Automatic Layout for Pedigree Diagram

by

Ed Yampratoom

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1991


The author hereby grants to MIT permission to reproduce
and to distribute copies of this thesis document in whole or in part.
Automatic Layout for Pedigree Diagram

by

Ed Yampratoom

Submitted to the Department of Electrical Engineering and Computer Science
May 10, 1991
in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis describes the design and implementation of a program which generates layouts for pedigree diagrams. The program is able to deal with non-trivial complications such as: intergenerational marriage, consanguinity, multiple marriage, and marriages between clans. The diagrams generated is aesthetically pleasing, the measure of which I have also developed. The program is flexible and can be quickly modified to accommodate its user's idiosyncratic choices of symbols, layout styles, and interface requirements.

Thesis Supervisor: Peter Szolovits
Title: Associate Professor
Acknowledgments

I would like to thank Peter Szolovits, my thesis supervisor, for his support and advice, and my parents and sister, for their love and understanding.

The work reported here has been supported (in part) by grant R01 LM 04493 from the National Library of Medicine.
Contents

1 Introduction .................................................. 1
   1.1 Overview of the Thesis .................. 2
   1.2 Review of Literature .................. 2

2 Design and Specification ................................. 4
   2.1 Pedigree Diagram .................... 4
   2.2 General Description of the System .. 6
   2.3 Aesthetics .......................... 8
       2.3.1 Generation Placement ...... 9
       2.3.2 Spacing .................. 10
       2.3.3 Line crossings ........... 11
       2.3.4 Marriage Line .......... 14
       2.3.5 Order of Birth ........ 14
       2.3.6 Parents-Children Placement 15
       2.3.7 Nearness ................ 15

3 Implementation ........................................... 18
   3.1 Object Classes .................. 18
   3.2 Interface to GENINFER .......... 20
   3.3 How It Works .................. 21
       3.3.1 Prior Processing ........ 21
       3.3.2 Generation Assignment .. 23
       3.3.3 Laying out the Most Populated Generation 25
3.3.4 Laying Out Other Generations .......................... 30
3.3.5 Drawing .................................................. 37

4 Results ......................................................... 40
  4.1 Evaluation .................................................... 40
    4.1.1 Strong Points ........................................... 40
    4.1.2 Weak Points ............................................. 41
  4.2 Possible Future Modifications .............................. 41
    4.2.1 Functional Extensions .................................. 41
    4.2.2 Implementation Modifications ......................... 43
  4.3 Summary ..................................................... 44

Bibliography ................................................... 45
List of Figures

2-1 Common Pedigree Symbols ........................................ 5
2-2 Variations on Pedigree Symbols .................................. 6
2-3 Using Spouse’s Generation ....................................... 9
2-4 Inter-generational Marriage .................................... 10
2-5 Spacing Between Symbols ........................................ 11
2-6 Clan Marriages ..................................................... 12
2-7 Double Marriage .................................................. 12
2-8 Triple Marriage .................................................... 13
2-9 Another Triple Marriage ........................................ 13
2-10 Yet Another Triple Marriage ................................... 14
2-11 Drop Join vs. Straight Join .................................... 15
2-12 Centering of Parents ............................................. 16
2-13 Closely-Related Measure ....................................... 17

4-1 Circular Diagram .................................................... 42
Chapter 1

Introduction

A pedigree diagram is the tool that provides a genetic counselor with the most important information, the family history of the consultand. Traditionally, pedigree diagrams are drawn by hand after the counselor has collected the family genealogical history and information pertaining to each individual’s affection. The counselor then performs probabilistic calculations based upon the given information and the disease’s characteristic to arrive at his conclusions.

There are a few possible ways to perform calculations on pedigrees. A Bayesian approach, using probabilistic belief networks, can be very complex to do manually, even with the aid of a calculator. GENINFER, a program written by Nomi Harris [5], can help automate this calculation process. Its user interface, however, requires the counselor to type in names and other pieces of data for each individual in the pedigree. While sufficient, this method is tedious and not easy to use. Moreover, it does not provide the pedigree diagram for the counselor to check on its calculations.

A better interface should take advantage of the high-resolution graphical interface of modern personal computer by letting the counselor draw the pedigree with a mouse on the screen, and questioning the counselor for the required information. The pedigree diagram generated should be “prettified” by the computer so that an inherently complex diagram will still be relatively easy to read.
1.1 Overview of the Thesis

The next section provides a review of the literature in the field of automatic layout of pedigree diagrams and a related field of automatic layout of graphs. Chapter 2 specifies the problem in details and enumerates the requirements that the program has to satisfy. It also discusses the aesthetics issue and the design of the program. Chapter 3 describes the implementation of the program. Finally, Chapter 4 examines and evaluates the results and discusses future modifications to the program.

1.2 Review of Literature

A number of pedigree drawing programs have been written before[12, 8, 20]. PedDraw [8], running on the Apple Macintosh computers, is one of the more capable and complete. It is intended to be used, however, as a stand-alone program whose sole purpose is to generate diagrams and thus cannot be used as a front-end to GENINFER. In fact, none of the existing programs can be easily adapted to be used with GENINFER.

If we generalize a little, however, we can find many more applicable papers. Pedigree diagrams can be thought of as a specialization of graphs. The field of automatic graph drawing is much more fertile. Many papers were written about ways to draw graphs automatically. Some of the approaches are very general and deal with any kind of graphs[19, 15]. Some deal specifically with trees, especially binary trees[22, 10]. Some deal only with a rectilinear graph[6], an entity-relationship diagram[18], or a data flow diagram[2]. An implementation also exists that allows the user to edit and examine graphs while minimizing the number of edge crossings[13].

These papers present many significant concepts that can be applied in laying out pedigree diagrams as well. The most important concept is that of aesthetics, which was explicitly stated in[22, 10, 19] and implicitly assumed in other papers. Aesthetics in this case means the set of rules that the diagram has to satisfy in order for it to be judged acceptable (aesthetically pleasing). For example, in drawing a tree, which
is a planar graph, the edges should never cross[22]. Aesthetics can be thought of as a kind of heuristic for evaluating whether the drawn diagram is satisfactory. In some instances, all the aesthetic rules may not be satisfied at the same time because they are contradictory. These self-contradicting rules can still be used, however, if they produce satisfactory graphs most of the time.
Chapter 2

Design and Specification

This chapter discusses the problem and design issues in details. The first section describes the pedigree diagram and its conventions. The next section describes the system and its requirements. Finally, the last section discusses the concept of readability of diagrams, specifically, their aesthetics; aesthetics measures are also presented.

