,cgenotype = father's,mother's,child's genotypes.
  ;;                                          They are of type GENOTYPE.
  ;; map =  stores distances between genes.  It is of type GENETICMAPTYPE.
  ;; mutationtable is of type MUTATIONTABLE.
  ;; requires: pgenotype and cgenotype must describe the same number of genes,
  ;;              and must describe at least 1 gene.
  ;; effect: Returns probability of child's genotype=cgenotype given father's and mother's
  ;;           genotype = fgenotype,mgenotype respectively.  Halt with error message if not
  ;;           all genotypes have the same number of genes.

(defun genotype-probability-aux (pgenotype cgenotype child_pORm map mutationrates)
  ;; helper function for genotype-probability.
  ;; pgenotype, cgenotype = the corresponding parent's and the child's genotype.
  ;; map = stores distances between genes.  It is of type GENETICMAPTYPE.
  ;; mutationrates is of type MRATES.
```

The *EQUATIONS* functions work this way:

(root-genotype-probability node) = $\prod p_j$
where p_j = (number of possible alleles at gene j)$^{-1}$.

```
(genotype-probability fgenotype mgenotype cgenotype map mutationtable))
=     P(child = cgenotype | father = fgenotype, mother = mgenotype,
                              genetic map = map, mutation table = mutationtable)
=     P(father's sex cell's genotype = paternal genotype of cgenotype) x
      P(mother's sex cell's genotype = maternal genotype of cgenotype)

(genotype-probability-aux pgenotype cgenotype child_pORm map mutationrates)
=     P(father/mother's sex cell's genotype = paternal/maternal genotype of
        cgenotype)
=     {  0.5 x P(ith to nth gene of father's/mother's paternal = ith to nth gene of
            cgenotype's paternal/maternal) +
         0.5 x P(ith to nth gene of father's/mother's maternal = ith to nth gene of
            cgenotype's paternal/maternal)}
note:    father/mother is determined by whether pgenotype = father's or mother's
         genotype, while paternal/materal is determined by child_pORm ('paternal,
         'maternal).
```

As can be seen, these functions perform the calculations for equations (5)-(13). The caller to these functions have to construct MUTATIONTABLE before hand. I assume that geninfer will have a data representation that contains all possible mutation rates at all genes. Therefore, MUTATIONTABLE, or MRATES, can be constructed by picking out the relevant mutation rates from that data representation.

The function *genotype-probability* will count on its callers to supply the GENOTYPES. Since the parents' genotypes are part of the parameters, it had been suggest that a function should be written to return a node's parents' genotypes. However, the user of this module, MIT Research Affiliate Sean Doyle, has expressed his preference to hard code those information into the equations themselves. We have agreed that it is an appropriate approach, but if the genotype of a certain node's parents change, then the user of this module should remember to modify the conditional-probability-equation of that node as well, since the node's equation carry the parents genotypes. Note that, if not handled properly, this might lead to a problem in ensuring consistency of genotypes among parents and nodes. Care must be taken to ensure that the node's genotype-probability contain the correct and updated genotype values of its parents'.

3.4 Testing

The overall testing strategy was as follow: we tested on cases where the pedigree tree contained only one node and more than one node, and where the genotype contained one locus, more than one locus, or no locus at all (as an hypothetical case). The test cases below satisfy a path-complete test, as defined by Guttag and Liskov[14], for the probability functions. Although a path-complete test does not guarantee that the procedures are error-free, it gives us good confidence. The actual testing codes can be found in "MEDG Users:Jervis: UROP:FINAL:calculateprobabilitiestest.lisp." In the test cases that involve pedigree trees with only 1 node, each gene can have three possible alleles.

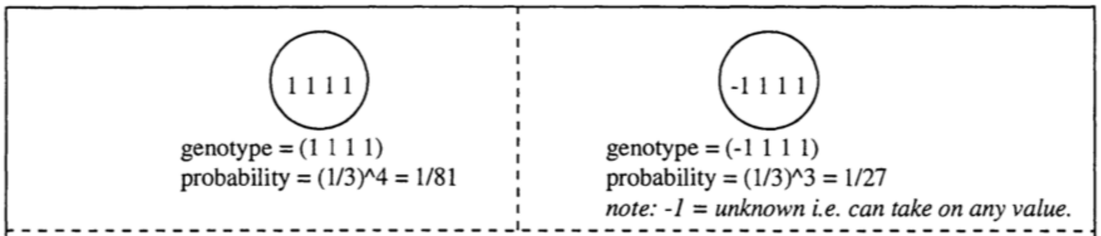
1 1 1 1
genotype = (1 1 1 1)
probability = (1/3)^4 = 1/81
-1 1 1 1
genotype = (-1 1 1 1)
probability = (1/3)^3 = 1/27
note: -1 = unknown i.e. can take on any value.

Figure 3.6: pedigree trees with only 1 node

[14] J. Guttag, and B. Liskov, *Abstraction and Specification in Program Development*, (The Massachusetts Institute of Technology, MA, 1995)

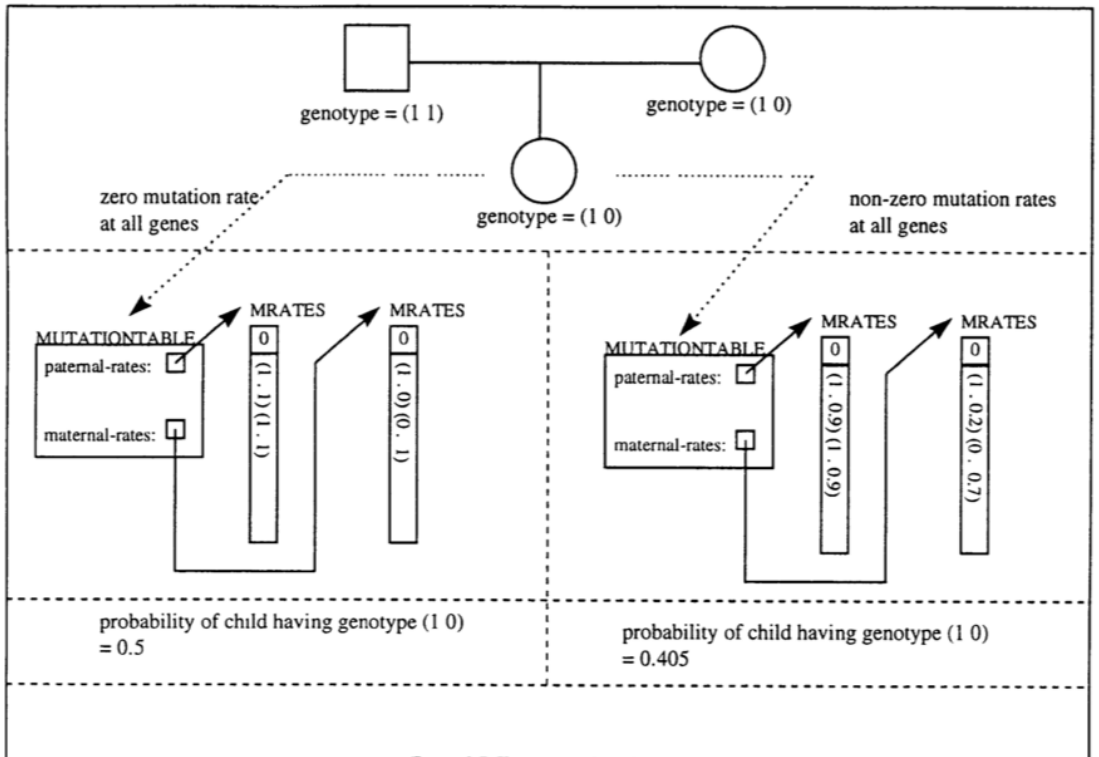

Figure 3.7: Test cases with 1 locus

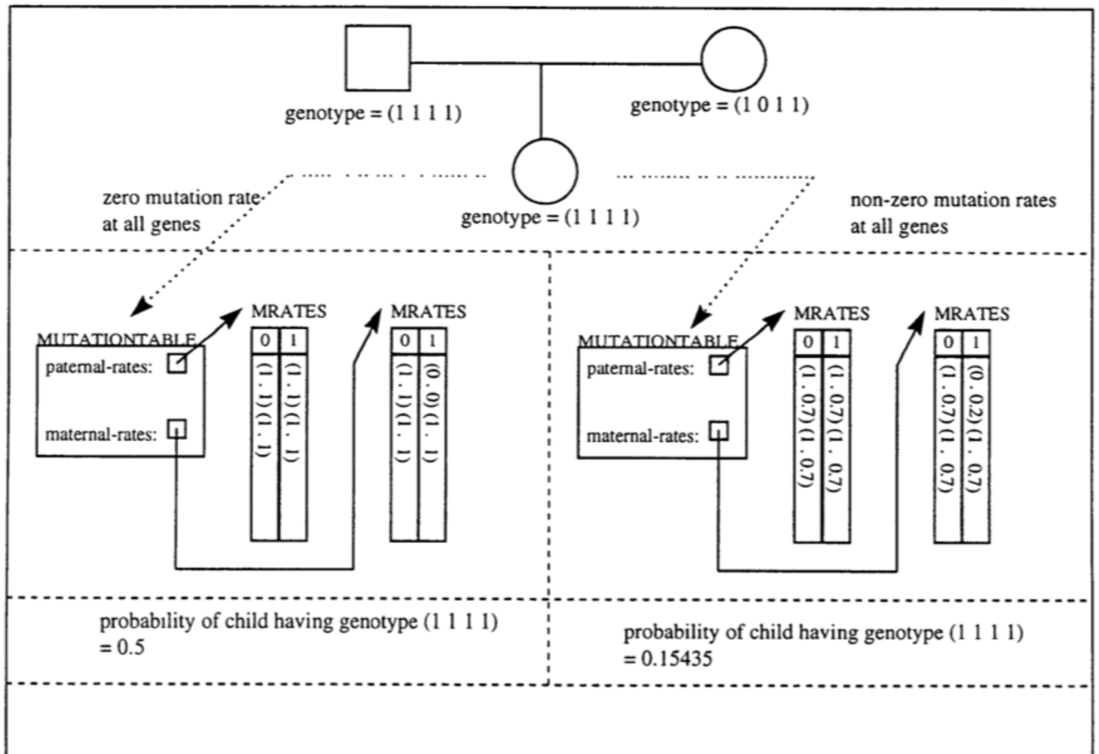

Figure 3.8: Test cases with 2 loci

Chapter 4

Modifying Procedures from the Previous Versions of Geninfer that Assumes One-Dimensional Values

4.1 Objective

Throughout the previous version of Geninfer, there were procedures that assumed 1 dimensional values and would work only if the nodes' values are 1 dimensional. A goal of this section is to identify some of these procedures and correct them appropriately so that they would work for multiple dimensional values as well. Another goal of this section is to define a type of object that will represent multiple dimensional value.

4.2 Solution and Implementation

The procedures in the bayes net system of Geninfer that assume 1-dimensional values include fast-val-known?, val-known?, SetKnownVal, ClearKnownVal, KnownVal, and setf val-known?, which are mostly defined in vars.lisp. They are no longer valid after the introduction of multiple dimensional values. To cope with this change, I modified these procedures into new ones with the following names and specifications:

```
defun vals-known? (value)
;; effects:        Returns true iff value does not contain unknown value in any of its
;;                 dimensions.

defun change-node-val (bayesnet node new-val)
;; modifies:       node, bayesnet
;; effects:        set (known-vals node) = new-val in bayesnet.

defun ClearKnownVal (node)
;; modifies:       node
;; effects:        set (known-vals node) = unknown value in all of its dimensions.

defun same-known-vals? (value1 value2)
;; effects:        Returns true iff value1=value2.

