Computer Notes

An Efficient Algorithm for the Additive Kinship Matrix

V. Backus and M. Gilpin

In this article we show how object-oriented programming can provide an efficient method for calculating kinship coefficients for very large pedigrees—large in number of individuals or generations, or both. We call our approach the "compressed kinship matrix." We use Java as our object-oriented language, but the algorithm should be similarly implemented in other object-oriented languages. The documented source code and illustrative Java applets are available on our Web site: www.consbio.com/kinshipAlgorithm.

Specification of the genealogical relationships between all individuals in a population is the most complete and fundamental nonempirical genetic analysis one can perform (Lacy et al. 1995). Kinship, the probability that two individuals share alleles identical by descent, plays a central role in the study of these relationships (Thompson 1976). Although first utilized for human pedigrees, it is also important for the genetic management of animals in captive or domesticated settings for which exact paternity information is available.

The kinship between two individuals is equal to the inbreeding coefficient an offspring of theirs would have regardless of whether the two individuals have actually mated. Boyce (1983) reviews the two main computation approaches for calculating inbreeding and kinship coefficients: path analysis and recursive algorithms. Path analysis algorithms lend themselves to computational problems on extended pedigrees owing to the large number of paths that need to be generated, stored, and searched. For pedigrees of substantial depth, programming of the recursive algorithm is more straightforward.

The recursive algorithm utilizes the fact that the kinship between two individuals, x and y, can be expressed in terms of the kinship between one of them, the elder individual, and the mother and father of the younger. That is to say, the kinship between two individuals, x and y, denoted $\psi(x, y)$, can be defined by

$$\psi(x,y) = 0.5(\psi(x,\phi(y)) + \psi(x,\rho(y)), \quad (1)$$

where x is the same age or older than y, and $\phi(y)$ and $\rho(y)$ are the mother and father of y, respectively. Kinship between two individuals is a symmetrical relationship, that is, $\psi(x, y) = \psi(y, x)$; however, the correct calculation using recursive algorithms requires that the first individual, x, be the older in equation (1).

If x and y are the same individual, then equation (1) becomes

$$\psi(x,x) = 0.5(1 + f_x), \quad (2)$$

where f_x is the inbreeding coefficient of x. Unless otherwise known, the initial individuals founding a pedigree are assumed to be noninbred and have coefficients of kinship equal to zero. This assumption, along with equations (1) and (2), allows the recursive calculation of $\psi(x, y)$ to be implemented in any computer language that supports recursively defined functions. It also permits a much faster calculation assuming all the kinships of older animals are stored; that is, if $\psi(x, \phi(y))$ and $\psi(x, \rho(y))$ are accessible as elements of a table or matrix, they can be quickly retrieved and thereby directly utilized to determine $\psi(x, y)$.

There is a practical issue regarding the use of kinship coefficients. The computer resources necessary to recursively compute $\psi(x, y)$ values depend on the depth of the pedigree and goes up roughly with the power of the depth of the pedigree. A different approach is to store the kinship relationships in a symmetric square matrix, termed the additive matrix (Ballou 1983). The number of rows and columns in the matrix are equal to the number of animals in the pedigree. The number of rows and columns scale exactly with pedigree size, but the size of the kinship matrix itself goes up with the square of the size of the pedigree. Both of these issues can lead to storage or stack overflow problems inside digital computers. We offer in this note an efficient algorithm that avoids these two issues.

Our algorithm is similar in concept to the additive matrix approach, so there is no problem with deep recursion. However, the method avoids having to store all pairwise kinship coefficients because it constantly discards all those corresponding to dead animals. For most management or analysis considerations, only the kinships between currently living animals are required. That is, if y is a newly born animal, its kinships with all living animals x can be calculated as the kinship between x and the mother of y and between x and the father of y, all of whom have to be living. This saves a substantial amount of computer memory; for example, if in a pedigree of 1,000 individuals only 100 are currently alive, then only a 100×100 matrix is required (10,000 elements), not a $1{,}000 \times 1{,}000$ matrix (1,000,000 elements). We refer to the kinship matrix of only the living animals as the "compressed kinship matrix" and our approach as the "compressed kinship matrix algorithm," even though, as discussed below, its implementation does not involve traditional, doubly subscripted arrays.