2.1 Pedigree Diagram

Genetic counselors use pedigree diagrams to construct a graphical record of the family health history. The relationships between members of the family are presented schematically. To be useful, the diagram must be arranged to aid easy viewing and analyzing.

There are many ways to draw a pedigree diagram. At present, an international standard for drawing and annotating pedigree diagrams does not exist. Some symbols are common to all schemes; a few others have a number of recognizable variants. The common symbols and notations are presented in Figure 2-1; existing variants are shown in Figure 2-2[4, 11, 9, 21, 3, 1].

Some definitions of the terms used in both Figures:

**consanguinity** The mating of two individuals who have one or more ancestors (with respect to the current pedigree diagram) in common. For example, marriages between first or second cousins are considered consanguineous.
Figure 2-1: The common pedigree symbols in most conventions. They are pseudo-standard because no standard exists.
heterozygous Having different alleles at a given locus on the same pair of chromosomes. In the context of genetic counseling, this usually means that the individual is the carrier of an autosomal recessive trait.

homozygous Having identical alleles at a given locus on the same pair of chromosomes. In the context of genetic counseling, this usually means that the individual is affected by an autosomal recessive trait.

proband The individual who first brings the family to attention for genetic evaluation and counseling; also called index case or propositus[3]. Note that the consultand is not necessarily the proband; for example, the consultand can be the parent of the proband.

twins Twins can be monozygotic (identical), or dizygotic (non-identical). Zygosity can also possibly be undetermined.

2.2 General Description of the System

GENInFER was first written for the Symbolics Lisp Machine in 1989[5] as a prototype
program. The Symbolics Lisp Machine, while being an excellent research and development tool on its own, is not a practical computer to be put in a typical genetic counselor's office. Fortunately, GENINFER was written in CommonLisp, a relatively portable language. Thus it was soon afterwards ported to the Apple Macintosh system running Macintosh Allegro Common Lisp.

While the program performs its calculation task satisfactorily, the tedious user interface prevents it from becoming a real workaday tool. To begin a session, the program will ask detailed questions about each individual in the family. People belonging to the earlier generations must be entered first to establish proper lineages from them to the later generations. Moreover, the program does not produce any pedigree diagram for the counselor to check on the calculations.

A better interface should take advantage of the high-resolution graphical interface of modern personal computer by letting the counselor draw the pedigree with a mouse on the screen, and questioning the counselor for the required information. At the present, a new implementation of GENINFER, GENINFER-II, has been developed for the Apple Macintosh computers[16]. The structure of of the program is now centered on a graphical user interface, which allow the user to design a pedigree diagram with great flexibility. The program is a functioning prototype which has all the capabilities of the original GENINFER. It will not automatically lay out the pedigree diagrams, however. The counselor will have to arrange the layout manually on the screen, using the mouse and a tool palette to draw the diagram.

Automating the task of laying out pedigree diagrams will then be a useful functionality to be added to GENINFER-II. The pedigree diagram generated should be "prettified" by the computer so that an inherently complex diagram will still be relatively easy to read. The pedigrees displayed by the program must be aesthetically pleasing. The aesthetic measures against which the diagrams will be judged and applied is described in the next section.
2.3 Aesthetics

The great value of the pedigree diagram to the counselor is that it provides, at the same time, detailed information about each individual and the conceptual structure of the family as a whole. The readability of the diagram is thus very important for the counselor to be able to grasp the structure of the pedigree easily. Readability can be characterized in term of two broad categories: semantics and aesthetics[18].

The diagram must be drawn with the knowledge of its meaning, or semantics. If the common, accepted standards in using symbols and connections are not followed, the readers of the diagram will certainly interpret the meaning of it differently from what the drawer intends. Thus there must be an explicit agreement on how to draw the diagram. This agreement can be thought of as a set of constraints within which the drawer has to stay when drawing, and the readers when interpreting. In the case of pedigree diagrams, these semantic-capturing constraints are evident when we examine the pedigree diagram:

- The meanings of symbols and connectives as described in Section 2.1.
- The hierarchical structure of the diagram, based upon the concept of generation. The diagram is thus separated into layers for easy analysis.
- The agreement on how the connectives are drawn. For example, the marriage line can be straight-joined or drop-joined (see Section 2.3.4). In this case, there is a choice between two constraints. Once a constraint is selected, however, it should be applied consistently.

In contrast to semantics, aesthetics is a somewhat fuzzy concept. Semantics can be specified using standards, while aesthetics can only use general guidelines. The rules of aesthetics are not set in stone, but rather are issues that the designer has to think about even if he eventually decides not to follow some of them. Aesthetics can be thought of as a kind of heuristic for evaluating whether the drawn diagram's readability is satisfactory.
Figure 2-3: The male pointed to by an arrow should belong to generation-II because he is married to a generation-II woman.

Usually, a number of aesthetic rules must be considered. Some of these rules can contradict one another in some, or even all, instances. The designer must then take into account the relative importance of each rules as applied to the situation and decide on the best combination. To some degree, aesthetics is subjective.

With the subjectivity of aesthetics in mind, I will now present a few aesthetic rules as applied to the pedigree diagram. All the rules should in fact be read with the suffix “as much as possible” appended to them because, as stated above, they can be contradictory and even impossible to follow. Yet they provide a good general guideline that will result in aesthetically pleasing diagrams.

2.3.1 Generation Placement

*People should be placed in the “correct” generation.*

In most cases, this rule is not hard to follow. It is useful to make one addition to the guideline in laying out people: If a person has no other direct relative in the pedigree diagram other than his/her own spouse, then that person is considered to belong to the generation of his/her spouse. Figure 2-3 illustrates this point.

However, if there is an inter-generational marriage, the correct generation of the
Figure 2-4: Inter-generational marriage occurs between the two shaded people, resulting in the blackened child. All three of them are assigned to the correct generations.

marrying couple and their children is not obvious. The couple are placed in the generation where they should be, had the marriage not occurred; that is, the marriage should not change their generation levels. The children resulting from inter-generational marriages should be placed in the generation right under that of the younger parent so that both parents will always be in the older generation than the children's. See Figure 2-4.