(defmethod num_universes ((node discrete-RV))
  ;; Returns the number of universes of node i.e. the appropriate dimension of
  ;; (known-vals node)

(defmethod known-vals-element-type ((node discrete-RV))
  ;; Returns the type of the elements of (known-vals node)

(defmethod unknown-vals ((node discrete-RV))
  ;; Returns a (known-vals node) type object that corresponds to unknown value.
```

Here is the scheme that has been used to replace invalid functions with the ones defined above:

1. Change fast-val-known? and val-known? to vals-known? if appropriate.
2. Change all calls to (setf (known-vals node) new-val) to (change-node-val node new-val).
3. Change all (= (known-vals node) ...) to (same-known-vals? (known-vals node) ...)
4. Change (setf (val-known? node) new-val) to (setf (vals-known? node) new-val)
5. Change (SetKnownVal node val) to (change-node-val node known-vals)

We have also defined a representation for multiple dimension values:

```
OBJECT: KNOWN-VAL
;; the object that represents the value of a node.
;; For example, when making a value, one should call (setf somevalue (make-known-vals
;;  ....))
;; rep is an array.

(defun make-known-vals (dimensions initial-contents element-type)
  ;; requires: initial-contents is passed as a list of initial contents, with
  ;;          the first content as the first element of initial-contents, etc.
  ;; effects:  Returns a known-vals object with dimension dimensions, initial
  ;;          contents initial-contents, and element type element-type.

(defun same-known-vals? (val1 val2)
  ;; is val1 same as val2?

(defun same-element? (e1 e2)
  ;; e1, e2 are some elements in a known-vals type.
  ;; Returns true if e1=e2; o.w. false.

(defun ith-element (val i)
  ;; Returns the ith element of val.

(defun num_dimensions (val)
  ;; returns the dimension of val (a known-vals type)

(defconstant unknownelement -1)
(defconstant kUnknownVal-1dim (make-known-vals 1 `(,unknownelement) 'integer))
```

4.3 Testing

The testing cases aimed to demonstrate that the program would work if the representation of (known-vals node) was a simple vector instead of an integer. Once this is established, no further testing is necessary because the code involved in incremental decomposition algorithm are not altered, and the code involved in probability calculations are independent of the code involved in this section. We built a bayes net similar to the one in figure 2.4, with nodes A through E, except that (known-vals node) was a simple vector instead of an integer. The purpose for choosing the same bayes net from the example in section 2.3 was so that a direct comparison of calculated results can be made. The actual code and test cases were described in details in Appendix A. The choice of test cases satisfy the requirements for a path-complete test, as defined by Guttag and Liskov[15]. In summary, the calculations were accurate for these test cases, reassuring us that the implementation was very reliable.

[15] J. Guttag, and B. Liskov, *Abstraction and Specification in Program Development*, (The Massachusetts Institute of Technology, MA, 1995)

Chapter 5

Conclusion

In summary, several goals have been accomplished: 1) incremental decomposition algorithm has been incorporated into the bayes net system of Geninfer, 2) a set of equations for probabilistic calculations involved in pedigree analysis were set up and ready to be incorporated into nodes with slot conditional-probability-equation, 3) several invalid functions that depend on 1-dimensional (known-vals node) have been replaced with valid ones that could assume 1-dimensional or multi-dimensional (known-vals node). Hopefully the successor of this project will be able to make use of these changes to build a version of Geninfer that can perform efficient calculations for pedigree analysis with multi-dimensional genotypes.

There are many more features that can be added to Geninfer. Currently, due to lack of information and objects available at this stage of the project, we could only write probability equations that take into account the genotypes of a nodes' parents, and neglect valuable information such as ethnicity and environmental factors. These are definitely elements that could be incorporated into the function genotype-probability at a later stage of the project. However, with these factors involved, the definition of genotype-probability may then vary among different members of the pedigree tree (e.g. members of different ethnic origins) and definitely vary from disease to disease. In addition, genotype-probability is valid only for autonomic disease; therefore, equations that calculate probability for sex-linked disease and cytoplasmic inheritance can definitely be added to Geninfer.

References

J.F. Crow, *Genetics Notes: An Introduction to Genetics* 8th ed, (Macmillan Publishing Co, NY, 1983).

G.F. Cooper, Bayesian Belief-Network Inference Using Recursive Decomposition, KSL-90-05, Stanford University, 1990.

W.M. Gelbart, A. Griffiths, R.C. Lewontin, J.H. Miller, and D.T. Suzuki, *An Introduction to Genetic Analysis* 5th edition, (W.H. Freeman & Company, NY, 1993)

J. Guttag, and B. Liskov, *Abstraction and Specification in Program Development*, (The Massachusetts Institute of Technology, MA, 1995)

N. Harris, *Probabilistic Reasoning in the Domain of Genetic Counseling*. S.M. thesis, Massachusetts Institute of Technology, May 1989.

S.P. Pauker, and P. Szolovits. Pedigree Analysis for Genetic Counseling. In Lun, K.C., et al. (eds.), *MEDINFO 92: Proceedings of the Seventh Conference on Medical Informatics,* pages 679-683. Elsevier (North Holland) 1992.

J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, (Morgan Kaufmann Publishers Inc, San Mateo, CA, 1988).

P. Szolovits, Uncertainty and Decisions in Medical Informatics, *Methods of Information in Medicine* **34** (1995) 111-21.

P.H. Winston, *Artificial Intelligence* 3rd ed, (Addison-Wesley Publishing Co, Reading, MA, 1992).

J.W. Wu, *Space-Time Trade Off via Loop-Unrolling applied to Algorithms in Genetic Linkage Analysis*. S.M. thesis, Massachusetts Institute of Technology, Feb. 1993.

APPENDIX

APPENDIX A: CODES FOR TEST CASES

```
;; SECTION 2.4
;; Description of Test Cases: set up a string bayes net as that
;; in Figure 2.1, and then demonstrate that the program works for
;; the example described in section 2.2 and figure 2.1.


;; Defining Variables
(defparameter stringnet (make-instance 'Cooper-BNET :name 'stringnet))

(defparameter x1
 (make-instance 'Cooper-RV
  :name 'x1
  :val-universes (list *true-false-universe*)
  :conditional-probability-table
  (make-array 2 :initial-contents '(.6 .4)
        :element-type 'float)))

(defparameter x2
 (make-instance 'Cooper-RV
  :name 'x2
  :val-universes (list *true-false-universe*)
  :conditional-probability-table
  (make-array '(2 2)
        :initial-contents '((.7 .3) (.1 .9))
        :element-type 'float)))

(defparameter x3
 (make-instance 'Cooper-RV
  :name 'x3
  :val-universes (list *true-false-universe*)
  :conditional-probability-table
  (make-array '(2 2)
        :initial-contents '((.7 .3) (.1 .9))
        :element-type 'float)))

(defparameter x4
 (make-instance 'Cooper-RV
  :name 'x4
  :val-universes (list *true-false-universe*)
  :conditional-probability-table
  (make-array '(2 2)
        :initial-contents '((.7 .3) (.1 .9))
        :element-type 'float)))

(defparameter x5
 (make-instance 'Cooper-RV
  :name 'x5
  :val-universes (list *true-false-universe*)
  :conditional-probability-table
  (make-array '(2 2)
        :initial-contents '((.7 .3) (.1 .9))
```

```
               :element-type 'float)))

(defparameter x6
  (make-instance 'Cooper-RV
    :name 'x6
    :val-universes (list *true-false-universe*)
    :conditional-probability-table
    (make-array '(2 2)
               :initial-contents '((.7 .3) (.1 .9))
               :element-type 'float)))

(defparameter x7
  (make-instance 'Cooper-RV
    :name 'x7
    :val-universes (list *true-false-universe*)
    :conditional-probability-table
    (make-array '(2 2)
               :initial-contents '((.7 .3) (.1 .9))
               :element-type 'float)))

;; Setting up the Bayes Net
(add-element x1 stringnet)
(add-element x2 stringnet)
(add-element x3 stringnet)
(add-element x4 stringnet)
(add-element x5 stringnet)
(add-element x6 stringnet)
(add-element x7 stringnet)
(add-arc stringnet x1 x2)
(add-arc stringnet x2 x3)
(add-arc stringnet x3 x4)
(add-arc stringnet x4 x5)
(add-arc stringnet x5 x6)
(add-arc stringnet x6 x7)

;; Testing correctness of calculation
(infer stringnet `((,x1 0) (,x2 0) (,x3 0)
                  (,x4 0) (,x5 0) (,x6 0) (,x7 0)))  ;; 0.0705894

;; Defining Records
(setf r1 (bnet-dissection-tree stringnet))
(setf r2 (net-y-ptr r1))
(setf r3 (net-y-ptr r2))
(setf r4 (net-z-ptr r2))
(setf r5 (net-z-ptr r1))
(setf r6 (net-y-ptr r5))
(setf r7 (net-z-ptr r5))


;; adding an arc between x1 and x3 shows that indeed only r2 is marked.
(add-arc stringnet x1 x3)
(setf (conditional-probability-table x3)
     (make-array '(2 2 2)
               :initial-contents '(((.5 .6) (.7 .8)) ((.9 .11) (.12 .13)))
```

```
             :element-type 'float))
(added-arcs stringnet)   ;; shows that the arc (x1.x3) is added to the slot added-arcs.
(mark-records stringnet) ;; mark(s) record(s).

