Algorithms for Comparing Pedigree Graphs

Bonnie Kirkpatrick

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-89
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-89.html

May 31, 2010

Acknowledgement

Many thanks to my supportive advisor Prof. Richard Karp.

Algorithms for Comparing Pedigree Graphs

Bonnie Kirkpatrick

Family relationships in the form of pedigree graphs are currently collected from genealogical records in an expensive process of determining which pairs of people are parent and child. With the end goal of reconstructing pedigrees, this paper considers how to compare two pedigree graphs for accuracy.

This paper reminds the reader of an alternative formulation of pedigree relationships, published earlier [1], that describes a pedigree as a list of descendant individuals, rather than parent-child relationships. This formulation is useful for comparing two similar pedigree graphs via a randomized algorithm that estimates the minimum number of edge changes that are necessary to change one pedigree into the other. This randomized algorithm runs in polynomial time.

1 Introduction

For diploid individuals, the traditional formulation of a pedigree is a directed graph where individuals I are nodes and where every edge represent parentage, i.e. there is a directed edge $i \rightarrow j$ if and only if i is the parent of j (see Figure 1). Specifically, for each type of chromosome, this edge represents the transmission, from parent i to offspring j, of a single (possibly recombinant) copy of that chromosome. In this formulation it is clear that the accuracy of the edges is of paramount importance and that the presence or absence of a single edge will determine whether many pairs of individuals are related to each other.

Whereas a pedigree graph represents all possible inheritance paths, identity by descent (IBD) can be thought of as the instantiation of particular inheritance paths for a single locus. Formally, an *inheritance path*, s^i, is a binary vector with m_i bits, one for each meiosis, with each bit indicating which grand-parental allele, paternal or maternal, was copied for that meiosis. To formally define identity by descent, we need also introduce a graphical representation of the inheritance path. An *inheritance graph*, R_s, equivalent to inheritance path s, has two nodes per individual (one for each allele) and edges from the child allele to the parent allele that it was copied from. This graph is a set of digraphs, and each connected component indicates alleles that are *identical by descent (IBD)*. R_x is a forest of trees. Assume that there is no mutation in this model, because the mutation process has negligible probability of effecting a single site in a single generation. Therefore alleles that are IBD have the same nucleotide state.

Assume that there is a set of labeled individuals in the pedigree; roughly speaking these individuals would be distinguished, typically because we have data for them. Let $X \subset I$ be the set of labeled individuals.

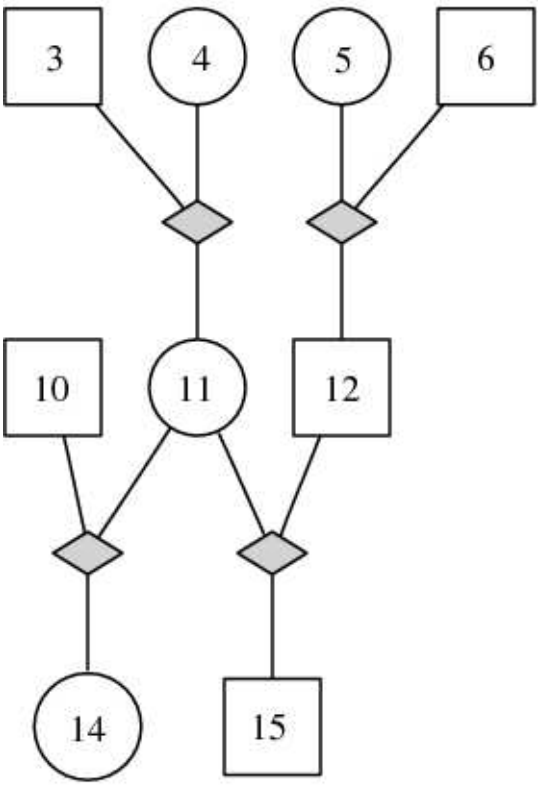

Figure 1: **An Example Pedigree.** Each node is an individual, with boxes representing males and circles representing females. The diamond nodes represent marriages, and the two individuals adjacent to and above the marriage node are the parents of the adjacent individual(s) below the marriage node. The marriage node is simply a slightly more compact way of representing edges from parents to children. Time proceeds in the downward direction, implicitly directing this graph. It is standard to discard the arrows on the edges and to use the top-to-bottom ordering of nodes in the graph to convey the directionality. Individuals without parents are called *founders*, and they are assumed to be unrelated.