2.3.2 Spacing

The diagram should be as compact as possible and still be comfortable to read. The spacing should also look natural.

Laying the symbols out on a fixed grid is thus not satisfactory because the "natural-looking" diagram has a variable amount of spacing between the symbols. Examinations of a few diagrams yield the conclusion that this variable amount of spacing is dependent upon the number of people in the generation. If there are many
Figure 2-5: Generation-III is the most crowded generation. The space between people in generation-III is thus slightly smaller than that of other generations.

More people in generation-II than in generation-I, the spacing between people in generation-II should be slightly less than the spacing between people in generation-I. See Figure 2-5. Note that the space between people in generation-IV is also slightly more than the space in generation-III because of the same reason.

2.3.3 Line crossings

The numbers of connection crossings should be minimal.

There are many reasons for connection crossings. Some pedigree diagrams are inherently complex such that it is not possible to avoid crossings. For example, if there are group marriages between clans (see Figure 2-6), we cannot avoid some crossings without violating a number of other aesthetics. Although the diagram is still a planar graph, to redraw it without crossings will necessitate moving people into wrong generations—the parents of the shaded clan must be moved to the place below their children, for example.
Figure 2-6: Clan marriages. The crossing cannot be avoided.

Figure 2-7: How to draw a double marriage.

Multiple marriages can also wreak havoc to an otherwise nicely-drawn diagram. While double marriage can be easily accommodated by joining the twice-married person with each of the spouses on each side of the symbol (see Figure 2-7), there is no agreed upon way to draw multiple marriages that are of greater degree than two.

Some variations in drawing triple marriages are presented in Figures 2-8, 2-9, and 2-10.

Some semantic-related constraints also contribute to the increased number of crossings. For example, in some cases, the diagram is required to preserve the order of birth (i.e., the older child should be placed to the left of the younger ones). As another example, some people prefer that the male is placed to the left of the female
Figure 2-8: A way to draw a triple marriage. Note that the "spliced in" husband belongs to the same generation as the other husbands, but he is placed slightly below them.

Figure 2-9: Another way to draw a triple marriage. The curved line is used to jump the extra marriage line up in order to avoid crossings.
Figure 2-10: Yet another way to draw a triple marriage. This method only works for the drop-join marriage convention.

in marriage. Both constraints can severely limit the number of possible layouts and can eliminate the layouts in which there are no crossings.

2.3.4 Marriage Line

*Use straight join to signify marriage.*

Currently, there are two different conventions with regard to how the marrying persons are to be joined. Both methods are illustrated in Figure 2-11.

The current trend, especially in the United States, is the straight join method[8], which is also the preferred way of leading journals in medical genetics[1]. The straight join method has a distinct advantage is that it is less complicated, requiring fewer line segments. The drop method, needing three separate lines as compared to one line of the straight method, tends to clutter the diagram more.

2.3.5 Order of Birth

*Do not preserve order of birth unless it is necessary.*

Some pedigree diagrams require that the older child is placed to the left of the younger ones. Most of the diagrams, fortunately, do not have this requirement. The
order-of-birth constraint, if followed, can greatly constrict the freedom in laying out symbols and can result in many line crossings.

2.3.6 Parents-Children Placement

*Parents should be centered over their children.*

The reason for this rule is obvious. A skewed pairing between children and parents does not look pleasing. Note that, by *children*, we also include the children's spouses who do not have any parents in the pedigree diagram. Figure 2-12 illustrates this point. We simply embed the spouse into the family as if it is one of the siblings. The other spouse which has a parent does not need to be included in the centering process.

2.3.7 Nearness

*People who should be close to one another should be placed so.*

Some of the obvious examples: twins should be placed next to each other. To formalize this rule, the concept of *nearness* is developed. Nearness score is high for
Figure 2-12: Parents are centered over their group of children (and the children’s spouses).

people who are:\(^1\):

- Twins.
- Siblings.
- Spouses.
- closely related (a catch-all clause).

The measurement of “closely related” is done by counting the number of in-between people that we must pass through as we trace the most direct path from one person to another. See Figure 2-13 for an example. As there are only two possible kinds of paths (marriage and parent-child), this measurement is not difficult to perform.

\(^1\)the list is of decreasing importance
Figure 2-13: $A$ is 3 persons away from $B$. 
Chapter 3

Implementation

Because the layout program is designed to be used with GENINFER, it must be clearly written so that the task of coupling it to GENINFER will not be difficult. Also, as further modifications and extensions are probably needed in the future, the clarity of the program is essential in order for it to be useful.

3.1 Object Classes

The layout program employs a significant amount of object-oriented programming, via CLOS (Common Lisp Object System)[14, 7], mostly for clarity and ease of modifications. With the objects and methods clearly defined, the code is easy to understand. Features can be added, in most cases, by adding slots to the class and/or defining a few more methods to handle the new functionalities.

Two classes of objects are used in the program. The more important one is the person class. Currently, the person class has the following slots:

- **name**—the person’s name (a symbol).
- **sex**—the person’s gender (a symbol: can be male, female, or unknown.)
- **father**—the person’s father (a person object or nil).
- **mother**—the person’s mother (a person object or nil).
• children—a list of the person's children (a list of person objects or nil).

• spouses—a list of the person's spouses (a list of person objects or nil).

Additional slots such as age, disease type, genotype, phenotype, abortion, miscarriage, stillborn, infant death, pregnant, and so on can be included later as needed. The person class actually has a few more slots which are used internally for book-keeping and drawing tasks. These slots are:

• level—the person's generation number.

• pos-x—the person's location in the x-dimension on the plot.

• pos-y—the person's location in the y-dimension on the plot.

• seen?—a boolean marker used in measuring “closely-relatedness” (see Section 2.3.7).

Another class of object used in the program is the marriage class. The marriage class contains the following slots:

• groom—the male member of the marriage (a person object or nil).

• bride—the female member of the marriage (a person object or nil).

• clist—the list of children resulting from this marriage (a list of person objects or nil).