;; Defining Records
(setf r1 (bnet-dissection-tree stringnet))
(setf r2 (net-y-ptr r1))
(setf r3 (net-y-ptr r2))
(setf r4 (net-z-ptr r2))
(setf r5 (net-z-ptr r1))
(setf r6 (net-y-ptr r5))
(setf r7 (net-z-ptr r5))
(marked r1)  ;; nil
(marked r2)  ;; t
(marked r3)  ;; nil
(marked r4)  ;; nil
(marked r5)  ;; nil
(marked r6)  ;; nil
(marked r7)  ;; nil


;; Incremental Decomposition: redissecting only the parts of the dissection tree that
;; is represented by record r2,r3,r4
(incremental-decompositions stringnet)
(initialize-caches stringnet)


;; Defining Records.
(setf r1 (bnet-dissection-tree stringnet))
(setf r2 (net-y-ptr r1))
(setf r3 (net-y-ptr r2))
(setf r4 (net-z-ptr r2))
(setf r5 (net-z-ptr r1))
(setf r6 (net-y-ptr r5))
(setf r7 (net-z-ptr r5))
(setf r8 (net-y-ptr r3))
(setf r9 (net-z-ptr r3))

;; Verify that the new dissection results in correction calculations.
(infer stringnet `((,x1 0) (,x2 0) (,x3 0)))  ;; 0.21




;; SECTION 2.4
;; Description of Test Cases: verify that it works when a node
;; is being deleted from a bayes net, as shown in the diagram
;; below:
;        A                  A
;       / \                / \
;       B  C      -->      B  C
;       V \                  V
;        D  E                D
; Defining Variables
(defparameter N-A
 (make-instance 'Cooper-RV
   :name 'n-a
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array 2 :initial-contents '(.6 .4)
             :element-type 'float)))
(defparameter N-B
 (make-instance 'Cooper-RV
   :name 'n-b
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2)
             :initial-contents '((.7 .3) (.1 .9))
             :element-type 'float)))
(defparameter N-C
 (make-instance 'Cooper-RV
   :name 'n-c
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2) :initial-contents '((.6 .4) (.2 .8))
             :element-type 'float)))
(defparameter N-D
 (make-instance 'Cooper-RV
   :name 'n-d
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2 2)
             :initial-contents '(((.6 .4) (.5 .5)) ((.8 .2) (.4 .6)))
             :element-type 'float)))
(defparameter N-E
 (make-instance 'Cooper-RV
   :name 'n-e
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2)
             :initial-contents '((.6 .4) (.1 .9))
             :element-type 'float)))
(defparameter MCBN1 (make-instance 'Cooper-BNET :name 'MCBN1))


; Setting up the bayes net with nodes N-A to N-E
;  A
; / \
; B  C
;  \/ \
;   D  E
(add-element N-A MCBN1)
(add-element N-B MCBN1)
(add-element N-C MCBN1)
(add-element N-D MCBN1)
(add-element N-E MCBN1)
```

```
(add-arc MCBN1 N-A N-B)
(add-arc MCBN1 N-A N-C)
(add-arc MCBN1 N-B N-D)
(add-arc MCBN1 N-C N-D)
(add-arc MCBN1 N-C N-E)

; Perform some calculations, so that a decomposition tree is being established.
(infer mcbn1 `((,n-b 1))) ; 0.54
(infer mcbn1 `((,n-a 0))) ; 0.6
(infer mcbn1 `((,n-a 1) (,n-b 1) (,n-c 0) (,n-d 0) (,n-e 0)))      ; 0.03456
(infer mcbn1 `((,n-a 1) (,n-b 1) (,n-c 0) (,n-d 0) (,n-e 1)))      ; 0.02304
(infer mcbn1 `((,n-a -1) (,n-b -1) (,n-c -1) (,n-d -1) (,n-e -1)))  ; 1.0


; Setting Up the Following Bayes Net
;  A
; / \
; B  C
;  \/
;   D
(delete-arc MCBN1 N-C N-E)
(delete-element n-e mcbn1)

;; verify calculations
(infer mcbn1 `((,n-b 1)))          ; 0.54
(infer mcbn1 `((,n-a 0)))          ; 0.6
(infer mcbn1 `((,n-a 1)))          ; 0.4
(infer mcbn1 `((,n-a 0) (,n-b 0)))  ; 0.42
(infer mcbn1 `((,n-a 0) (,n-b 1)))  ; 0.18
(infer mcbn1 `((,n-a 1) (,n-b 0)))  ; 0.04
(infer mcbn1 `((,n-a 1) (,n-b 1)))  ; 0.36
(infer mcbn1 `((,n-a 1) (,n-b 1) (,n-c 1) (,n-d 0)))     ; 0.1152
(infer mcbn1 `((,n-a 1) (,n-b 0) (,n-c 0) (,n-d 1)))     ; 0.0032
(infer mcbn1 `((,n-a 1) (,n-b 1) (,n-c 1)))             ; 0.288
(infer mcbn1 `((,n-a 1) (,n-b 1) (,n-d 0)))             ; 0.1728
(infer mcbn1 `((,n-a -1) (,n-b -1) (,n-c -1) (,n-d -1)))  ; 1.0
```

;; SECTION 4.3
;; Description of Test Cases: demonstrate that the program would
;; work if the representation of (known-vals node) is a simple
;; vector instead of an integer. This example is similar to the
;; one used in Section 3.4, except that (known-vals node) is now
;; a simple vector.

;; Defining Variables

```
(defparameter N-A
  (make-instance 'Cooper-RV
   :name 'n-a
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array 2 :initial-contents '(.6 .4)
             :element-type 'float)))
; (dissect mcbn1)

(defparameter N-B
  (make-instance 'Cooper-RV
   :name 'n-b
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2)
             :initial-contents '((.7 .3) (.1 .9))
             :element-type 'float)))

(defparameter N-C
  (make-instance 'Cooper-RV
   :name 'n-c
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2) :initial-contents '((.6 .4) (.2 .8))
             :element-type 'float)))
(defparameter N-D
  (make-instance 'Cooper-RV
   :name 'n-d
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2 2)
             :initial-contents '(((.6 .4) (.5 .5)) ((.8 .2) (.4 .6)))
             :element-type 'float)))
(defparameter N-E
  (make-instance 'Cooper-RV
   :name 'n-e
   :val-universes (list *true-false-universe*)
   :conditional-probability-table
   (make-array '(2 2)
             :initial-contents '((.6 .4) (.1 .9))
             :element-type 'float)))

(defparameter MCBN1 (make-instance 'Cooper-BNET :name 'MCBN1))

(add-element N-A MCBN1)
(add-element N-B MCBN1)
(add-element N-C MCBN1)
(add-element N-D MCBN1)
(add-element N-E MCBN1)
(add-arc MCBN1 N-A N-B)
(add-arc MCBN1 N-A N-C)
(add-arc MCBN1 N-B N-D)
(add-arc MCBN1 N-C N-D)
```

```
(add-arc MCBN1 N-C N-E)
;;  A
;; / \
;; B  C
;;  \/ \
;;   D  E


;; this function is for testing purpose only; it returns "ok." if actual = expect,
;; with a difference tolerance of 0.0000001 due to roundings in calculations.
(defun verify (actual expect identifier)
  (let ((tolerance 0.0000001))
    (format t "~%")
    (if (< (abs (- actual expect)) tolerance)
      (format t "ok.")
      (format t "not ok."))
    identifier))

; Defining more variables
(setf one  (make-known-vals 1 '(1) 'integer))
(setf zero (make-known-vals 1 '(0) 'integer))
(setf unknown (make-known-vals 1 '(-1) 'integer))

;; Test Cases.  If correct, then the function verify will return "ok."; otherwise
;; it will return "not ok."
;; #1
(verify (infer mcbn1 `((,n-b ,one))) 0.54 '1)
;; #2
(verify (infer mcbn1 `((,n-a ,zero))) 0.6 '2)
;; #3
(verify (infer mcbn1 `((,n-a ,one))) 0.4 '3)
;; #4
(verify (infer mcbn1 `((,n-e ,one))) 0.68 '4)
;; #5
(verify (infer mcbn1 `((,n-e ,zero))) 0.32 '5)
;; #6
(verify (infer mcbn1 `((,n-a ,zero) (,n-e ,one))) 0.36 '6)
;; #7
(verify (infer mcbn1 `((,n-a ,zero) (,n-b ,one) (,n-c ,one) (,n-d ,one) (,n-e ,one)))
      0.03888 '7)
;; #8
(verify (infer mcbn1 `((,n-a ,zero) (,n-b ,zero))) 0.42 '8)
;; #9
(verify (infer mcbn1 `((,n-a ,zero) (,n-b ,one))) 0.18 '9)
;; #10
(verify (infer mcbn1 `((,n-a ,one) (,n-b ,zero))) 0.04 '10)
;; #11
(verify (infer mcbn1 `((,n-a ,one) (,n-b ,one))) 0.36 '11)
;; #12
(verify (infer mcbn1 `((,n-a ,zero) (,n-b ,one) (,n-c ,one) (,n-d ,one) (,n-e ,zero)))
      0.00432 '12)
;; #13
(verify (infer mcbn1 `((,n-a ,one) (,n-b ,zero) (,n-c ,one) (,n-d ,zero) (,n-e ,one)))
      0.0144 '13)
```

```
;; #14
(verify (infer mcbn1 `((,n-a ,zero) (,n-b ,zero) (,n-c ,one) (,n-d ,one) (,n-e ,one)))
        0.0756 '14)

;; #15
(verify (infer mcbn1 `((,n-a ,one) (,n-b ,one) (,n-c ,zero) (,n-d ,one) (,n-e ,zero)))
        0.03456 '15)

;; #16
(verify (infer mcbn1 `((,n-a ,one) (,n-b ,one) (,n-c ,zero) (,n-d ,zero) (,n-e ,one)))
        0.02304 '16)

;; #17
(verify (infer mcbn1 `((,n-a ,unknown) (,n-b ,unknown) (,n-c ,unknown) (,n-d ,unknown)
        (,n-e ,unknown))) 1 '17)

;; #18
(verify (infer mcbn1 `((,n-a ,unknown))) 1 '18)
;;#19
(verify (infer mcbn1 `((,n-a ,unknown) (,n-b ,unknown))) 1 '19)
```

APPENDIX B: Program Listing

All files are stored in LCS machine "hippocrates," under the directory "MEDG Users:Jervis:UROP:FINAL2." The machine is located in Room NE43-417 of MIT Clinical Decision Making Group.

File: calculateprobabilities.lisp

```
(defun root-genotype-probability (node)
 (do* ((valuniverses (val-universes node) (cdr valuniverses))
      (valuniverse  (car valuniverses) (car valuniverses))
      (i 0 (+ i 1))
      (numelements  (if valuniverse
                  (cardinality valuniverse)
                  nil))
      (result 1))
     ((null valuniverse) result)
   (if (not (same-element? (ith-element (known-vals node) i) unknownelement))
    (setf result (* result (/ 1 numelements))))))


(defun genotype-probability (fgenotype mgenotype cgenotype map mutationtable)
 ;; fgenotype,mgenotype,cgenotype = father's,mother's,child's genotypes.
 ;; fgenotype=mgenotype=nil.
 ;; map =  stores distances between genes.
 ;; mutationtable = stores mutation rates that alleles in cgenotype mutates to alleles
 ;;             in fgenotype and mgenotype, as well as the probability of no
 ;;             mutation at all.