2 Alternative Formulation

2.1 Descent-Splits Equivalence

[This section is a revised version of a similar section in [1].] An alternative formulation of a pedigree would allow the hypothesis that a set of individuals is descended from a common ancestor (called a **descent split**), without specifying the number of generations between each of the individuals and their common ancestor(s). The presence or absence of a single descent hypothesis may only change the closeness of the relationship between a pair of individuals (perhaps from cousins to 2nd-cousins), rather than removing the relationship entirely. This is in contrast to the traditional formulation of a pedigree as a collection of parent-offspring edges, where a missing edge entirely changes the nature of many relationships.

Definition. Let I be the set of individuals in a pedigree, and let X be the set of labeled individuals in a pedigree. The **descent split** (or **d-split**) of an individual $i \in I$ is defined as a subset of X:

$$D_i(X) \quad = \quad \{j \in X | \ \ j \text{ is descended from } i\}$$

where an individual is *not* a descendant of itself. For a particular set of interest, X, refer to the set of d-splits as $\mathcal{D}_X = \{D_i(X) \mid i \in I\}$.

Each d-split specifies some relationship between all the individuals in D_i. For the example given in Figure 1, when all individuals are labeled the list of d-splits, $\mathcal{D}_I$, are: $D_{14} = \emptyset$, $D_{15} = \emptyset$, $D_{10} = \{14\}$, $D_{12} = \{15\}$, $D_{11} = \{14, 15\}$, $D_3 = \{11, 14, 15\}$, $D_4 = \{11, 14, 15\}$, $D_5 = \{12, 15\}$, and $D_6 = \{12, 15\}$. Similarly, if we restricted our attention to $X = \{14, 15\}$, then $\mathcal{D}_X$ would contain: $\emptyset$, $\{14\}$, $\{15\}$, and $\{14, 15\}$.

The term "descent split" is deliberately chosen to evoke the image of a split in a perfect phylogeny and to pay homage to phylogenetic reconstruction methods [2] which are a source of inspiration for this work. Just as a set of splits determines a class of perfect phylogeny trees that are compatible with the splits, a set of descent splits specifies a class of pedigree graphs that are compatible with the splits. We will formalize this idea with several lemmas.

Lemma 2.1. *Let $\mathcal{D}_I = (D_i(I) \mid i \in I)$ be the d-splits defined by a pedigree P. This set can be used to reconstruct a unique pedigree which is identical to pedigree P.*

Lemma 2.2. *For pedigree P, let $\mathcal{D}_X = \{D_i(X) \mid i \in I\}$ be the set of d-splits that partition the genotyped individuals $X \subset I$. This set of d-splits specifies a class of pedigrees compatible with the splits. Pedigree P is one of the pedigrees compatible with the d-splits.*

First consider the d-splits in $\mathcal{D}_I$. Any trivial d-split, $D_i \in \mathcal{D}_I$ with $D_i = \emptyset$, clearly represents an individual that is childless. Therefore these d-splits represent individuals in the most recent generation of the pedigree. Now, find some ancestor i_1 and examine any directed path descending from that person, for example, $i_1 \rightarrow i_2 \rightarrow ... \rightarrow i_{k-1} \rightarrow i_k$, where the arrow indicates a directed parent-offspring relationship. We see that the d-splits along that descent path are ordered $D_{i_1} \supset D_{i_2} \supset ... \supset D_{i_k}$. Indeed, the cardinality of the d-split sets D_{i_j} strictly decrease as we consider

individuals lower in the path. These two ideas result in a simple algorithm for reconstructing the pedigree.