The marriage class contains all the information we need about how people related to one another in the pedigree. It can accommodate the case of a single parent (i.e., the other parent is not defined in the pedigree) by setting one of the groom/bride slots to nil as appropriate. Multiple marriages are handled gracefully likewise, because different marriages will create different pairs of bride and groom and thus we will not get confused by a groom whose name appears in five different marriage objects. A so-far childless marriage can also be noted by setting the clist slot to nil. Clearly, the marriage object is all that we need to keep track of relations between people.
3.2 Interface to GENINFER

All that is necessary to set up a pedigree is to define all person and marriage objects. The parent-child relation (i.e., the father, mother, and children slots) do not need to be specified as the program can figure it out from the marriage objects. The marriage-relation (spouses slot) is also explicitly specified by the marriage objects already and therefore is not needed.

There are three possible ways to define the person and marriage objects. The most straightforward method is to define them directly in Lisp:

(setf dad (make-instance 'person :name 'dad :sex 'male))
(setf mom (make-instance 'person :name 'mom :sex 'female))
(setf bro (make-instance 'person :name 'bro :sex 'male))
(setf sis (make-instance 'person :name 'sis :sex 'female))
(setf m (make-instance 'marriage
    :groom dad
    :bride mom
    :clist (list bro sis)))

This method, while perfectly useful for testing the layout algorithm, is not acceptable to be used by the counselor. With a few short procedures acting as a pure verbal front-end simply to eliminate the repetitive typing, however, the interface is up to the level of the original GENINFER, with less tedium.

With the minimal requirement that the user specify person and marriage objects (see Section 3.1), the program can also readily read in a text file of the format:

dad male
mom female
bro male
sis female
+++
;;; the +++ is placed between person and marriage definitions
dad mom bro sis
the marriage format is: groom bride child1 child2 ...

Similar to the querying interface, a few short routines are sufficient to convert between this file format and the underlying object definitions.

To be really useful, however, the program should be coupled to an interactive pedigree drawing tool. The interface will then allow the counselor to draw the pedigree on the fly as the family history of the consultand is being collected. The interactive pedigree tool has been written before[16]. The tool, as it currently exists, does not perform any automatic layout and thus is suitable to be coupled with the layout program.

### 3.3 How It Works

The layout program subdivides the task into five phases: prior processing, which performs bookkeeping chores necessary before any real work can be done; generation assignment, in which the people are assigned their generation levels; laying out the most populated generation, which lay out the crucial generation; laying out other generations; and drawing the pedigree on the screen. The rest of this chapter will describe each phase in details.

#### 3.3.1 Prior Processing

The program first needs to set up the slots of person objects to correctly reflect the relationships between people within the pedigree. The father, mother, children, and spouses slots are cleared. Note that two global variables are used to keep track of the person and marriage objects: *all-p* and *all-m*. Each one is a list that contain all the objects: person in *all-p*, and marriage in *all-m*.

The children slots are filled first by looking at the marriage objects, finding any marriage that results in children:

```lisp
(defun set-children ()
  (dolist (m *all-m*)
    (if (groom m) ; must check for existence
```
(setf (children (groom m))) ; because it may be nil.

(append (children (groom m))

(clist m))))

(if (bride m) ; similarly, we need to

(setf (children (bride m))) ; check the bride, too.

(append (children (bride m))

(clist m))))))

The mother and father slots are then easily filled because all the parent-child
information needed is already put in the children slots:

(defun set-parents ()

;; look at each person's children

;; and set the children's parent slot accordingly

(dolist (p *all-p*)

  (cond ((eq (sex p) 'female)

    (dolist (c (children p))

      (setf (mother c) p)))

    (t

      (dolist (c (children p))

        (setf (father c) p)))))))

The spouses slots are filled last by setting it according to the information stored
in the marriage objects.

(defun set-spouses ()

(dolist (m *all-m*)

  (when (and (groom m) (bride m)) ; when there is a marriage

    (setf (spouses (groom m))) ; set spouses respective slots

    (append (spouses (groom m)) (list (bride m))))

    (setf (spouses (bride m))

      (append (spouses (bride m)) (list (groom m)))))))
3.3.2 Generation Assignment

To assign people to generations, we first clear the level slot of each person, which contain the generation level value, to 0. The assignment algorithm makes use of a hash table, *gentab*. People are hashed into the table with their generation levels as keys. The *gentab* will also be useful later on. Two abstractions, get-genlist and set-genlist are used in reading from and writing to the hash table, respectively.

First, we tentatively assign people who do not have any parents within the pedigree to generation 0th. The key value of 0 will now fetch the list of these people.

(defun assign-level ()
  (let ((remain nil)) ; keep track of people unassigned
    ;; First, we put all the "orphan" nodes at generation 0th
    (dolist (p *all-p*)
      (cond ((and (null (father p))
                  (null (mother p)))
            (setf (level p) 0)
            (set-genlist 0 (append (list p) (get-genlist 0))))
            (t
             (push p remain))))))

Then, we go through the 0th generation and put all their children to the 1st generation. We then do this for the 1st generation to get the 2nd generation, and so on, until we have assigned generation levels to all people.

;;; When REMAIN is nil, we have seen all nodes.
;;; we also have to check that the last generation assigned
;;; are all leaves.
(setf *lastgen*
  (do ((gen 1 (+ gen 1)))
       ()
    (dolist (p (get-genlist (- gen 1)))
      (dolist (c (children p))
        (setf remain (remove c remain))

23
(setq (level c) gen)
(set-genlist gen
    (append (list c) (get-genlist gen))))
;; exit the do loop when we have seen all
(if (and (null remain) (all-leaf? gen))
    (return gen)))

After this initial assignment, there may be some duplicates in the generation table from two possible causes. Each generation may contain the same person twice because that person’s mother and father are both in the previous generation. In this case, we simply have to go through each entry in the hash table and remove any duplicates. Another cause for duplicates is due to the person’s father and mother being assigned to different generation. The person is then assigned to two entries in the hash table. According to the generation placement aesthetics (see Section 2.3.1), we should put the person lower than both parents. Thus the earlier generation duplicates are the ones that we remove.

;;; delete duplicated persons inside each level
(dotimes (gen (1+ *lastgen*))
    (set-genlist gen (remove-duplicates (get-genlist gen))))

;;; Inter-level duplicates

;;; We remove the higher generation duplicates.
(loop for gen from *lastgen* downto 0
    do (dolist (p (get-genlist gen))
        (loop for lower-g from (- gen 1) downto 0
            do (set-genlist lower-g
                (remove p
                    (get-genlist lower-g))))))

Each person should be placed in the same generation as his/her spouses, if possible. So we compare each person’s generation to his/her spouses’ generation. If they are inconsistent, shift the person who belong to an earlier generation down to the
appropriate level. We also need to propagate this generation change to that person’s children and parents, and update them as well.