 ;; effect: Returns probability of child's genotype=cgenotype given father's
 ;; and mother's genotype = fgenotype,mgenotype respectively.  Halt with error
 ;; message if not all genotypes have the same number of genes.



 (cassert (= (num_genes fgenotype) (num_genes mgenotype)
         (num_genes cgenotype))
       (fgenotype mgenotype cgenotype)
       "Not all genotypes have the same number of genes.")

 ;; resultp, resultm = the probabilities of child having the corresponding
 ;; paternal and maternal genotypes.
 (if (= (num_genes fgenotype) 0)
   1
  (let ((resultp (genotype-probability-aux fgenotype cgenotype 'paternal map
                          (paternal-rates mutationtable)))
      (resultm (genotype-probability-aux mgenotype cgenotype 'maternal map
                                  (maternal-rates mutationtable))))

    ;; return value.
    (* resultp resultm))))


(defun genotype-probability-aux (pgenotype cgenotype child_pORm map mutationrates)
  ;; requires: pgenotype and cgenotype must describe the same number of genes,
  ;;          and must describe at least 1 gene.

  (+ (* 0.5
      (mutationrate mutationrates 0 (ith-gene pgenotype 0 'maternal))
      (genotype-probability-auxaux pgenotype 'maternal
                                   cgenotype child_pORm 1 1 map
                                   mutationrates))
    (* 0.5
      (mutationrate mutationrates 0 (ith-gene pgenotype 0 'paternal))
      (genotype-probability-auxaux pgenotype 'paternal
                                   cgenotype child_pORm 1 1 map
                                   mutationrates))))



(defun genotype-probability-auxaux (pgenotype pORm cgenotype child_pORm result
                                   i map mutationrates)
  ;; pgenotype = corresponding parent's genotype
  ;; cgenotype = child's genotype
  ;; result = returning value.
  ;; pORm, child_pORm = paternal/maternal
  ;; i = index of the genee being checked.
  ;; map = stores distances between genes.

  ;; requires: pgenotype describes the same genes as cgenotype
  ;; modifies:
  ;; effects:  Returns the probability that the genotype "pgenotype" will
  ;;          recombine s.t. the pORm (paternal/maternal) genotype of pgenotype
  ;;          will become identical to the child's child_pORm
  ;;          (paternal/maternal) genotype (cgenotype).
  ;;          e.g. if pORm='paternal and child_pORm='maternal, then it returns
  ;;          the probability that pgenotype will recombine s.t. the paternal
  ;;          genotype of pgenotype will become identical to the maternal
  ;;          genotype of cgenotype.


  (if (or (= i (num_genes cgenotype)) (= result 0))
    result
```

```
   (let ((pORmgene (ith-gene pgenotype i pORm))
        (comppORgene (ith-gene pgenotype i (comp pORm))))

     (+ (genotype-probability-auxaux pgenotype pORm
                                cgenotype child_pORm
                                (* result
                                  (mutationrate mutationrates i pORmgene)
                                  (- 1 (mapdistance map (- i 1) i)))
                                (+ i 1) map mutationrates)
       (genotype-probability-auxaux pgenotype (comp pORm)
                                cgenotype child_pORm
                                (* result
                                  (mutationrate mutationrates i comppORgene)
                                  (mapdistance map (- i 1) i))
                                (+ i 1) map mutationrates)))))



(defun comp (f)
 (cond ((equal f 'maternal) 'paternal)
      ((equal f 'paternal) 'maternal)))

;; mutationtable module begins

(defclass mutationtable ()
;; (paternal-rates mutationtable) and (maternal-rates mutationtable) are mrates
;; objects.

 ((paternal-rates :accessor paternal-rates :initarg :paternal-rates)
  (maternal-rates :accessor maternal-rates :initarg :maternal-rates)))

;; mutationtable module ends



;; mrates module beings
;; rep invariant: sum of all mutation rates for a certain gene cannot be greater than
;;              one.
(defclass mrates ()
 ((mrates-aux :accessor mrates-aux :initform (list) :initarg :mrates-aux)))

(defmethod add-mrates-aux ((mutationrates mrates) genenumber rates)
;; effects: add the mutation rates from child's allele to parents' allele to mrates-var
;; mrates-var is a mrates object
;; genenumber = the index that is used to identify the gene in question
;;            the "i" in ith-gene
;; rates is created by make-rates.
  (setf (mrates-aux mutationrates) (acons genenumber rates
                                     (mrates-aux mutationrates))))


(defun make-rates (ppair mpair)
;; requires: sum of all the rates less than or equal to 1.
;; effects: returns an alist of three pairs.
;;      The first  pair is (allele1 . mutation rate from allele1 to child's allele)
;;      The second pair is (allele2 . mutation rate from allele2 to child's allele)
;;      where allele1, allele2 = the paternal, maternal alleles of the parent in
;;      question.

  (let* ((result  (list mpair))
         (result2 (cons ppair result)))
    result2))



(defmethod mutationrate ((mutationrates mrates) genenumber allele)
  (let* ((rates-aux (mrates-aux mutationrates))
         (rates     (cdr (assoc genenumber rates-aux)))
         (rate      (cdr (assoc allele rates))))
    rate))


;; mrates module ends



;; geneticmaptype module begins
;; the rep of geneticmaptype is simple-vector.

(defclass geneticmaptype ()
  ((geneticmap :accessor geneticmap :initarg :geneticmap)))

(defun make-geneticmaptype (length initial-contents element-type)
  (make-instance 'geneticmaptype
    :geneticmap (make-array length :initial-contents initial-contents
                            :element-type     element-type)))

(defmethod totallength ((map geneticmaptype))
  (length (geneticmap map)))

(defmethod mapdistance ((map geneticmaptype) i j)
  ;; effects:  Returns the genetic map distance between gene i and j according
  ;;           to map.

  (let ((gmap (geneticmap map)))
    (cassert (and (>= j i) (< i (totallength map)) (<= j (totallength map)))
             (i j)
             "Invalid arguments i,j.")

    (do ((start i (1+ start)) (end j) (result (svref gmap i)
                                       (+ result
                                         (svref gmap (1+ start)))))
        ((= start (1- end))
         (if (> result 0.5)
           0.5
           result)))))


(defmethod set-mapdistance ((map geneticmaptype) i distance)
  ;; modifies: map
  ;; effects:  set map distance between locus i and i+1 on map to be distance.
  (setf (svref (geneticmap map) i) distance))

;; geneticmaptype module ends





;; genotype module begins (no macro version)
;; the rep is simple-vector.  The structure is as follows: if the genotype
;; describes n loci/genes, then the length of the structure, say g, will be 2n.
;; g[0]..g[n-1] represent the paternal genotype, while g[n]..g[2n] represent
;; the maternal genotype.

(defun make-genotype (dimensions initial-contents element-type)
  ;; requires: initial-contents is passed as a list of initial contents, with
  ;;           the first content as the first element of initial-contents, etc.
  ;; effects:  Returns a genotype object with dimension dimensions, initial
  ;;           contents initial-contents, and element type element-type.

  (make-known-vals dimensions initial-contents element-type))


(defun num_genes (g)
  ;; effects:  Returns the number of loci/genes that g describes.
```

```
 (cassert (evenp (length g))
        (g)
        "Invalid genotype argument: not an even number of elements.")
 (/ (length g) 2))


(defun ith-gene (g i origin)
 ;; effects:  If origin='paternal, then returns the paternal allele of the gene
 ;;          indexed by i in g i.e. i-th gene in g.  Similar for origin =
 ;;          'maternal.
 (cassert (< i (num_genes g))
        (i)
        "Argument i is out of range: must be less than number of genes.")

 (cond ((equal origin 'paternal)
       (svref g i))
      ((equal origin 'maternal)
       (svref g (+ i (num_genes g))))))


(defun half-genotype (g origin)
 ;; effects: if origin=paternal, then returns paternal genotype of g.
 ;;         if origin=maternal, then returns maternal genotype of g.
 (let* ((numgenes (num_genes g))
       (result (do ((i 0 (1+ i)) (lst nil))
                   ((= i numgenes) (nreverse lst))
                 (setf lst (cons (ith-gene g i origin) lst)))))
   (make-genotype numgenes result 'integer)))


(defun maternal (g)
 ;; effects: Returns the maternal genotype of g
 (half-genotype g 'maternal))

(defun paternal (g)
 ;; effects: Returns the paternal genotype of g
 (half-genotype g 'paternal))

;; genotype module ends (no macro version)




;; genetype module begins
;; the rep of gene is integer i.e. 0, 1, 2, etc, represent different alleles of
;; of the same gene.  alleleunknown is the allele that stands for unknown.
;; allelenull is the allele that stands for null i.e. a point deletion.

(defconstant alleleunknown kunknownval "Unknown allele/gene value")
(defconstant allelenull    -2          "Null allele/gene value")

(deftype genetype () '(integer))

(defun same-allele? (allele1 allele2)
  (= allele1 allele2))

;; genetype module ends
```

File: cooper.lisp

Note: only functions that are new or modified from the previous version are shown.