The technique of compressing the size of the matrix by storing only the kinship coefficients relating to living or breeding

individuals instead of the full pedigree matrix itself is not new (e.g., Lacy 1993a,b). Implementation of any such compression algorithm involves two programming issues: indexing and matrix manipulation. Additionally, depending on the particular language, there can be issues of memory allocation and deallocation. Keeping track of only living animals means the kinship in the *i*th row and *j*th column of the compressed kinship matrix will not necessarily be the kinship between animals *i* and *j*, where *i* and *j* are the birth order identification numbers that begin with the founders of the pedigree. Accessing kinship values in the array will thus require a one-to-one mapping between the living animals' identification numbers and the compressed kinship matrix indices. Keeping track of only the living animals also means the compressed kinship matrix will have to be manipulated for each birth and each death. New rows and columns will need to be added when new animals are born; this will not be difficult, as these rows and columns will be added at the far right and bottom edge of the matrix. Deaths of animals, however, necessitate the removal of internal rows and columns. Removing internal rows and columns means all elements with higher index numbers in the matrix will have to be shifted up, to the left, or both, a very computer-resource intensive manipulation.

Conventionally solving these two programming issues involves writing explicit programming code to manually adjust the subscripts in a two-dimensional array. Our object-oriented solution to these programming issues avoids writing code for the manipulation of individual matrix elements by utilizing standard classes and methods found in the programming language Java 2 SDK, standard edition, version 1.2.2. Other object-oriented programming languages (OOPs) would doubtlessly work in much the same way. We have developed this approach for use in individual-based animal simulation studies of genetic management strategies, and this will be the context for our discussion below.

In object-oriented programming languages, "objects" are created or "instantiated" from classes. In the most generic sense, classes are similar in architecture to subroutines of procedural languages, since they contain both variables and methods that perform some operation on or with the variables. In Java, some classes and their variables can be used directly; however, for other classes, one must create an instance of the class. This instance is referred to as an object. OOPs allow instantiation of numerous objects of the same class. The Java language provides some classes and it likewise allows individual programmers to construct their own. We utilize two standard Java classes, *Vector* and *Double*, and design one custom class, *Animal*, to construct our "compressed kinship matrix."

Tables in spreadsheet software packages and arrays in procedural programming languages usually have fixed dimensions and any manipulation of their size and shape usually has to be carried out with user-written code that requires great computer resources. Java has a dynamic array class, *Vector*. *Vector*s are one-dimensional constructs that store objects as indexed lists. Java provides predefined methods for manipulating the indexed lists. To form a two-dimensional structure, one needs to have a *Vector* object that contains other *Vector* objects. While this can be done, we use a different approach to construct our "compressed kinship matrix." We let each individual in a population keep track of its kinship relationships with other members of the population in its own *Vector* object. By doing this, we decompose the additive kinship matrix—a traditional, doubly subscribed array—into individuals with indexed lists. To do this, we first create an *Animal* class that has the variables id, mom, dad, and kinshipWith. The variable id is an integer representing an *Animal* object's identification number, and the variables mom and dad are integers representing the identification numbers of its mother and father, respectively. The variable kinshipWith is a *Vector* object containing the kinship coefficients of the *Animal* object with all the other currently living animals, including itself. Because *Vector*s only store objects, each kinship coefficient in the kinshipWith *Vector* object is stored as an instance of Java's *Double* class. You can think of kinshipWith as being a single row in the conventional additive kinship matrix. We instantiate one *Animal* object and one kinshipWith *Vector* object for each living animal. Next we instantiate a *Vector* object called livingAnimals that contains all the *Animal* objects, denoted animal_0, animal_1, etc., representing the current population. It is worth noting that because the livingAnimals *Vector* contains the current population and each individual animal in the population stores its parents' identification numbers, the exact pedigree of the current population can be known at any time during the simulation.