(dolist (m *all-m*)
  (if (and (bride m) (groom m))
    (cond ((< (level (bride m)) (level (groom m)))
       (correct (bride m) (level (groom m))))
       ( (> (level (bride m)) (level (groom m)))
       (correct (groom m) (level (bride m)))))))

When the generation assignment procedure is done, the hash table *gentab* will contain all the people in the pedigree, with the key to the hash table mapping the generation number to the list of people in that generation.

### 3.3.3 Laying out the Most Populated Generation

Because the most populated generation is the crucial generation that determines how the whole pedigree is going to be laid out due to the spacing aesthetics (see Section 2.3.2), this generation will be the first to be laid out.

To find the most populated generation:

(defun most-pop ()
  ; return the most populated generation
  (let ((pop 0) (max -1))
    (dotimes (i (1+ *lastgen*))
      (when (> (length (get-genlist i)) max)
        (setf max (length (get-genlist i)))
        (setf pop i)))
    pop))

The layout algorithm makes use of another hash table, *poptab*, which will contain groups of people. People are grouped according to their nearness (see Section 2.3.7). Generally, people in the same group are siblings, but there can also be embedded spouses (see Section 2.3.6).

First, we tentatively group people according to their parents. So we use a cons of
father and mother as the key.

(defun group-pop ()
  (setq *poptab* (make-hash-table :test #'equal))
  (setq *popkeys* nil)
  (let ((plist (get-genlist (most-pop))))
    (dolist (p plist)
      (let ((k (cons (mother p) (father p))))
        (setq (gethash k *poptab*)
          (append (list p)
            (gethash k *poptab*)))
          (pushnew k *popkeys* :test #'equal))))

The variable *popkeys* is used to keep the list of all the keys used in the *poptab* hash table.

To embed spouses, we merge spouses into the biggest group. Then, we merge any leftover spouses into the other groups. To find the biggest group, we use:

(defun bgg-key ()
  ; return the key to *pop-tab* for the largest group
  (let ((key nil) (max 0))
    (dolist (k *popkeys*)
      (when (and (> (length (gethash k *poptab*)) max)
          (not (equal k (cons nil nil))))
        (setf key k)
        (setf max (length (gethash k *poptab*))))
    key))

The reason we filter out the group with the key of (cons nil nil) is that this group consist of orphans (people with no parents in the pedigree). They are not closely related to each other but are lumped together only because of them being orphans. Thus we don’t want them to be the core group, no matter how numerous they are.

Merging into the biggest group is done by looking for spouses of each person in the biggest group that also belong to this generation. The spouses are then removed
from their group and placed into the biggest group, wedged next to the person he/she marries.

(defun merge-pop ()
  (let ((bg (gethash (bgg-key) *poptab*)))))
  (dolist (p bg)
    (when (spouses p)
      (dolist (s (spouses p))
        (dolist (k *popkeys*)
          (when (member s (gethash k *poptab*))
            (setf (gethash k *poptab*)
              (remove s (gethash k *poptab*)))))
          (setf (gethash (bgg-key) *poptab*)
              (insert-after s
                (position p (gethash (bgg-key) *poptab*))
                (gethash (bgg-key) *poptab*)))))
        )))
  )))

The procedure insert-after is defined as:

(defun insert-after (thing place list)
  ;; return the list with THING inserted after PLACE position
  (append (subseq list 0 place)
    (list thing)
    (subseq list place)))

Merging spouses into the other groups is done similarly. We hide the biggest group (so that the people in it are not moved out) by simply removing the biggest group's key from the key list.

(let ((keys (remove (bgg-key) *popkeys* :test #'equal)))
  (dolist (k keys)
    (dolist (p (gethash k *poptab*))
      (when (spouses p)
        (dolist (s (spouses p)))
      ))
(dolist (other-k (remove k keys :test #'equal))
  (when (member s (gethash other-k *poptab*))
    (setq (gethash other-k *poptab*)
      (remove s (gethash other-k *poptab*)))
    (setq (gethash k *poptab*)
      (insert-after s
        (position p (gethash k *poptab*))
        (gethash k *poptab*))))
)))

We do not need to worry about a spouse being pulled back and forth indefinitely because, once it is moved, it will be hidden and thus immune from being pulled out again. Hiding is done by removing the key to its new group from the key list.

A function is also defined to help with the closely-related measure of the nearness aesthetics (see Section 2.3.7). Hop takes as arguments two person objects and return the closely-related measure between the two persons.
(defun hop (source destination)
  (clear-marker)
  (mark source)
  (if (eq source destination)
    0
    (hop-helper (expand source 0) destination)))

The clear-marker procedure performs the bookkeeping task of clearing the seen? slot of each person. The search performed is a BFS and the expand procedure sets up the queue of persons to be searched. It takes as input the current person being examined, and his/her distance from the source, and return a list of the person’s mother, father, spouses, and childrenl, with each of these direct relatives cons’ed with their distances from the source.
(defun expand (source distance)
  ; Expand the source and return the list of
  ;( (mother.d+1) (father.d+1) (spouse1.d+1)
; (spouse2.d+1) ... (child1.d+1) (child2.d+1) ...)
; The relatives who don't exist in the pedigree will
; not be included in the list.
(let ((d (1+ distance))
     (m (mother source))
     (f (father source))
     (c (children source))
     (s (spouses source)))
(remove (cons nil d)
     (append (list (cons m d) (cons f d))
             ; spouses & children are spliced out
             (mapcar #'(lambda (p) (cons p d)) c)
             (mapcar #'(lambda (p) (cons p d)) s))
     :test #'equal)))

The procedure that does the real work of measurement is hop-helper, which is a
bread-first search.

(defun hop-helper (searchlist destination)
  ;; do a BFS
  (let ((newlist nil))
    (dolist (chunk searchlist)
      (if (eq (car chunk) destination)
          (return-from hop-helper (cdr chunk)))
      (dolist (chunk searchlist)
        (unless (seen? (car chunk))
          (mapcar #'(lambda (i) (pushnew i newlist :test #'equal))
                  (expand (car chunk) (cdr chunk)))))
    (hop-helper newlist destination)))

Finally, we actually lay out the most populated generation by assigning numerical
coordinates to the people. We can assign the number directly because, if later on we
ran out of place to draw, the position assigning procedure will simply call a "trap"
procedure which shift all people's position in the right direction to regain enough drawing space. People grouped together will be placed a little closer than people belonging to different groups.