```
(defclass Cooper-BNET-mixin ()
  ((dissection-tree :accessor BNET-dissection-tree :initform nil)
   (compiled-evaluation-function :accessor BNET-compiled-evaluation-function
                          :initform nil)
   (added-elements :accessor added-elements :initform nil)
   (deleted-elements :accessor deleted-elements :initform nil)
   (added-arcs :accessor added-arcs :initform nil)
   (deleted-arcs :accessor deleted-arcs :initform nil))
  (:default-initargs :element-type 'Cooper-RV))


(defun reset-all-caches (BNET)
  (do-elements (n (set-representation BNET (elements BNET)) BNET)
    (reset-caches n)))


(defun testelements (lst domain range)
  ;; effects: Returns true iff domain is the first element of lst.  Otherwise,
  ;;         returns false.
  (declare (ignore range))
  (equal (car lst) domain))

(defun testarcs (lst domain range)
  ;; effects: Returns true iff (domain . range) is the first element of lst.
  ;;         Returns false otherwise.
  (and (equal (caar lst) domain) (equal (cdar lst) range)))


(defun searchNresult (CBNET get-lst test domain &optional (range nil))
  ;; requires: domain or (domain . range) appears at most once in
  ;;          (funcall get-lst CBNET).
  ;; modifies: lst = (funcall get-lst CBNET)
  ;; effects:  #1. If the dissection tree of CBNET is empty, then returns cons
  ;;              cell (nil . 'null-dissection-tree).
  ;;          #2. If (funcall get-lst CBNET) is empty, then returns cons cell
  ;;              (nil . 'not-found).
  ;;          #3. If range is not supplied and domain is an element of lst,
;;            then removes domain from lst and returns the cons cell
;;            (modified lst . 'removed).
;;          #4. If range is supplied and the cons cell (domain . range) is an
;;            element of lst, then removes (domain . range) from lst and
;;            returns cons cell (modified lst . 'removed).
;;          #5. In #3 and #4, if no matched element is found, then returns
;;            cons cell (lst . 'not-found).
;;
;;          Note, however, that lst is not modified if the wanted element is
;;          the first element of lst.  As a result, if the caller's goal is
;;          to obtain a modified lst, then the caller should check if cdr of
;;          returned result is 'removed; if so, then setf lst to car of the
;;          returned result.


(let ((lst (funcall get-lst CBNET)))

  (cond ((null (bnet-dissection-tree CBNET))
         (cons nil 'null-dissection-tree))
        ((null lst)
         (cons nil 'not-found))
        (t
         (let ((result 'not-found))
           (cond ((funcall test lst domain range)
                  (setf lst (cdr lst))
                  (cons lst 'removed))
                 (t
                  (do ((p1 lst (cdr p1)) (p2 (cdr lst) (cdr p2)))
                      ((or (equal result 'removed) (null p2))
                       (cons lst result))
                    (cond ((funcall test p2 domain range)
                           (setf result 'removed)
                           (setf (cdr p1) (cdr p2)))))))))))))



;; ??? Should I change do-elements to donodes?


(defun dissect (BNET &optional (compile? *compile-F*) (create-caches? t)
                     &aux (Update-Message-P (fboundp 'Update-Status-Message)))
  "Dissects BNET and returns value of F(BNET)"
  (declare (type function compile-eval))
  ;; First, reset old indexed records and recalculate Markov blanket and
  ;; neighborhood size; presumably, these changed to cause a new dissection
  ;; to occur.
```

```
 (do-elements (n (set-representation BNET (elements BNET)) BNET)
   (setf (BN-indexed-records n) nil)
   (BN-setup n BNET))
 (setf (BNET-dissection-tree BNET) nil)
 (setf (BNET-compiled-evaluation-function BNET) nil)
 (time-debug (setf (BNET-dissection-tree BNET) (CDT BNET)))
 ; jervis
 (time-debug (assign-indexed-records BNET))
 (time-debug (when create-caches?
             (create-caches BNET)))
 (time-debug (when compile?
             (if update-message-p (update-status-message "Compiling BNET"))
             (compile-eval BNET)))
 (initialize-caches BNET))


(defun assign-indexed-records (BNET)
 (let ((dtree (BNET-dissection-tree BNET)))
   (assign-indexed-records-aux dtree BNET)))

(defun assign-indexed-records-aux (dtree BNET)
 (cond ((null dtree) nil)
       (t (assign-indexed-records-aux (net-y-ptr dtree) BNET)
          (assign-indexed-records-aux (net-z-ptr dtree) BNET)
          (do-elements (n (variable-set dtree) BNET)
            (add-to-index n dtree)))))



;; this is an alpha version of mark-records
(defun mark-records-alpha (BNET)
 ;; modifies: BNET
 ;; effects:  Marks the records that represent the smallest subnetworks in
 ;;           the decomposition tree of BNET that needs to be redissect,
 ;;           according to Cooper's incremental decompositions algorithm
 ;;           on pg. 71 of Cooper's paper on "Bayesian Belief-Network
 ;;           Inference Using Recursive Decomposition".
 ;;           Clears the record of added and deleted arcs.

 (let ((dtree (BNET-dissection-tree BNET)))

   ;; for every pair of nodes of the added arcs of BNET
   (dolist (node-pair (added-arcs BNET))
     (let* ((node-set (set-representation BNET
                                (list (car node-pair)
                                      (cdr node-pair)))))
```

```
      (mark-records-arc dtree node-set BNET)))
    (setf (added-arcs BNET) nil)

    ;; for every pair of nodes of the added arcs of BNET
    (dolist (node-pair (deleted-arcs BNET))
     (let* ((node-set (set-representation BNET
                                  (list (car node-pair)
                                        (cdr node-pair)))))
      (mark-records-arc dtree node-set BNET)))
    (setf (deleted-arcs BNET) nil)))


(defun indtree? (BNET node)
  ;; effects: Returns true if the bnet dissection tree of BNET contains node
  ;;          in any of its records' variable set; otherwise returns false.

  (let* ((ei (element-index node BNET)))
   (b-in-set? ei (variable-set (bnet-dissection-tree BNET)))))


(defmethod set-representation-aux ((u universe) elements &aux (bits (b-empty-set)))
  (dolist (e elements bits)
   (setq bits
       (b-union bits
             (b-singleton-set (rass u e))))))


;; this is a beta version of mark-records
(defun mark-records (BNET &optional (make-cache-array-now? t))
  ;; requires: (delete-arc x y BNET) or (delete-arc z x BNET) where y = all
  ;;           childrens of x and z = all parents of x must be called before
  ;;           (delete-element x BNET) is called.  This can easily be
  ;;           accomplished by having the caller of delete-element checks all
  ;;           connections of x before calling delete-element, and this
  ;;           requirements saves a lot of computation time and allows simpler
  ;;           algorithms to be used in this function.
  ;; modifies: BNET
  ;; effects:  Marks the records.
  ;;           Clears the record of added and deleted arcs and nodes.

  (let ((dtree (BNET-dissection-tree BNET)))

    ;; for every pair of nodes of the deleted arcs of BNET
    (dolist (node-pair (deleted-arcs BNET))
     (let* ((node-set (set-representation-aux BNET
                                  (list (car node-pair)
                                    (cdr node-pair)))))
   (mark-records-arc dtree node-set BNET)))
(setf (deleted-arcs BNET) nil)

;; for every pair of nodes of the added arcs of BNET
(setf (added-arcs BNET)
     (do ((arcs (added-arcs BNET)) (progress t) (arcsNprogress nil))
        ((or (null arcs) (null progress)) arcs)

       (setf arcsNprogress
            (do ((temparcs nil) (work nil))
               ((null arcs) (cons temparcs work))
              (let* ((nodepair (pop arcs))
                  (a (car nodepair))
                  (d (cdr nodepair))
                  (ain (indtree? BNET a))
                  (din (indtree? BNET d)))
                (cond ((and ain din)        ;; when both are old elements.
                     (mark-records-arc dtree
                                  (set-representation BNET
                                                 (list a d))
                                  BNET)
                     (setf work t))
                    ((and ain (null din)) ;; when only d is new element.
                     (mark-records-arc dtree
                                  (set-representation BNET
                                                 (list a))
                                  BNET
                                  'add-element
                                  (element-index d BNET))
                     (setf work t))
                    ((and (null ain) din) ;; when only a is new element.
                     (mark-records-arc dtree
                                  (set-representation BNET
                                                 (list d))
                                  BNET
                                  'add-element
                                  (element-index a BNET))
                     (setf work t))
                    ((and (null ain) (null din))
                     ;; when both are new elements.
                     (push nodepair temparcs))))))
       (setf arcs (car arcsNprogress))
       (setf progress (cdr arcsNprogress))))
```

```
;; if there are arcs between nodes that are both independent of the rest of
;; the network.
(cond ((not (null (added-arcs BNET)))
      (let* ((newvs
              (do ((arcs (added-arcs BNET)) (vs (b-empty-set)))
                 ((null arcs) vs)
               (let* ((aNd (pop arcs))
                      (a (car aNd))
                      (d (cdr aNd))
                      (aindex (element-index BNET a))
                      (dindex (element-index BNET d)))
                 (if (not (b-in-set? aindex vs))
                   (setf vs (b-union (b-singleton-set aindex) vs)))
                 (if (not (b-in-set? dindex vs))
                   (setf vs (b-union (b-singleton-set dindex) vs))))))
             (new-rec
              (ConstructRecord (b-empty-set) newvs BNET)))
        (appendroot BNET new-rec))))
(setf (added-arcs BNET) nil)


;; for every node in the deleted elements of BNET
(dolist (e (deleted-elements BNET))
  (let* ((es (set-representation-aux BNET (list e))))
    (mark-records-arc dtree es BNET 'delete-element
                  (element-index-force e BNET))))
(setf (deleted-elements BNET) nil)


;; for every node in the added elements of BNET
(dolist (e (added-elements BNET))
  (unless (indtree? BNET e)
    (let* ((ei (element-index e BNET))
           (eis (b-singleton-set ei))
           ;; ??? must re-check here.
           (new-rec (make-rec
                     :summation-set eis
                     :evaluation-set eis
                     :instantiation-set (b-empty-set)
                     :variable-set eis
                     :instantiation-cache (create-inst-cache (b-empty-set)
                                              BNET)
                     :net-y-ptr nil
                     :net-z-ptr nil
                     :Y-Q-copy (b-empty-set)
                     :Z-Q-copy (b-empty-set)
```

```
                    :H-U-BestS-copy eis)))
        (if make-cache-array-now?
          (create-inst-cache-now new-rec BNET))
        (appendroot BNET new-rec))))
   (setf (added-elements BNET) nil)))



(defun appendroot (BNET new-z &optional (make-cache-array-now? t))
  ;; modifies: dtree=(BNET-dissection-tree BNET)
  ;; effects:  Attach the subnetwork described by new-z to dtree, in a way such
  ;;           that new-z is totally independent of dtree probabilistically
  ;;           i.e. no arc attaches any node between new-z and dtree.