Reconstruction Algorithm.
$Heap := (D_{i_0}, ..., D_{i_k})$ where $|D_{i_0}| \leq |D_{i_1}| \leq ... \leq |D_{i_k}|$
Create pedigree P with nodes $\{i_0, i_1, ..., i_k\}$.
While $Heap \neq \emptyset$
$D_{i_j} := pop(Heap)$
Look for the smallest D_{i_f} and D_{i_m}, such that $D_{i_j} \subseteq D_{i_f}$ and $D_{i_j} \subseteq D_{i_m}$
If D_{i_f} and D_{i_m} are found, add to P the edges $i_m \to i_j$ and $i_f \to i_j$
Else i_j is a founder and has no parents.
End While

Example. If we take the d-splits $\mathcal{D}_I$ from the example in Figure 1, we can apply the algorithm to reconstruct the pedigree. Figure 2 shows the d-splits using a Venn diagram. The upper picture shows the reconstruction generated by the algorithm after the first three iterations. The bottom picture shows the full reconstruction in which the last two iterations of the algorithm construct the second generation of the pedigree.

Proof. of Lemma 2.1
Since we have a d-split for every individual, the algorithm will either assign founder status or parents to every individual. Now, if we look at a single step in the algorithm, each individual will be assigned the correct parents, due to the strictly increasing cardinality of d-splits as we consider d-splits for individuals in older generations. □

Interestingly, we can use the same algorithm when we consider d-splits on a subset of the individuals. As long as we have a separate d-split for each person in the pedigree, we will know the number of generations in each lineage. The main difference is that each lineage has *non-decreasing* cardinality of d-splits as we more backwards in time. The missing information, now, is in not knowing which d-split was generated by the parent versus a more distant ancestor. For the example we gave above, if $X = \{14, 15\}$, then $D_{11}(X)$ and $D_3(X)$ are indistinguishable.

Proof. of Lemma 2.2
Again, since we have a d-split for every individual, the algorithm will either assign founder status or parents to every individual. Now, if we look at a single step in the algorithm, each individual will be assigned some parents, due to the non-decreasing cardinality of d-splits as we consider d-splits for individuals in older generations. However, the reconstruction will be different for re-orderings of the d-splits. This means that we cannot resolve the correct labels for individuals $I \setminus X$ in the interior of the pedigree. □

2.2 Edge Edit Distance on Pedigree Graphs

To determine how good a predicted pedigree is, we want to compare it to a correct pedigree graph and use this to evaluate reconstruction methods. Informally, given two arbitrary pedigree graphs,

