The American Journal of Human Genetics, Volume 88

Supplemental Data

A Fast, Powerful Method for Detecting Identity by Descent

Brian L. Browning and Sharon R. Browning

Pseudocode for fastIBD algorithm

Notation and Conventions

We assume the phased haplotype data has N samples (S[1:N]), L markers (M[1:L]), K sampled haplotype pairs per individual, and 2NK sampled haplotypes (H[1: 2NK]). The sample corresponding to a haplotype $h \in$ H is denoted sample(h). If a and b are two elements of S, M, or H, then $a < b$ means that the index of a is less than the index of b. We assume markers are indexed in chromosomal order and that samples and haplotypes have consistent ordering so that for $h, g \in$ H, $h \leq g$ if and only if sample(h) $\leq$ sample(g). All iterations over collections are in increasing order. For haplotypes $h, g \in$ H, and markers $a, b \in$ M with $a < b$, $h(a, b) == g(a, b)$ means that h and g have identical model states when restricted to the marker window starting at marker a (inclusive) and ending at marker b (exclusive).

B[1:R] is a subsequence of distinct markers with B[1]==M[1], B[R]==M[L], and having approximately equal genetic distance between all adjacent markers in the subsequence.

Map is an associative map whose keys are ordered pairs of samples (s, t) satisfying $s < t$, and whose values are sorted Lists of shared haplotype tracts for the sample pair, sorted in order of starting marker index. Shared haplotype tracts are defined in the Methods section of the main text.

Threshold is the threshold for printing shared haplotype tracts. Tracts with score less than the threshold are printed.

For simplicity, the pseudocode presents the basic fastIBD algorithm. There are two additional optimizations that BEAGLE employs to decrease computation time:

1. A hash map is used to eliminate the iteration over all pairs of haplotypes in the ADD-TRACTS algorithm. This optimization is nicely described in Gusev et al. (Genome Res 2009;19(2):318-26).

2. Iteration over all pairs of samples in EXTEND-OR-REMOVE-TRACTS is replaced with iteration over the keys of Map which are mapped to a non-empty sorted tract list.

fastIBD Pseudocode

Algorithm: fastIBD

```
ADD-TRACTS(Map, B[1], B[2])
FOR (j=2, 3, …, R-1) DO
  ADD-TRACTS(Map, B[j], B[j+1])
  EXTEND-OR-REMOVE-TRACTS(Map, B[j], Threshold)
EXTEND-OR-REMOVE-TRACTS(Map, B[R], Threshold)
```

Algorithm: ADD-TRACTS(Map, Start, End)

```
FOR h ∈ H DO
  For g ∈ H WITH sample(g) > sample(h) DO
    IF (h(Start, End) == g(Start, End)) THEN
      a = min {m ∈ M : h(m, Start)== g(m, Start)}
      b = max {m ∈ M : Score(h, g, a, m) ≤ 1} //Score() is defined in Methods
      c = arg max_{a≤m≤b} {Score(h, g, a, m)}
      x = Score(h, g, a, c)
      Tract = NEW Tract(H1=h, H2=g, start=a, end=c, score=x)
      SortedTractList = Map.get(sample(h), sample(g))
      SortedTractList.add(Tract)
```

Algorithm: EXTEND-OR-REMOVE-TRACTS(Map, Boundary, Threshold)

```
FOR s ∈ S DO
  FOR t ∈ S WITH t > s DO
    SortedTractList = Map.get(s, t)
    EXTEND-TO-BOUNDARY(SortedTractList, Boundary)
    MERGE-TO-BOUNDARY(SortedTractList, Boundary)
    FOR Tract ∈ SortedTractList WITH Tract.End < Boundary DO
      SortedTractList.remove(Tract)  // Tract could not be extended
      IF (Tract.score < threshold) THEN
        PRINT Tract
```

Algorithm: EXTEND-TO-BOUNDARY(SortedTractList, Boundary)

```
IF (SortedTractList.size()==0) THEN
  RETURN
MERGE-COVERED-TRACTS(SortedTractList)
// Next, recover Map Key (ordered sample pair) mapped to SortedTractList
Sample1 = sample(SortedTractList[1].H1)
Sample2 = sample(SortedTractList[1].H2)
Gap = FIRST-GAP(SortedTractList)
WHILE (Gap < Boundary) DO
  PreviousGap = Gap
  Extension = NULL
  MaxExtensionScore = -1.0
  FOR h ∈ H WITH sample(h)==Sample1 DO
    FOR g ∈ H WITH sample(g)==Sample2 DO
      b = max {m ∈ M : m ≥ Gap AND h(Gap, m)==g(Gap, m)}
      x = Score(h, g, Gap, b)   // Score() is defined in Methods
      IF (b > Gap AND x > MaxExtensionScore) THEN
        MaxExtensionScore = x
        Extension = NEW Tract(H1=h, H2=g, start=Gap, end=b, score=x)
  IF (EXTENSION ≠ NULL) THEN
    SortedTractList.add(Extension)
  Gap = FIRST-GAP(SortedTractList)
  If (Gap == PreviousGap) THEN
    Gap = Boundary
```

Algorithm: FIRST-GAP(SortedTractList)

```
IF (SortedTractList.size()==0) THEN
  Return M[L]
End = SortedTractList[1].end
For Tract ∈ SortedTractList DO
  IF (Tract.Start ≤ End) THEN
    IF (Tract.End > End) THEN
      End = Tract.End
RETURN End
```

Algorithm: MERGE-TO-BOUNDARY(SortedTractList, Boundary)

```
MERGE-COVERED-TRACTS(SortedTractList)
TractA = NULL
IF (SortedTractList.size() ≥ 2) THEN
  TractA = SortedTractList[1]
WHILE (TractA ≠ NULL AND TractA.end < Boundary) DO
  startingSize = SortedTractList.size()
  TractB = NULL
  FOR T ∈ SortedTractList WITH (T ≠ TractA AND T.Start ≤ TractA.end) DO
    Score1 = Score(T.H1, T.H2, TractA.End, T.end)
    Score2 = TractA.Score/Score(TractA.H1, TractA.H2, T.Start, TractA.end)
    LeftScore = TractA.score × MIN {1, (100 × Score1)}
    RightScore = MIN {1, (100 × Score2)} × T.score
    X = MIN {LeftScore, RightScore}
    IF (TractB == NULL OR X < TractB.score) THEN
      TractB = NEW Tract(H1=T.H1, H2=T.H2, start=TractA.start,end=T.end,score=X)
  TractA = NULL
  IF (TractB ≠ NULL) THEN
    SortedList.add(TractB)
    MERGE-COVERED-TRACTS(SortedTractList)
    IF (SortedTractList.size() ≥ 2) THEN
      TractA = SortedTractList[1]
```

Algorithm: MERGE-COVERED-TRACTS(SortedTractList)

```
FOR Tract1 ∈ SortedTractList DO
  FOR Tract2 ∈ SortedTractList WITH Tract2 ≠ Tract1 DO
    IF (Tract1.start ≤ Tract2.start) THEN
      IF (Tract2.end ≤ Tract1.end)) THEN
           IF (Tract2.score < Tract1.score) THEN
             Tract1.score = Tract2.score
           SortedTractList.remove(Tract2)
```