  ;; ??? must re-check the logic here.
  (let* ((dtree (bnet-dissection-tree BNET))
         (vs (variable-set dtree))
         (new-root (make-rec
                    :summation-set (b-empty-set)
                    :evaluation-set (b-empty-set)
                    :instantiation-set (b-empty-set)
                    :variable-set (b-union (variable-set new-z) vs)
                    :instantiation-cache (create-inst-cache (b-empty-set) BNET)
                    :net-y-ptr dtree
                    :net-z-ptr new-z
                    :Y-Q-copy vs
                    :Z-Q-copy (variable-set new-z)
                    :H-U-BestS-copy (b-empty-set))))
    (if make-cache-array-now?
      (create-inst-cache-now new-root BNET))
    (setf (bnet-dissection-tree BNET) new-root)))



(defun create-inst-cache-now (rec BNET)
  ;; modifies: ic = (instantiation-cache rec)
  ;;           icv= (instantiation-cache-vec rec)
  ;; effects:  same as create-inst-cache, except that ic is immediately made
  ;;           to be the intended representation (in this case an array), and
  ;;           icv is immediately made to be a displaced ic.

  ;; the instantiation cache must have room for values indexed by any legit
  ;; combination of values of set-of-vars.
```

```
 (let* ((dims (create-inst-cache (instantiation-set rec) BNET))
        (a (make-array dims :initial-element nil)))
   (setf (instantiation-cache rec) a)
   (setf (instantiation-cache-vec rec)
         (make-array (list (apply #'* dims)) :displaced-to a))))


(defun incremental-decompositions (BNET)
 ;; modifies: BNET
 ;; effects:  Sets up the markov blanket and neighborhood size of every nodes.
 ;;           Dissects the subnetworks corresponding to the records in
 ;;           the decomposition tree of BNET that are marked, according
 ;;           to the incremental decompositions method described on
 ;;           pg.71 of Gregor Cooper's paper on "Bayesian Belief-Network
 ;;           Inference Using Recursive Decomposition".
 ;;           Assign records to nodes in their variable sets.

 (let ((dtree (BNET-dissection-tree BNET)))
   (cond ((null dtree) nil)
         ((marked dtree) (dissect BNET))
         (t
          (do-elements (n (set-representation BNET (elements BNET)) BNET)
            (setf (BN-indexed-records n) nil)
            (BN-setup n BNET))
          (setf (BNET-compiled-evaluation-function BNET) nil)
          (incremental-decompositions-aux dtree BNET)
          (assign-indexed-records BNET)))))


(defun F (BNET &optional (opt nil) (dtree nil))
 (if (null opt)
   (if (BNET-compiled-evaluation-function BNET)
     (funcall (BNET-compiled-evaluation-function BNET))
     (Cooper BNET))
   (Cooper BNET opt dtree)))
```

```
(defclass Cooper-RV-equ (Cooper-mixin equation-DRV)
 ())


(defclass Cooper-BNET-equation-mixin ()
 ((dissection-tree :accessor BNET-dissection-tree :initform nil)
  (compiled-evaluation-function :accessor BNET-compiled-evaluation-function
                        :initform nil)
  (added-elements :accessor added-elements :initform nil)
  (deleted-elements :accessor deleted-elements :initform nil)
  (added-arcs :accessor added-arcs :initform nil)
  (deleted-arcs :accessor deleted-arcs :initform nil))
 (:default-initargs :element-type 'Cooper-RV-equ))

(defclass Cooper-BNET-equation (Cooper-BNET-equation-mixin Discrete-Bayes-Net)
 ())





;; modified it so that it iterates all values even for multiple dimension values.
(defun iterate-over-vals (node proc)
 ;;; Proc is called once for each possible value of node.  Because node may
 ;;; have multi-dimensional values, proc is actually given a list of the
 ;;; values.  Note that the values are represented by the representation of
 ;;; the value in the universe (read integers), not the actual elements.
 ;;; We need to iterate over all valid indices for elements of the set.
 (declare
  (optimize (speed 3) (space 0) (safety 0))
  (inline car cdr cons >= 1+ ash)
  (type function proc))
 (labels ((iter (rest-univs revlist node ith)

          (if rest-univs
            (let ((u (pop (the list rest-univs)))
                 (ithelement (ith-element (known-vals node) ith)))
              (if (same-element?  ithelement unknownelement)
                (do ((i 0 (1+ i))
                     (usize (number-of-domain-elements u)))
                    ((>= i usize) nil)
              (declare (type  (fixnum 0 2048 ) i))
              (unless (b-empty-set? (valid-bits u (ash 1 i)))
                (iter  rest-univs (cons i (the list revlist)) node
                      (+ 1 ith))))
             (iter rest-univs (cons ithelement (the list revlist))
                  node (+ 1 ith))))

          (funcall proc (reverse (the list revlist)))))) ; Changed from nreverse
   (iter (val-universes node) nil node 0)
   nil))



;; modified the specifications such that in instantiations, nodes are binded to
;; a known-vals object rather than a single-digit value such as 1, 0.
(defun infer (BNET instantiations)
  "Instantiations is an alist of node . value pairs.  We compute the probability
   that this set of instantiations is present in the network."
  ;; In this version of infer, we assume that the instantiated value is of the
  ;; same type of (known-vals node) where node is any node being instantiated.

  ;; (declare (optimize (speed 3) (size 0) (safety 0)))

  ;; prepares (bnet-dissection-tree bnet)
  (cond ((null (BNET-dissection-tree BNET))
        (dissect BNET)
        (setf (added-arcs BNET) nil)
        (setf (deleted-arcs BNET) nil))
       ((or (added-arcs BNET) (deleted-arcs BNET)
           (added-elements BNET) (deleted-elements BNET))
        (selective-redecomposition BNET)))


  (if instantiations
    (do-elements (n (set-representation BNET (elements BNET)) BNET)
      ;; Loop over all n RVs

      ;; Is assoc a bottleneck here? See 'to do' at top.
      (let* ((new-inst-spec (assoc n instantiations))
            (new-val (cadr new-inst-spec))
            (old-flagged (vals-known? n))
            (old-val (known-vals n)))
        ;; The cases of interest are:
        ;; 1.  Value was pinned but is no longer: reset cache
        ;; 2.  Value was not pinned but now is: reset cache
      ;; 3.  Value is and was pinned, but to different values: reset cache
      ;; 4.  otherwise, leave cache intact.  This occurs only if there is&was
      ;;     no value or if the value happened to be the same.
;      (format t "~&~a Old-flag ~a Old-val ~a, new-val ~a" (name n)
;              old-flagged old-val new-val)


      (if new-inst-spec
        (if old-flagged
          (cond ((not (same-known-vals? old-val new-val)) ;;if there is new and
                 (change-node-val n new-val)             ;;and old value, and
                 (reset-caches n)))                      ;;new~=old (case 3).
          (cond (t
                 (change-node-val n new-val)      ;; if there is new but no old
                 (reset-caches n))))              ;; value (case 2).
        (if old-flagged
          (cond (t                             ;; if there is no new but has
                 (ClearKnownVal n)             ;; old value (case 1).
                 (setf (vals-known? n) nil)    ;; if there is no new and no
                 (reset-caches n)))))                 ;; and no old value (case 4).
      )))


  (F BNET))




;; modifies cooper s.t. it no longer assumes 1-dimensional values.
(defun Cooper (BNET &optional (opt nil) (dtree nil)
                 &aux (inst-set (b-empty-set)))
  (declare
   (optimize (speed 3) (space 0) (safety 0))
   (inline  cooper-evaluate aref known-vals-of-set))
  (labels
    (

    ;; Returns the product of all conditional probabilities
    ;; in the specified evaluation set.
    (Cooper-evaluate (evaluation-set &aux (prod 1.0d0))

      (do-elements (n evaluation-set BNET prod)
        (setq prod (* prod (conditional-probability n BNET)))))
```

```
(inner (rec)
 (if (null rec)
  1.0d0
  (let ((iset-vals
        (known-vals-of-set (instantiation-set rec) BNET))

       (cache (instantiation-cache rec))
       )
   (declare (list iset-vals))
   ;(format t "~& Iset-vals are ~a" iset-vals)
   (cond
    ((apply #'aref cache iset-vals))
    (t (setf
        (apply #'aref cache iset-vals)
        (let ((sum 0.0d0))
          (declare (type double-float sum))
          (labels
            ((sum-over (remaining-nodes)
              ;(format t "~& Sum-over remaining nodes ~a" remaining-nodes)
              (if (null remaining-nodes)
                (let* ((CE (Cooper-evaluate
                          (evaluation-set rec)))
                      (CY (if (zerop CE)
                            0.0d0
                            (inner (net-y-ptr rec))))
                      (CZ (if (zerop CY)
                            0.0d0
                            (inner (net-z-ptr rec)))))
                  (declare (type double-float CE CY CZ))
                  (incf sum (the double-float (* CE CY CZ)))
                  ;(format t "~& Sum is ~a" sum)
                  sum
                  )
                (let ((node (car remaining-nodes)))
                  (iterate-over-vals
                   node
                   #'(lambda (known-values)
                       ;(format t "~& Known-Values ~a" known-values)

                       ;;??? changed this line to the line below
                       ;;(SetKnownVal  node  (first known-values))
                       (change-node-val node
                                        (make-known-vals
                                         (length known-values)
                                         known-values
                                         'integer))
```

```
                    (sum-over (cdr remaining-nodes))))
                 ;; Why clear?
                 (ClearKnownVal node)))))

             (sum-over
              (set-elements BNET
                       (b-setdifference (summation-set rec)
                                inst-set))))

            (abs sum)))))))))

   (do-elements (n (set-representation BNET (elements BNET)) BNET)
    (when (vals-known? n)
     (setq inst-set (b-union inst-set
                    (b-singleton-set (element-index n BNET))))))

   (if (null opt)
    (inner (BNET-dissection-tree BNET))
    (progn (format t "entering inner dtree")
        (inner dtree)))))


(defun selective-redecomposition (BNET)
 ;; modifies: BNET
 ;; effects:  Selective redecomposition of only parts of the belief network
 ;;         BNET using incremental decompositions method as described on
 ;;         pg.71 of Gregor Cooper's paper on "Bayesian Belief-Network
 ;;         Inference Using Recursive Decomposition".


 (cond ((added-elements bnet)
     (dissect BNET)
     (setf (added-elements BNET) nil)
     (setf (deleted-elements BNET) nil)
     (setf (added-arcs BNET) nil)
     (setf (deleted-arcs BNET) nil))
    (t
     (mark-records BNET)
     (incremental-decompositions BNET)
     (initialize-caches BNET))))



(defun ConstructRecord (H X BNET &optional (make-cache-array-now? nil))
;; note: the old version assigns new-rec to BN-indexed-records of all nodes
;;      in the variable set.  This new version does not.  It relies on its
;;      callers to call the function (assign-indexed-records BNET) to do the
;;      assignment.  This change is made for the purpose of the implementation
;;      of Cooper's incremental decomposition algorithm.