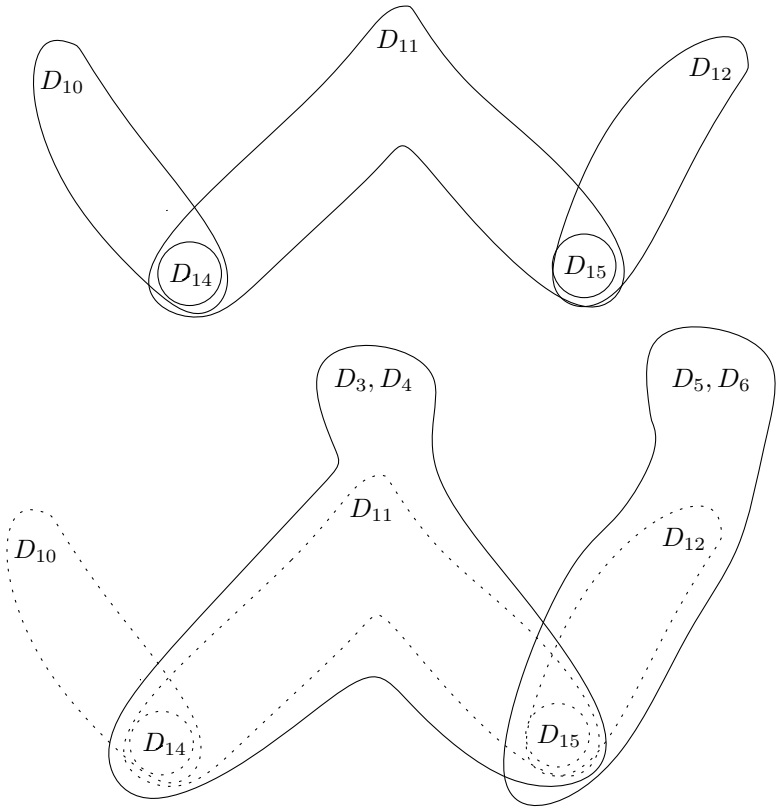

Figure 2: **Reconstructing a Pedigree from the Full D-Splits.** Given the d-splits in $\mathcal{D}_I$ for the set I of all the individuals in the pedigree in Fig. 1, we can use the reconstruction algorithm to recover the pedigree. These are the Venn diagrams of the reconstruction at two different steps in the algorithm. The upper panel shows the first three steps of the algorithm, while the bottom panel shows the complete reconstruction. Each d-split is drawn as a set containing the related individuals. Each set in the diagram is labeled with the name of its d-split, and the names of the d-splits are arbitrary as long as they are distinct.

$P = (I(P), E(P))$ and $Q = (I(Q), E(Q))$, find the minimum number of parent-child edge changes required to convert Q into P and call this the *edge edit distance* between P and Q.

Let M be a bipartite *matching* on the individuals from P and the individuals from Q where gender is preserved. For an individual i, let $s(i) \in \{m, f\}$ indicate male and female gender, respectively. The matching is a list of match edges, $M \subset I(P) \times I(Q)$ such that every $i \in I(P) \cup I(Q)$ is incident on one match edge and such that for every $(i, j) \in M$, gender is preserved, $s(i) = s(j)$. Define $f_M(i)$ as a function that takes $i \in I(P) \cup I(Q)$ and returns the adjacent vertex in M.

For a given matching, M, we define the *match distance* as

$$d(M) = d_{P,Q}(M) + d_{Q,P}(M),$$

where $d_{G,G'}(M) = |\{(p_i, i) \in E(G) | (f_M(p_i), f_M(i)) \notin E(G')\}|$. These are the parent-child edges in pedigree G that are not mapped via M to a parent-child edge in G'.

Now, we can define a distance between pedigrees. Let the *edit distance* between pedigrees P and Q be defined as the minimum matching distance:

$$D_{P,Q} = \min_M d(M).$$

Assume that we have a set of labeled individuals X, as before, and that the set of labeled individuals is the same in both pedigrees P and Q. Recall that this is a set of distinguishable individuals; for instance we may have data for them. Now, we can define a *labeled matching* M_X that preserves the labeled individuals. M_X is a bipartite matching, as above, with the additional condition that for $i \in X$, $(i, i) \in M$. This forces the labeled individuals to be matched to each other. Thus, a matching on the people induces a matching on the edges.

Two-Generation Polynomial-Time Algorithm. For two-generation, monogamous pedigrees with the most recent generation being fully labeled, there is a polynomial-time algorithm. We can construct a maximum bipartite matching instance whose solution gives us the best matching.

Let $G = (V(G), E(G))$ be the bipartite graph on which we will find a matching, and let the edge weights be $w(u, v)$ for $(u, v) \in E(G)$. For every $(i, j) \in I(P) \times I(Q)$, we will create two nodes $i, j \in V(G)$. Edges are created as follows:

1. For all $i \in X$, create edge $(i, i) \in E(G)$.
2. For all $(i, j) \in I(P) \times I(Q)$ where $i \notin X$, $j \notin X$ and $s(i) = s(j) = m$, if i and j each have a child $c \in X$, create $(i, j) \in E(G)$ with weight $w(i, j)$ equal to the number of children having the same label.

Notice that gender becomes an issue when there are half-siblings (i.e. non-monogamy). In that situation, we need a matching for each gender, and we need to force consistency between gendered matchings of monogamous parents.

When we compute the maximum matching on this graph G, we will obtain the matching of the parents of the labeled individuals that minimizes the number of mis-matched child edges. This holds even for P and Q which are each collections of families.

Exponential-Time Matching Algorithm. Let a *regular* pedigree be one where every individual is monogamous and only individuals of the same generation mate with each other. Consider the case of two regular pedigrees both having the same set of labeled individuals, all of whom are in the most recent generation. One might think that we could recursively perform the two-generation matching algorithm above. However, it is easy to design pedigrees P and Q with three generations, each having the same number of individuals per generation, for which the minimum matching for the whole pedigree does not contain the minimum matching for the second-generation. An obvious exact solution to that particular problem is to branch on the possible matchings for each generation before continuing recursively to consider older generations. For P an Q with the same number of individuals in every generation, the recursively generated matching with the least total mis-matches would seem to yield the correct edit distance.