3.3.4 Laying Out Other Generations

After laying out the most populated generation, we lay out the other generations, using the most populated generation as the reference point. First, we lay out the earlier generation, starting from the one closest to the most populated one, going from the bottom up. Then we lay out the later generation, also starting from the one closest to the most populated one, going from the top down.

(defun layout-rest ()
  (loop for g from (1- (most-pop)) downto 0
    do (lay-up g (* (- (most-pop) g) 75)))
  (loop for g from (1+ (most-pop)) to *lastgen*
    do (lay-down g (* (- g (most-pop)) 75))))

The loop primitive can deal with the case of most-pop returning 0 (the most populated generation being the topmost one), and will not erroneously get into an infinite loop. The variable *lastgen* keeps track of the maximum generation value used in the pedigree (see Section 3.3.2).

Both lay-up and lay-down also makes use of a hash table, *temptab*, which performs similar task to that of the *poptab* hash table. The lay-temp function is used to set up this hash table by grouping and merging people like group-pop and merge-pop (see Section 3.3.3).

(defun lay-temp (people)
  ;; reset the table first of all
  (setf *temptab* nil)
  (setf *tempkeys* nil)
  (group-temp people)
  (merge-temp))
(defun group-temp (people)
  (setq *temptab* (make-hash-table :test #'equal))
  (setq *tempkeys* nil)
  (dolist (p people)
    (let ((k (cons (mother p) (father p))))
      (setq (gethash k *temptab*)
        (append (list p)
          (gethash k *temptab*)))))
  (pushnew k *tempkeys* :test #'equal)))))

(defun merge-temp ()
  (let ((bg (gethash (tbgg-key) *temptab*)))
    (dolist (p bg)
      (when (spouses p)
        (dolist (s (spouses p))
          (dolist (k *tempkeys*)
            (when (member s (gethash k *temptab*))
              (setq (gethash k *temptab*)
                (remove s (gethash k *temptab*)))))
            (setq (gethash (tbgg-key) *temptab*)
              (insert-after s
                (position p (gethash (tbgg-key) *temptab*))
                (gethash (tbgg-key) *temptab*))))
      ))))))
  (let ((keys (remove (tbgg-key) *tempkeys* :test #'equal)))
    (dolist (k keys)
      (dolist (p (gethash k *temptab*))
        (when (spouses p)
          (dolist (s (spouses p))
            (dolist (other-k (remove k keys :test #'equal))
(when (member s (gethash other-k *temptab*))
    (setf (gethash other-k *temptab*)
        (remove s (gethash other-k *temptab*)))
    (setf (gethash k *temptab*)
        (insert-after s
            (position p (gethash k *temptab*))
            (gethash k *temptab*)))))

(defun tbgg-key ()
    (let ((key nil) (max 0))
        (dolist (k *tempkeys*)
            (when (and (> (length (gethash k *temptab*)) max)
                (not (equal k (cons nil nil))))
                (setf key k)
                (setf max (length (gethash k *temptab*))))
        key))

To lay up, first we set up the *temptab* by calling lay-temp on the people belonging to this generation. Then we set the x-pos slot of each person to nil, which marks it as being unassigned. We then find out whether they have any children in the generation just below them. If so, center these parents over their children. If the parent is a lone parent (i.e., the other parent is not in the pedigree), we simply assign the average of the x-coordinates of the children. If both parents are present, we put both of them slightly off the average, one on each side. After these parents are assigned, the only people left are those with no children in the generation below; they usually are sterile siblings of these parents. The "fertile" person will get his/her x-coordinate basing upon the x-coordinate of the closest relative in this generation who has an x-coordinate assigned.
(defun lay-up (generation y)
  (let ((people (get-genlist generation)))
    (dolist (p people)
      (setf (y-pos p) y)
      (setf (x-pos p) nil)) ; clear them all
    (lay-temp people)
    (dolist (k *tempkeys*)
      (let (((group (gethash k *temptab*)))
            (dolist (p group)
              (when (null (pos-x p))
                (if (has-children-in? p (1+ generation))
                  (if (find (spouse p) people)
                    (progn
                      (setf (x-pos p)
                          (- (middle (children p)) 50))
                      (setf (x-pos (spouse p))
                          (+ (middle (children p)) 50)))
                    (progn
                      (setf (x-pos p) (middle (children p)))))))))
      ;; sterile people look for closest relative
      (dolist (p group)
        (if (null (pos-x p))
          (assign-from-closest-one (position p group)
            group)))))))

The procedure has-children-in? simply check whether a given person has any child in a specified generation:
(defun has-children-in? (person child-gen)
  (let ((people (get-genlist child-gen)))
    (dolist (c (children person))
(if (find c people)
    (return-from has-children-in? t)))
nil))

The procedure middle takes a list of people and return their average x-coordinate.

(defun middle (people)
  (let ((x 0))
    (dolist (p people)
      (setf x (+ x (pos-x p))))
    (/ x (length people))))

The procedure assign-from-closest-one takes a group of people and the position of the unassigned person. It then assigns an x-coordinate to the unassigned person by looking for the closest person to the unassigned person who has his/her x-coordinate assigned. If the closest person is to the left of the unassigned person, then we assign a position slightly to the right of the found x-coordinate to the unassigned person, and conversely.

(defun assign-from-closest-one (here people)
  (let ((c (find-closest here people)))
    (if (< c here)
      (setf (x-pos (nth here people))
            (+ (x-pos (nth c people)) 50))
      (setf (x-pos (nth here people))
            (- (x-pos (nth c people)) 50)))))

The procedure find-closest returns the position of the closest person having the x-coordinate assigned. It compares the closest person to the left to the closest person to the right, and return the position that is closer. Note that sometimes a search in one direction will return nil, meaning that there is no one in that direction of the unassigned person who has his/her x-coordinate assigned.