(debug-dissection "~2%Constructing Record for X(~d)=~s, H(~d)=~s."
          (b-set-cardinality X) (describe-nodes X BNET)
          (b-set-cardinality H) (describe-nodes H BNET))
(and (not (b-empty-set? X))
   (multiple-value-bind (BestA BestB BestS)
                (Bisect X H BNET)
     (let* ((H-U-BestS (b-union H BestS))
          (Q (let ((set (b-empty-set)))
             (do-elements (n (b-intersection X H-U-BestS) BNET set)
                  (if (b-subsetp (BN-bv-parents n BNET) H-U-BestS)
                   (setq set (b-union (BN-bit-rep n BNET) set))))))
          (K-BestA (all-children-of-set BestA BNET))
          (K-BestA-A-X (b-intersection K-BestA X))
          (Y (b-union BestA K-BestA-A-X))
          (Z (b-setdifference (b-union BestB BestS) K-BestA-A-X))
          (Y-Q (b-setdifference Y Q))
          (Z-Q (b-setdifference Z Q))
          #|(foo (debug-dissection
               "~% H u BestS(~d)=~s, ~% Q(~d)=~s,~
               ~% K-BestA(~d)=~s, ~% K-BestA^X(~d)=~s,~
               ~% Y-Q(~d)=~s, ~% Z-Q(~d)=~s."
               (b-set-cardinality H-U-BestS) (describe-nodes H-U-BestS BNET)
               (b-set-cardinality Q) (describe-nodes Q BNET)
               (b-set-cardinality K-BestA) (describe-nodes K-BestA BNET)
               (b-set-cardinality K-BestA-A-X) (describe-nodes K-BestA-A-X BNET)
               (b-set-cardinality Y-Q) (describe-nodes Y-Q BNET)
               (b-set-cardinality Z-Q) (describe-nodes Z-Q BNET)))|#
          (Y-rec (ConstructRecord (b-intersection (theta Y-Q BNET) H-U-BestS)
                       Y-Q
                       BNET make-cache-array-now?))
          (Z-rec (ConstructRecord (b-intersection (theta Z-Q BNET) H-U-BestS)
                       Z-Q
                       BNET make-cache-array-now?))
          (new-rec (make-rec
                :summation-set (b-setdifference BestS H)
                :evaluation-set Q
                :instantiation-set H
                :variable-set (b-setdifference X H)
                :instantiation-cache (create-inst-cache H BNET)
                :net-y-ptr Y-rec
```

```
                    :net-z-ptr Z-rec
                    :marked nil
                    :Y-Q-copy Y-Q
                    :Z-Q-copy Z-Q
                    :H-U-BestS-copy H-U-BestS)))
        (if make-cache-array-now?
          (create-inst-cache-now new-rec BNET))
        new-rec))))



(defstruct (rec (:conc-name nil)
             (:print-function
              (lambda (p s k)
                (declare (ignore p k))
                (format s "#<REC>"))))
  summation-set
  evaluation-set
  instantiation-set
  variable-set
  instantiation-cache
  instantiation-cache-vec       ;; a vector displaced to cover the cache (hack)
  net-y-ptr
  net-z-ptr
  marked           ;; for incremental-decomposition
  Y-Q-copy          ;; same as above
  Z-Q-copy          ;; same as above
  H-U-BestS-copy     ;; same as above
  (modifications nil) ;; same as above
  )


(defun initialize-caches (BNET &aux (Update-Message-P
                                (fboundp 'Update-Status-Message)))
  ;; modifies: BNET
  ;; effects:  Evaluating the F function with the values of all variables
  ;;          temporarily set to unknown.  Upon exit, all variables resume
  ;;          their values prior to entering this function.

  ;; Evaluating the F function for the first time is different than for all others
  ;; because we want to evaluate for all possible cases so that when we get 0's
  ;; in the caches, we can freeze them permanently; they represent a structural
  ;; constraint in the network given the current network structure and the current
  ;; conditional probability tables, independent of the instantiated values of any
  ;; nodes.  Therefore, for this evaluation, we UNinstantiate any nodes that are
  ;; instantiated, temporarily.  After running F and pinning the zeros, we then
```

```
 ;; invalidate the caches corresponding to nodes we know are really (currently)
 ;; instantiated.

 (let ((save-pinned-values nil)
      (temp))

   (do-elements (n (set-representation BNET (elements BNET)) BNET)
    (when (vals-known? n)
      (push (cons n (known-vals n)) save-pinned-values)
      (ClearKnownVal n)
      (reset-caches n)
      (setf (vals-known? n) nil)
      )
    )
   ; (format t "~& Dissect: Save-pinned-values ~a" save-pinned-values)
   (if update-message-p (update-status-message "Computing probabilities"))
   (unwind-protect
    (prog1
      (time-debug (F BNET)) ; Returned value
      (time-debug (caches-pin-zeros (BNET-dissection-tree BNET))))
    ;; Reset values even if there was an error in #'F
    (dolist (spv save-pinned-values)
      (setq temp (car spv))
      (change-node-val temp (cdr spv))
      (setf (vals-known? temp) t)
      (reset-caches temp)))))



(defun incremental-decompositions-aux (dtree BNET)
 ;; modifies: dtree
 ;; effects:  Dissects the subnetworks corresponding to the records in
 ;;           dtree that are marked, according to the incremental
 ;;           decompositions method described on pg.71 of Gregor
 ;;           Cooper's paper on "Bayesian Belief-Network Inference Using
 ;;           Recursive Decomposition".


 ;; several cases needed to be taken care of:
 ;; 0: when network Y or network Z is nil.
 ;; 1: when root of network Y of dtree is or is not marked.
 ;; 2: when root of network Z of dtree is or is not marked.

 ;; case 1
 (cond ((and (net-y-ptr dtree) (marked (net-y-ptr dtree)))
       (let ((newH (b-intersection (theta (Y-Q-copy dtree) BNET)
                              (H-U-BestS-copy dtree)))
             (newX (Y-Q-copy dtree)))
           (dolist (actionNelementindex (modifications (net-y-ptr dtree)))
             (let ((a (car actionNelementindex))
                   (ei (cdr actionNelementindex)))
               (cond ((equal a 'delete)
                      (setf newX (b-setdifference newX (b-singleton-set ei)))))))
           (setf (net-y-ptr dtree) (ConstructRecord newH newX BNET
                                                    'make-cache-array-now))))
        ((net-y-ptr dtree)
         (setf (modifications dtree) nil)
         (incremental-decompositions-aux (net-y-ptr dtree) BNET)))

  ;; case 2
  (cond ((and (net-z-ptr dtree) (marked (net-z-ptr dtree)))
         (let ((newH (b-intersection (theta (Z-Q-copy dtree) BNET)
                                     (H-U-BestS-copy dtree)))
               (newX (Z-Q-copy dtree)))

           (dolist (actionNelementindex (modifications (net-z-ptr dtree)))
             (let ((a (car actionNelementindex))
                   (ei (cdr actionNelementindex)))
               (cond ((equal a 'delete)
                      (setf newX (b-setdifference newX (b-singleton-set ei)))))))
           (setf (net-z-ptr dtree) (ConstructRecord newH newX BNET
                                                    'make-cache-array-now))))
        ((net-z-ptr dtree)
         (setf (modifications dtree) nil)
         (incremental-decompositions-aux (net-z-ptr dtree) BNET))))




(defun mark-records-arc (dtree node-set BNET
                         &optional (mode nil) (ei nil))
  ;; modifies: dtree
  ;; effects:  marks the deepest record of dtree that contains the nodes
  ;;           in node-set in its variable set.  Flush the cache of all records
  ;;           in dtree that contain node-set in their variable sets.


  ;; several cases needed to be taken care of:
  ;; 1: when the tree is empty.
  ;; 2: when the record's variable set does not contain the set node-set.
  ;;
```

```
;; When the record's variable set contains the set node-set, and
;; 3: The record is already marked.
;; 4: Neither of dtree's left or the right subnetwork's root record's
;;     variable set contains the set node-set.
;; 5: If dtree's left subnetwork's root record's variable set contains
;;     the set node-set.
;; 6: If dtree's right subnetwork's root record's variable set contains
;;     the set node-set.


(cond ((not dtree) nil) ;;case1
      ((not (b-subsetp node-set (variable-set dtree))) nil) ;;case2
      (t
       (cond ((equal mode 'add-element)
              (setf (variable-set dtree)
                    (b-union (variable-set dtree)
                             (b-singleton-set ei)))
              (push (cons 'add ei) (modifications dtree)))
             ((equal mode 'delete-element)
              (setf (variable-set dtree)
                    (b-setdifference (variable-set dtree)
                             (b-singleton-set ei)))
              (push (cons 'delete ei) (modifications dtree))))
       (cond ((marked dtree) ;; case 3
              t)
             (t
              (create-inst-cache-now dtree BNET)
              (let ((y-result
                     (mark-records-arc (net-y-ptr dtree) node-set BNET mode
                                       ei))
                    (z-result
                     (mark-records-arc (net-z-ptr dtree) node-set BNET mode
                                       ei)))

                (if (and (not y-result) (not z-result)) ;;case4
                  (progn
                    (setf (marked dtree) t) t)
                  t))))) ;;case5 and case6
))
```

File: DAG.lisp
Note: only functions that are new or modified from the previous version are shown.

```
(defmethod add-arc ((d DAG) (source-index set-index) (target-index set-index))
 (cassert (and (and (< source-index (number-of-domain-elements d))
                    (not (b-in-set? source-index (deleted d))))
               (and (< target-index (number-of-domain-elements d))
                    (not (b-in-set? target-index (deleted d)))))
          (source-index target-index)
          "Source and target indices must identify nodes in the DAG.")
 (add-association (DAG-arcs-forward d) source-index target-index)
 (add-association (DAG-arcs-backward d) target-index source-index)
 t)

(defmethod delete-arc ((d DAG) (source-index set-index) (target-index set-index))
 (cassert (and (and (< source-index (number-of-domain-elements d))
                    (not (b-in-set? source-index (deleted d))))
               (and (< target-index (number-of-domain-elements d))
                    (not (b-in-set? target-index (deleted d)))))
          (source-index target-index)
          "Source and target indices must identify nodes in the DAG.")
 (delete-association (DAG-arcs-forward d) source-index target-index)
 (delete-association (DAG-arcs-backward d) target-index source-index)
 t)


(defmethod add-element :after (element (CBNET Cooper-BNET))
 ;; modifies: (added-elements CBNET), (deleted-elements CBNET)
 ;; effects:  If (null-dissection-tree CBNET), then don't do anything;
 ;;           otherwise,
 ;;           #1. If element is in (deleted-elements CBNET), then remove
 ;;               element from (deleted-elements CBNET).
 ;;           #2. If element is not in (deleted-elements CBNET), then push
 ;;               element into (added-element CBNET).