Random Matching Algorithm using D-Splits. This paper will focus focus on polynomial-time algorithms for practical purposes, rather than complexity theory. The randomized matching algorithm for regular pedigrees is exceptionally simple. At each generation, among individuals of the same gender, choose a match with probability proportional to the number of same-labeled individuals in the descent sets of the two pedigrees. This algorithm is polynomial, because at each generation it creates a $n \times n$ matrix of individual match probabilities for each gender (where there are $2n$ individuals per generation). The matches are then drawn iteratively from these probability matrices (without replacement of previously matched individuals).

3 Code

The method has been implemented in C++ for random pedigrees. Each random pedigree graph is drawn from a Wright-Fisher simulation where each generation has a fixed number of individuals ($2n$), and each individual uniformly chooses their parents from n monogomous parent-pairs.

In this particular implementation, one of the pedigrees, $sim_pedigree$, has no gender, so for that pedigree, the gender is assigned uniformly at random. Before applying the above matching algorithm.

```
enum Gender {UNKNOWN, MALE, FEMALE};


class Indiv
{
public:
```

```
  Indiv(int _indiv_index,int _generation){ indiv_index = _indiv_index;
  generation = _generation; gender = MALE;};

  Indiv(const Indiv& j);
  const Indiv& operator= (const Indiv& j);

  void PushDescendant(int d){descendants[d]=1;};
  void PushDescentList(const Indiv& j);

  void InsertChildren(const Indiv& j);
  void PushChild(int c){children.push_back(c);};

  void PushParent(int p){parents.push_back(p);};
  void PushParentList(const Indiv& j);

  void GetDescentList(vector<int>& d_split);
  bool HasDescendant(int d);

  bool HasChild(int c);
  void GetChildList(vector<int>& child_list);

  void GetParents(int& f, int& m); // only two parents

  void SetGender(Gender g){gender = g;};

  void ResetColor(){color = 0;};
  void Print();

public:
  int indiv_index;
  int generation;
  Gender gender;

  map<int,int> descendants;
  vector<int> children;
  vector<int> parents;

  unsigned int color;

  vector<set<int> > distinct_alleles;
};


Indiv::Indiv(const Indiv& j)
{
  color = 0;
```

```
  indiv_index = j.indiv_index;
  gender = j.gender;
  generation = j.generation;
  PushDescentList(j);
  InsertChildren(j);
  PushParentList(j);
}
const Indiv& Indiv::operator= (const Indiv& j)
{
  color = 0;
  indiv_index = j.indiv_index;
  gender = j.gender;
  generation = j.generation;
  PushDescentList(j);
  InsertChildren(j);
  PushParentList(j);
  return *this;
}
void Indiv::PushDescentList(const Indiv& j){
  map<int,int>::const_iterator mi = j.descendants.begin();
  while(mi != j.descendants.end()){
    descendants[mi->first] = 1;
    mi++;
  }
}
void Indiv::InsertChildren(const Indiv& j)
{
  for (unsigned int c = 0;  c < j.children.size();  c++){
    children.push_back(j.children[c]);
  }
}
void Indiv::PushParentList(const Indiv& j)
{
  if (j.parents.size() > 2)
    printf("\nWARNING: too many parents for person %i\n", indiv_index);
  for (unsigned int p = 0;  p < j.parents.size();  p++)
    parents.push_back(j.parents[p]);
}

void Indiv::GetDescentList(vector<int>& d_split){
  map<int,int>::const_iterator mi = descendants.begin();
  while(mi != descendants.end()){
    d_split.push_back(mi->first);
    mi++;
  }
}
```

```
bool Indiv::HasDescendant(int d){
  map<int,int>::const_iterator mi = descendants.find(d);
  if (mi != descendants.end()){
    return true;
  }