(defun find-closest (here people)
  (let ((left (closest-left here people))
        (right (closest-right here people)))
(cond ((null left)
   right
   ((null right)
     left
     ( (> (- here left) (- right here))
     right)
     t)
   left)))))

The procedures closest-left and closest-right returns the closest position to the left and right, respectively. If no one in that direction has his/her x-coordinate assigned, the procedures will return nil, which will be caught by find-closest (see above).

(defun closest-left (start people)
  (loop for place from start downto 0
    do (if (pos-x (nth place people))
      (return-from closest-left place)))
  nil)

(defun closest-right (start people)
  (loop for place from start to (1- (length people))
    do (if (pos-x (nth place people))
      (return-from closest-right place)))
  nil)

To lay down, first we also set up the *temptab* by calling lay-temp on the people belonging to this generation. Then we set the x-pos slot of each person to nil, which marks it as being unassigned. We then find out whether they have any parents in the generation just above them. If so, spread these children under the parents. If the parent is a lone parent (i.e., the other parent is not in the pedigree), we simply spread the children under him/her. If both parents are present, we spread the children under the average of the parents’ x-coordinates. After these children are assigned, the only people left are those with no parents in the generation above; they usually
are embedded spouses of these children. These people will get their x-coordinates basing upon the x-coordinate of the closest relative in this generation who has an x-coordinate assigned.

(defun lay-down (generation y)
  (let ((people (get-genlist generation)))
    (dolist (p people)
      (setf (y-pos p) y)
      (setf (x-pos p) nil)) ; clear them all
    (lay-temp people)
    (dolist (k *tempkeys*)
      (let* ((group (gethash k *temptab*))
            (mom (group-mother group))
            (dad (group-father group)))
        (cond ((null mom)
               (spread group (x-pos dad)))
              ((null dad)
               (spread group (x-pos mom)))
              (t
               (spread group (/ (+ (x-pos mom)
                                (x-pos dad)) 2)))))
    (dolist (p group)
      (if (null (pos-x p))
        (assign-from-closest-one (position p group)
                                  group))))

The "real" father of the group is the father that most people in the group share.

Similar case applies to the mother.

(defun group-father (group)
  (let* ((dad nil)
         (current-count 0)
         (fatherlist (mapcar #'father group))
         (n (length (fatherlist)))
         (max 0)
         (d 0)
         (loop for x from 0 to (1- n)
               do (let ((p (fatherlist x)))
                    (setq d (max d (count (member p group))))
                    (if (= d max)
                        (setq max d)
                        (setq max 0)
                    )
               )))

  (defun group-mother (group)
    (let* ((mom nil)
           (current-count 0)
           (motherlist (mapcar #'mother group))
           (h (length (motherlist)))
           (max 0)
           (m 0)
           (loop for y from 0 to (1- h)
                 do (let ((q (motherlist y)))
                      (setq m (max m (count (member q group))))
                      (if (= m max)
                        (setq max m)
                        (setq max 0)
                      )
                 )))
  )

  (defun position (p group)
    (cond ((null group)
           'not-found)
          ((eq p group)
           0)
          ((< (x-pos p) (x-pos group))
           (1+ (position p (next group)))
           )
          ((< (y-pos p) (y-pos group))
           (1+ (position p (next group)))
           )
          (t
           (1+ (position p (next group)))
           )))

  (defun assign-from-closest-one (position group)
    (cond ((null (rest group))
           group)
          ((null (next group))
           group)
          (t
           (assign-from-closest-one (next group) group)))

  (defun next (group)
    (let ((first nil)
           (firstpos nil))
      (loop for x from 1 to (length group)
            do (let ((p (nth x group)))
                 (setf firstpos (position p group))
                 (setf first p)))
      (if firstpos
          (nth firstpos group)
          group))

  (defun count (group)
    (length group))

  (defun next (group)
    (let ((next-group nil)
           (first (first group)))
      (loop for x from 1 to (length group)
            do (let ((p (nth x group)))
                 (if (position p group)
                     (setf next-group (nconc next-group p))
                     (setf next-group p))
                 (setf first p)))
      next-group)

  (defun gethash (k keys)
    (find k keys))

  (defun get-genlist (gen)
    (let ((group (ref (list (list) (list)))))
      (group (mapcar (lambda (p)
                        (cons p (ref group))))
             gen))))
(fatherset (remove-duplicates fatherlist)))

(dolist (f fatherset)
  (when (> (count f fatherlist) current-count)
    (setf dad f)
    (setf current-count (count f fatherlist))))

dad))

(defun group-mother (group)
  (let* ((mom nil)
         (current-count 0)
         (motherlist (mapcar #'mother group))
         (motherset (remove-duplicates motherlist)))
    (dolist (m motherset)
      (when (> (count m motherlist) current-count)
        (setf mom m)
        (setf current-count (count m motherlist))))
    mom))

Spreading out the children under an x-coordinate is done by the spread procedure.

(defun spread (people x)
  (let ((n (length people)))
    (loop for pos from (* -1 (/ (- n 1) 2)) to (/ (- n 1) 2)
          for place from 0 to (- n 1)
          do (setf (pos-x (nth place people))
                   (+ x (* 50 pos))))))

3.3.5 Drawing

When we get to this point, each person in the pedigree will have his/her coordinates set. Each person is then drawn by calling a draw method on each person. The draw method, in skeletal form is presented:
(defmethod draw ((p person) (v view))
  (let ((x (pos-x p))
        (y (pos-y p)))
    (cond ((eq (sex p) 'unknown)
           (move-to v x (- y 10))
           (line v 10 10) (line v -10 10)
           (line v -10 -10) (line v 10 -10))
         ((eq (sex p) 'male)
          (frame-rect v (- x 10) (- y 10)
                        (+ x 10) (+ y 10)))
         ((eq (sex p) 'female)
          (frame-oval v (- x 10) (- y 10)
                        (+ x 10) (+ y 10))))))

Extending this skeletal form to account for all the symbols and variations is easy. The method itself can be modified, or a few :before and :after methods can be added, for example, to make the program display the person's name next to the symbol. An :after method to display the person's name and add the crossed-out symbol to signify a deceased person is shown:

(defmethod draw :after ((p person) (v view))
  (let ((x (pos-x p))
        (y (pos-y p)))
    (when (and (dead? p) (not (aborted? p)))
      (move-to v (+ x 15) (- y 15))
      (line v -30 +30))
    (when (name p)
      (move-to v (+ x 10) (- y 10))
      (format v (string (name p)))))

The marriage lines are easily drawn by looking at the marriage objects and join all the brides and grooms, using the marry method.
(defmethod marry ((m person) (f person) (v view))
  (if (eq (sex m) (sex f))
      (error "-s and -s are of the same sex."
        (name m) (name f)))
  (if (or (eq (sex m) 'unknown) (eq (sex f) 'unknown))
      (error "One of the partner is of unknown sex.")
  (if (eq (sex m) 'female)
      (marry f m v))
  (let ((mx (pos-x m)) (my (pos-y m))
        (fx (pos-x f); (fy (pos-y f)))
      (when (< mx fx)
        (move-to v (+ mx 10) my)
        (line-to v (- fx 10) fy))
    (when (< fx mx)
      (move-to v (+ fx 10) fy)
      (line-to v (- mx 10) my))))

Consanguinity is drawn in almost the same way, except that we use a double line.