To begin a simulation we create a founding population of *N* animals. The initial animals will be given identifications numbered sequentially based on age. All new animals will have an index number based on the order of their birth. The elements of livingAnimals, will naturally be arranged in ascending order, from oldest to youngest. The *Animal* object, animal_0, remains permanently in the 0th position and represents an animal outside the population. During the simulation, livingAnimals is continually updated and only contains the current living animals, for example, something like {animal_0, animal_22, animal_31, animal_44, animal_46, animal_102}, which shows that animal_22 is the oldest living animal in the population. The length of this *Vector* is the size of the compressed additive kinship matrix. The kinship between, say, animals 31 and 44 in an additive matrix would be stored in either the 31st row and the 44th column or the 44th row and 31st column. In our "compressed kinship matrix," since animal_31 is the second element of livingAnimals and animal_44 is the third element, the kinship coefficient between animal_31 and animal_44 will be stored in the 3rd position of animal_31's kinshipWith and in the 2nd position of animal_44's kinshipWith objects. The one-to-one mapping necessary to access kinship coefficients is accomplished via the *Vector* methods that are part of the Java language, the indexOf( ) method and the elementAt( ) method. In OOPs a "." operator between an object and a method means that that method is applied to the particular object. For example, livingAnimals.indexOf (animal_31) returns 2 and livingAnimals. elementAt(2) returns animal_31. Thus the kinship between animal_31 and animal_44 would be accessed by

```
animal_31.kinshipWith.elementAt
(livingAnimals.indexOf(animal_44)).
```

During simulation, animals will be born and will die. The maintenance of all the kinship relationships that occurs with demographic events, the adding of kinship coefficients associated with newborns and the deleting of the coefficients associated with deceased

animals, is handled by the two *Vector* methods already discussed and four additional methods, addElement( ), insertElementAt( ), removeElement( ), and removeElementAt( ). Death is handled straightforwardly. For example, assume animal_31 dies in the livingAnimals *Vector* given above, one could do livingAnimals.removeElementAt(2) or livingAnimals.removeElement(animal_31), and with either of these livingAnimals would become {animal_0, animal_22, animal_44, animal_46, animal_102} and it would be one element smaller in size. The Java language handles all lower-level computational tasks, such as adjustment of indices and memory allocation, associated with such manipulations. Next, all the kinship coefficients in the second position of all the still living animals' kinshipWith objects need to be removed. In pseudo-code, this is carried out in the loop:

```
for each animal object in the
livingAnimals Vector

  use removeElement At( ) to
  remove the element in the
  animal's kinshipWith Vector
  at position 2

next animal
```

Finally, animal_31 has been utilizing computer memory to store all of its instance variables. These resources need to be freed. The programmer executes the statement

```
animal_31 = null.
```

Java has what is termed run-time garbage collection. That is, there is a process (a thread) running in the background looking for disposable objects. Disposable objects are objects that have been set equal to **null** or are no longer referenced by any other objects. It automatically returns the memory associated with such objects.

Handling births is similarly straightforward. If a new animal is born, say, an animal with id = 103, a new object is created from the *Animal* class. First, this new object is added to the end of the livingAnimals *Vector* with the statement

```
livingAnimals.addElement
(animal_103).
```

If the size of livingAnimals needs to be increased, Java handles the task of allocating additional memory. Next, its kinship with all other animals in livingAnimals, including itself, is calculated using equations (1) and (2). These calculations do not require recursive operations, but are directly obtained via lookup once the position of animal_103's mother and father are obtained using the indexOf( ) method as discussed above. The kinship between an older animal_i and animal_103 then becomes the average of the kinship between animal_i and ϕ(animal_103) and the kinship between animal_i and ρ(animal_103). The values are then entered subsequently as elements, in the appropriate position, in the kinshipWith objects of all the animals using the insertElementAt( ) method. The pseudo-code looks like this:

```
use indexOf( ) to find the
index in livingAnimals Vector
of animal_103

for each animal object in the
livingAnimals Vector

  use indexOf( ) to find the in-
  dex of the animal object

  use indexOf( ) to find the in-
  dex of animal_103's mother
  and father

  calculate the kinship coef-
  ficient between the animal
  object and animal_103, using
  equations (1) and (2)

  use insertElementAt( ) to in-
  sert the calculated kinship
  coefficient into ani-
  mal_103's
  kinshipWith Vector at index
  position of the animal object

  use insertElementAt( ) to in-
  sert the calculated kinship
  coefficient into the animal
  object's kinshipWith Vector
  at index position of the ani-
  mal_103

next animal.
```