  return false;
}
bool Indiv::HasChild(int c){
  vector<int>::const_iterator vi = find(children.begin(), children.end(), c);
  if (vi != children.end()){
    return true;
  }
  return false;
}
void Indiv::GetChildList(vector<int>& child_list)
{
  vector<int>::const_iterator vi = children.begin();
  while(vi != children.end()){
    child_list.push_back(*vi);
    vi++;
  }
}

void Indiv::GetParents(int& p1, int& p2)
{
  if (parents.size() > 2)
    printf("\nWARNING: too many parents for person %i\n", indiv_index);

  p1 = parents[0];
  if (parents.size() == 2)
    p2 = parents[1];
}
void Indiv::Print()
{
  printf("  Person %i, generation %i, color %i, gender %i\n",
indiv_index, generation, color, gender);

  printf("      Parents:     ");
  for (unsigned int i = 0;  i < parents.size(); i++)
    printf("%i, ", parents[i]);
  printf("\n      Children:    ");
  for (unsigned int i = 0;  i < children.size(); i++)
    printf("%i, ", children[i]);
  printf("\n      Descendants: ");
  map<int,int>::const_iterator mi = descendants.begin();
```

```
  while(mi != descendants.end()){
    printf("%i,", mi->first);
    mi++;
  }
  printf("\n");
}
```

```
// integer indexes numbered [1...n]
//
void setParents(vector<Indiv>* pedigree, int i, int m, int f)
{
    (*pedigree)[i].PushParent(m);
    (*pedigree)[i].PushParent(f);

    (*pedigree)[m].PushChild(i);
    (*pedigree)[m].PushDescentList((*pedigree)[i]);

    (*pedigree)[f].PushChild(i);
    (*pedigree)[f].PushDescentList((*pedigree)[i]);
}

// Wontonly delete the children and descent lists of parents m & f
// Delete the parents of i
void unsetParents(vector<Indiv>* pedigree, unsigned int i,
unsigned int m, unsigned int f)
{
    (*pedigree)[i].parents.clear();
    (*pedigree)[m].children.clear();
    (*pedigree)[m].descendants.clear();
    (*pedigree)[f].children.clear();
    (*pedigree)[f].descendants.clear();
}

// make a new person
unsigned int makeUntypedPerson(vector<Indiv>* pedigree, int generation)
{
  unsigned int index = pedigree->size();
  Indiv new_person(index,generation);
  pedigree->push_back(new_person);
  return index;
}
```

```
unsigned int makeTypedPerson(vector<Indiv>* pedigree, int generation)
{
  unsigned int index = pedigree->size();
  Indiv new_person(index,generation);
  new_person.PushDescendant(index);
  pedigree->push_back(new_person);
  return index;
}
```

```
unsigned int randomMatch(vector<Indiv>& sim_pedigree,
 vector<Indiv>& pedigree,
 vector<unsigned int>& generations, // for ped.
 int num_extant)
{

  map<unsigned int, unsigned int> matching; // pedigree indiv, sim_ped indiv
  for (int i = 0;  i < num_extant; i++)
    {
      matching[i] = i;
    }


  // In the estimated pedigree, assign gender randomly.
  // Init gender to UNKNOWN
  for (unsigned int i = 0;  i < pedigree.size(); i++)
    {
      pedigree[i].gender = UNKNOWN;
    }


  // Each generation is randomly matched with probability proportional
  // to the number of same-labeled individuals in the descent set.


  // matching prob. for prev generation
  map<unsigned int, unsigned int> prev_idx_i;
  map<unsigned int, unsigned int> prev_idx_j;
  vector<vector<double> > first_gen_match_prob(num_extant,
vector<double>(num_extant, 0.0));
  // init to first generation (where match forced to same label)
  for (int i = 0;  i < num_extant; i++)
    {
```

```
      first_gen_match_prob[i][i] = 1;
      prev_idx_i[i] = i;
      prev_idx_j[i] = i;
    }
  vector<vector<double> >* prev_gen_match_prob;
  prev_gen_match_prob = &first_gen_match_prob;



  // For each generation, compute the matching prob