By looking at the marriage objects again, we grow a short vertical line extending up from each children, and a horizontal line is drawn, for each marriage, to join these small vertical lines if there is more than one child. A vertical line is then drawn from the average x-coordinates of the bride and groom down to this horizontal line. The diagram is done.
Chapter 4

Results

4.1 Evaluation

4.1.1 Strong Points

The best part of the program is its clarity and flexibility. The code is clear enough to be read at a single sitting. The use of objects and the separation of different functionalities make it easy to modify and interface the code.

There is no inherent limit on the program's capacity. The hash tables used can automatically grow as they are overfilled by a large pedigree. Actually, the program currently can handle 50 people/pedigree, which is quite a large number. Very few pedigrees, if any at all, will need larger hash tables than the current ones. The drawing trap procedure, mentioned in Section 3.3.3, will also correctly adjust the drawing area if the pedigree is so large or unbalanced that the diagram spills over the current display area.

Because of the numerous heuristics used, the program gains in speed with just a little lost in the quality of the diagram. In fact, only an exhaustive generate and test approach (see Section 4.2.2) can adequately handle the pedigrees that cause troubles with our program.
4.1.2 Weak Points

The program is not yet coupled to GENINFER and the interactive pedigree drawing tool. The full testing of the system is thus not possible until all three modules are put together.

The layout algorithm, due to its consideration for speed, uses heuristic in some places. These heuristics occasionally fail, resulting in a number of less-than-optimal diagrams. To be fair, however, the pedigrees that create this problem are almost always inherently complex and the optimal diagrams are not intuitively obvious even to a human drawer.

4.2 Possible Future Modifications

4.2.1 Functional Extensions

The most needed extension to the program is interfacing it to GENINFER. The program's transparency and flexibility should make that task easy, however. A few front-end routines are needed to process the user input into a pedigree (see Section 3.2). They are not written because, as of now, I do not have a clear idea of how GENINFER is going to be used by the counselor. Once the requirements for the workaday GENINFER are specified, however, the routines should be easy to write.

The more interesting extension is interfacing the program to the interactive pedigree drawing tool. As with the interface to GENINFER, the interfacing routines are not written out because the working requirements of how the program is going to be used are yet to be specified.

One usual, but possible extension, which may or may not be useful, is that of a circular diagram. Figure 4-1 shows an example of such a diagram. This diagram is a compact way to present a very large pedigree. A number of aesthetics will have to be revised to accommodate this new style, however.
Figure 4-1: Circular Pedigree Diagram[21].
4.2.2 Implementation Modifications

The current method of layout relies heavily on the most populated generation as a heuristic on where to place people. This heuristic works well almost all of the time because the most populated level is almost always the one that is the most complex (containing large numbers of connections). The heuristic can sometimes fail, however.

If we want to generate the "best" diagram possible, it might be better if, instead of using the nearness measurement (see Section 2.3.7) to guess the best layout, we generate all possible layouts from the starting set of people in the most populated generation by looking at all possible (and not obviously stupid) permutations of them. The layouts are then compared using the rules of aesthetics as specified in Section 2.3. The most aesthetically pleasing diagram is then chosen to be displayed.

There is a slight problem with this approach, however. The pedigrees that cause the nearness heuristic to fail tend to contain a large number of people and connections. In this case, the time taken to generate and test so many possible permutation on the most populated level will be intolerably long. This approach is thus not suitable for interactive layout updating and display. It can be quite useful, however, if we want a pretty display after we have completed drawing the whole pedigree.

We can also apply graph algorithms, especially planarization algorithms, to the layout problem. By separating the diagram into planar subgraphs, the crossings problem should be minimized. But there is a difficult problems in applying planarization to pedigree diagrams. Although pedigree diagrams are special cases of graphs, some aesthetic considerations are more important than edge crossings. For example, the rule requiring higher position of a parent with regard to its children is quite inflexible; the pedigree diagram might otherwise be very difficult to read. We can, of course, adapt the algorithm to accommodate these higher rules but it is not clear how the algorithm should be modified. Moreover, a planarization algorithm can be very time-consuming. We may conceivably take advantage of the new development of a near-optimum parallel planarization algorithm which runs in $O(1)$ time[17]. But the approach requires parallel computations to provide a decent simulacrum of a big neural network array, making the approach impractical because of GENINFER's personal
computer hardware base. Adapting this new parallel algorithm to the constraints of pedigree diagrams may also be harder than adapting the normal, sequential planarization algorithms; it is not clear how we can adjust the numerous coefficients in the planarizing neural network in order to stay within the constraints.

One promising implementation modification is in the implementation of the nearness aesthetics (see Section 2.3.7). Currently, the nearness score is actually an estimate rather than an exact numerical score. To fully apply the nearness aesthetics, we must calculate the nearness values between each pair of persons, for all persons in the pedigree. We can then apply an assignment algorithm, probably a variation of linear programming, to find the optimal layout. The problem, again, is speed. This approach appears to take shorter time than the generate and test approach mentioned above, however, especially if the linear programming part is optimized enough for speed.

### 4.3 Summary

In this thesis I have discussed the issues in the layout of pedigree diagrams. The concept of aesthetics is created and categorized in details. A program to perform automatic layout of pedigree diagrams is implemented according to the aesthetics rules specified. The program is able to deal with non-trivial complications and generate aesthetically pleasing diagrams. The program is flexible and can be quickly modified to accommodate its user's idiosyncratic choices of symbols, layout styles, and interface requirements.
Bibliography