There are some concerns regarding the speed of Java. In 1995, when Sun Microsystems, Inc., first introduced Java, it was promoted as the "write once run anyway" programming language. This versatility is obtained by partially compiling a Java program into bytecode that is not computer or operating system specific. The byte code then is "interpreted" by Java's Virtual Machine (JVM) into executable instructions for a specific platform's CPU. Compilers for programming languages like C and C++, on the other hand, translate programming code into native machine language that is platform specific. Interpretation is inherently slow and is why Java initially obtained the reputation of being a slow language. However, more recent versions of the JVM come with a "just-in-time" (JIT) compiler. A JIT translates and stores entire class files, eliminating the need to repeatedly "interpret" the same byte code to machine language. Tests show that with the use of Java's JIT compilers, Java can be as fast as C++ (Mangione 1998).

Ignoring the relatively minor issue of execution speed, our intent is to provide object-oriented programmers with a technique that minimizes programming effort by utilizing vendor-defined classes and methods, as well as to provide individual-based modelers with a technique that is congruent with the individual-based modeling perspective. For individual-based modeling, it seems appropriate that each individual should itself store and maintain a record of its kinship relationship to all other individuals in a population.

This completes the discussion of the compressed kinship matrix algorithm. We used Java's dynamic array class, *Vector*, and its associated methods to solve the indexing and matrix manipulation issues necessary for its implementation. We have performed the kinship coefficient calculations for a panmictic population of 100 animals for 100 nonoverlapping generations and determined the mean kinship and its variance for the population in each generation. A Java applet demonstrating our procedures for a set number of individuals in a comparable simulation is available at our Web site: http://www.consbio.com/kinshipAlgorithm. Detailed documentation of the actual Java source code for our algorithm, together with a simplified but similar simulation, is likewise provided.

From the University of Utah, Department of Geography, Salt Lake City, UT 84112, and Montana State University, Department of Ecology and Evolution, Bozeman, MT 59717. This work was supported by a University of Utah Graduate Research Fellowship and NSF Grant 58500721 to Vickie Backus. We would like to thank Eric Ward, Chris Ray, and two anonymous reviewers for helpful comments on the manuscript. Address all correspondence to Vickie Backus at the address above, or e-mail: vickie.backus@geog.utah.edu.

References

Ballou JD, 1983. Calculating inbreeding coefficients from pedigrees. In: Genetics and conservation: a reference for managing wild animal and plant populations (Schonewald-Cox CM, Chambers SM, MacBryde B, and Thomas WL, eds). Menlo Park, CA: Benjamin/Cummings; 509–520.

Boyce AJ, 1983. Computation of inbreeding and kinship coefficients on extended pedigrees. J Hered 74:400–404.

Lacy RC, 1993a. GENES: a computer program for the analysis of pedigrees and genetic management. Brookfield, IL: Chicago Zoological Society.

Lacy RC, 1993b. VORTEX: a computer simulation model for population viability analysis. Wildlife Res 20:45–65.

Lacy RC, Ballou JD, Princée F, Starfield A, and Thompson EA, 1995. Pedigree analysis for population management. In: Population management for survival and recovery (Ballou JD, Gilpin M, and Foose TJ, eds). New York: Columbia University Press; 57–75.

Mangione C, 1998. Performance tests show Java as fast as C++. Accessed at JavaWorld on August 28, 2002, at www.javaworld.com/jw-02-1998/jw-02-jperf.html.

Sun Microsystems, Inc. Java™ 2 SDK, standard edition, version 1.2.2. Accessed on August 28, 2002, at http://java.sun.com/products.

Thompson EA, 1976. Inference of genealogical structure. Soc Sci Info 15:477–526.

Received May 11, 2002
Accepted October 1, 2002

Corresponding Editor: Stephen J. O'Brien