  unsigned int sim_first_in_gen = generations[1]; // sim ped
  unsigned int est_first_in_gen = generations[1]; // estimated ped
  // generations[g] is the person to not include in this gen
  for (unsigned int g = 2;  g < generations.size();  g++)
    {
      if (est_first_in_gen != generations[g-1] ||
  pedigree[est_first_in_gen].generation
!= sim_pedigree[sim_first_in_gen].generation){
printf("ERROR: mismatch in generation for estimated pedigree\n");
exit(-34);
      }
      unsigned int sim_next_gen = 0;


      map<unsigned int, unsigned int> idx_i;
      map<unsigned int, unsigned int> idx_j;

      map<unsigned int, bool> used_j;  // key j, value 0,1 for used or not


      //vector<vector<double> >* match_prob
= new vector<vector<double> >(0, vector<double>(0, 0.0));
      vector<vector<double> > match_prob;
      for (unsigned int i = est_first_in_gen; i < generations[g]; i++)
{
  vector<double> counts = vector<double>(0,0.0);
  //idx_i[i] = match_prob->size();
  idx_i[i] = match_prob.size();

  vector<int> D_i;
  pedigree[i].GetDescentList(D_i);

  unsigned int sum_male = 0;
  unsigned int sum_female = 0;

  unsigned int j = sim_first_in_gen;
```

```
  while(j < sim_pedigree.size() &&
sim_pedigree[j].generation == (signed) g)
    {
      int common_descendant_count = 0;
      for (unsigned int d = 0;  d < D_i.size();  d++)
{
  if (sim_pedigree[j].HasDescendant(D_i[d]))
    common_descendant_count++;
}
      if (used_j[j] != 1)
{
  if (sim_pedigree[j].gender == MALE)
    sum_male += common_descendant_count;
  else
    sum_female += common_descendant_count;
}

      if (i == est_first_in_gen)
idx_j[j] = counts.size();

      counts.push_back(common_descendant_count);

      j++;
    } // end for each sim_pedigree person in gen
  sim_next_gen = j;
  //(*match_prob).push_back(counts);
  match_prob.push_back(counts);



  // Normalize
  j = sim_first_in_gen;
  while(j < sim_pedigree.size() &&
sim_pedigree[j].generation == (signed) g)
    {
      if (sum_male+sum_female > 0)
{
  if (sim_pedigree[j].gender == MALE)
    match_prob[idx_i[i]][idx_j[j]] /= (sum_male);
  else
    match_prob[idx_i[i]][idx_j[j]] /= (sum_female);
}

      j++;
    } // end for each sim_pedigree person in gen
```

```
  // draw the gender for this person and their co-parent
  double u = drand48();
  if (pedigree[i].gender == UNKNOWN)
    {
      Gender other_gender = UNKNOWN;
      if (u < 0.5) {
pedigree[i].gender = MALE;
other_gender = FEMALE;
      } else {
pedigree[i].gender = FEMALE;
other_gender = MALE;
      }
      // set other parent
      unsigned int child = pedigree[i].children[0];
      unsigned int other_parent = pedigree[child].parents[0];
      if (other_parent == i)
other_parent = pedigree[child].parents[1];
      pedigree[other_parent].gender = other_gender;

      u = drand48();
    }

  // draw the matching for this person
  j = sim_first_in_gen;
  while(j < sim_pedigree.size() &&
sim_pedigree[j].generation == (signed) g)
    {
      if (sim_pedigree[j].gender == pedigree[i].gender &&
  used_j[j] != 1)
{
  //u -= (*match_prob)[idx_i[i]][idx_j[j]];
  u -= match_prob[idx_i[i]][idx_j[j]];
  if (u <= 0.0)
    {
      matching[i] = j;
      used_j[j] = 1;
      break;
    }
}

      j++;
    } // end for each sim_pedigree person in gen

  //printf("  match ped %i to sim %i\n", i, matching[i]);
```

```
} // for each i in pedigree


      sim_first_in_gen = sim_next_gen; // sim ped
      est_first_in_gen = generations[g]; // estimated ped


    } // end for each generation
```

References

[1]

[2] C. Semple and M. Steel. *Phylogenetics*. Oxford University Press, 2003.