 (let* ((lstNresult (searchNresult CBNET
                                   #'deleted-elements
                                   #'testelements
                                   element))
        (result (cdr lstNresult)))

   (cond ((equal result 'null-dissection-tree))
         ((equal result 'not-found)
          (push element (added-elements CBNET)))
         ((equal result 'removed)
        (setf (deleted-elements CBNET) (car lstNresult))))))


(defmethod delete-element :after (element (CBNET Cooper-BNET))
  ;; modifies: (added-elements CBNET), (deleted-elements CBNET)
  ;; effects:  If (null-dissection-tree CBNET), then don't do anything;
  ;;          otherwise,
  ;;          #1. If element is in (added-elements CBNET), then remove
  ;;             element from (added-elements CBNET).
  ;;          #2. If element is not in (added-elements CBNET), then push
  ;;             element into (deleted-element CBNET).

  ;; ??? should I also delete all associated arcs here?
  ;;    think about the consequences when readd the node!  The user would have to
  ;;    readd all the arcs as well.  want them to do that?

  (let* ((lstNresult (searchNresult CBNET
                                    #'added-elements
                                    #'testelements
                                    element))
         (result (cdr lstNresult)))

    (cond ((equal result 'null-dissection-tree))
          ((equal result 'not-found)
           (push element (deleted-elements CBNET)))
          ((equal result 'removed)
           (setf (added-elements CBNET) (car lstNresult))))))



(defmethod add-arc :after ((CBNET Cooper-BNET) (source set-index)
                           (target set-index))
  ;; modifies: (added-arcs CBNET), (deleted-arcs CBNET)
  ;; effects:  If (null-dissection-tree CBNET), then don't do anything;
  ;;          otherwise,
  ;;          let arc=arc from source to target.
  ;;          #1. If arc is in (deleted-arcs CBNET), then remove
  ;;             arc from (deleted-arcs CBNET).
  ;;          #2. If arc is not in (deleted-arcs CBNET), then push
  ;;             arc into (added-arc CBNET).

  (let* ((s (element-with-index source CBNET))
         (tg (element-with-index target CBNET))
         (lstNresult (searchNresult CBNET
                                    #'deleted-arcs
```

```
                                 #'testarcs
                                 s
                                 tg))
      (result (cdr lstNresult)))

   (cond ((equal result 'null-dissection-tree))
         ((equal result 'not-found)
          (push (cons s tg) (added-arcs CBNET)))
         ((equal result 'removed)
          (setf (deleted-arcs CBNET) (car lstNresult))))))


(defmethod delete-arc :after ((CBNET Cooper-BNET) (source set-index)
                              (target set-index))
  ;; modifies: (added-arcs CBNET), (deleted-arcs CBNET)
  ;; effects:  If (null-dissection-tree CBNET), then don't do anything;
  ;;           otherwise,
  ;;           let arc=arc from source to target.
  ;;           #1. If arc is in (added-arcs CBNET), then remove
  ;;               arc from (added-arcs CBNET).
  ;;           #2. If arc is not in (added-arcs CBNET), then push
  ;;               arc into (deleted-arc CBNET).

  (let* ((s (element-with-index source CBNET))
         (tg (element-with-index target CBNET))
         (lstNresult (searchNresult CBNET
                                 #'added-arcs
                                 #'testarcs
                                 source
                                 target))
         (result (cdr lstNresult)))

   (cond ((equal result 'null-dissection-tree))
         ((equal result 'not-found)
          (push (cons s tg) (deleted-arcs CBNET)))
         ((equal result 'removed)
          (setf (added-arcs CBNET) (car lstNresult))))))
```

File: universe.lisp

Note: only functions that are new or modified from the previous version are shown.

```
(defmethod element-index (element (u universe-))
  (declare (optimize (safety 0) (speed 3)))
  (let ((i (call-next-method element u)))
    (if (or (null i) (b-in-set? i (deleted u)))
      nil
      i)))

(defmethod element? (e (u universe-))
  (not (null (element-index e u))))

(defmethod element-index-force (element (u universe-))
  (rass u element))

(defmethod element-with-index (n (u universe-))
  (ass u n))

(defmethod element-index-2 (element (u universe-))
  (declare (optimize (safety 0) (speed 3)))
  (rass u element))
```

File: vars.lisp

Note: only functions that are new or modified from the previous version are shown.

```
;; nodes module begins

(defmacro ClearKnownVal (node)
 `(change-node-val ,node (unknown-vals ,node)))


(defmethod change-node-val ((node discrete-rv)
                           new-val)

  (setf (known-vals node) new-val))


(defmethod vals-known? ((node discrete-RV))
  ;;Are all elements of (known-vals node) not an unknown element?

  (let ((val (known-vals node)))

    (if (null val)
      nil
      (let ((dim (num_dimensions val)))
        (do ((i 0 (1+ i)) (result t))
            ((or (= i dim) (null result)) result)
          (if (same-element? (ith-element val i) unknownelement)
            (setf result nil)))))))

(defmethod (setf vals-known?) (new-val (v discrete-RV))
  ;; We can set val-known? to false, which simply removes the existing val.
  ;; Setting it to true does not make sense; known-vals should be set
  ;; instead.  All we can do is check to see if in fact the value is known.
  (if new-val
    (assert (not (null (known-vals v)))
          ()
          "Cannot set val-known? of ~s true except by setting known-vals."
          v)
    (ClearKnownVal v)))


(defmethod num_universes ((node discrete-RV))
  ;; Returns the number of universes of node i.e. the appropriate dimension of
  ;; (known-vals node)
 (length (val-universes node)))

(defmethod known-vals-element-type ((node discrete-RV))
 ;; Returns the type of the elements of (known-vals node)
 (vals-element-type (known-vals node)))

(defmethod unknown-vals ((node discrete-RV))
 ;; Returns a (known-vals node) type object that corresponds to unknown value.

 (let* ((dim (num_universes node))
       (initialcontents (do ((i 0 (1+ i)) (result nil))
                          ((= i dim) result)
                         (setf result (cons unknownelement result)))))
   (make-known-vals dim initialcontents (known-vals-element-type node))))

;; nodes module ends



(defun make-known-vals (dimensions initial-contents element-type)
 ;; requires: initial-contents is passed as a list of initial contents, with
 ;;          the first content as the first element of initial-contents, etc.
 ;; effects:  Returns a known-vals object with dimension dimensions, initial
 ;;          contents initial-contents, and element type element-type.
 (make-array dimensions
            :initial-contents initial-contents
            :element-type element-type))

(defun vals-element-type (val)
 ;; ??? may need to change this code.
 'integer)


(defun same-known-vals? (val1 val2)
 ;; Does val1=val2?
 (let ((len1 (num_dimensions val1))
      (len2 (num_dimensions val2)))
   (if (neq len1 len2)
     nil
     (do ((i 0 (1+ i)) (result t))
        ((or (= i len1) (null result)) result)
       (if (not (same-element? (ith-element val1 i)
                               (ith-element val2 i)))
         (setf result nil))))))
```

```
(defun same-element? (e1 e2)
  ;; e1, e2 are some elements in a known-vals type.
  ;; Returns true if e1=e2; o.w. false.
  (= e1 e2))

(defun ith-element (val i)
  ;; Returns the ith element of val.
  (svref val i))


(defun num_dimensions (val)
  ;; returns the dimension of val (a known-vals type)
  (length val))

(defun known-vals2list (val)
  ;; converts val from a known-vals object to a list.
  (let* ((numdim (num_dimensions val))
        (result (do ((i 0 (1+ i)) (result nil))
                    ((= i numdim) result)
                  (setf result (cons (ith-element val i) result)))))
    (nreverse result)))



(defconstant unknownelement -1)
(defconstant kUnknownVal-1dim (make-known-vals 1 `(,unknownelement) 'integer))


(defclass equation-DRV (conditional-probability-equation-mixin discrete-RV)
  ())

(defclass conditional-probability-equation-mixin ()
  ((conditional-probability-equation :accessor conditional-probability-equation
                                     :initarg :conditional-probability-equation
                                     :initform nil))
  (:documentation "Equation for conditional probability for discrete-RV"))

(defmethod conditional-probability ((node conditional-probability-equation-mixin)
                                    (bayesnet Discrete-Bayes-Net)
                                    &optional
                                      parents-values-alist
                                      node-values-list)

  (cond ((and parents-values-alist node-values-list)
        (funcall (conditional-probability-equation node) node bayesnet
                 parents-values-alist node-values-list))
```

```
      (parents-values-alist
       (funcall (conditional-probability-equation node) node bayesnet
             parents-values-alist))
      (node-values-list
       (funcall (conditional-probability-equation node) node bayesnet
             node-values-list))
      (t    ;; when parents-values-alist and node-values-list are not provided
       (funcall (conditional-probability-equation node) node bayesnet))))



(defun get-value-indices (node valist)
  (let ((a (assoc node valist)))
    (cond (a (cdr a))
          (t (cassert (vals-known? node)
                      ()
                      "Value of node ~s must be known in ~
                       get-value-indices because it is not~
                       given in optional args."
                      node)
             (fast-known-vals node)))))


(defmethod conditional-probability ((node conditional-probability-table-mixin)
                                    (bnet Discrete-Bayes-Net)
                                    &optional
                                    parents-values-alist
                                    node-values-list)
  "Gives conditional probability of node given its parents.  If optional args are not
given, then node and parents must all have known values.  If given, args override
known vals."
  (unless node-values-list
    (assert (vals-known? node) ()
          "Value of node ~s must be known in conditional-probability~
             because it is not given by optional args."
          node)
    (setq node-values-list
          (fast-known-vals node)))
  (let (
      ; (prob NIL)
       (sorted-parents-values-alist
        (mapcar #'(lambda (par)
                    #| (cons par (get-value-indices par parents-values-alist)) |#
                    ;; less consing -- Ugh.
                    (let ((a (assoc par parents-values-alist)))
                 (or a
                    (progn
                      (assert (vals-known? par)
                              ()
                              "Value of parent node ~s must be known in ~
                               conditional-probability for ~s because it is not~
                               given in optional args."
                              par node)
                      (cons par (known-vals2list (known-vals par)))))))
           (parents bnet node)))
    )

;(setq prob
(apply #'aref
       (conditional-probability-table node)
       (append (apply #'append (mapcar #'cdr sorted-parents-values-alist))
               (known-vals2list (known-vals node))))
;)
;(format t "~& Prob is ~a" prob)
; prob
))
```