Bitvector genealogy

Sunil S Nandihalli

July 24, 2011

1 Problem statement

The BitVectors are an ancient and immortal race of 10,000, each with a 10,000 bit genome. The race evolved from a single individual by the following process: 9,999 times a BitVector chosen at random from amongst the population was cloned using an error-prone process that considers each bit independently, and flips it with 0.2 probability.

Write a program to guess the reproductive history of BitVectors from their genetic material. The randomly-ordered file bitvectors-genes.data.gz contains a 10,000 bit line for each individual. Your program's output should be, for each input line, the 0-based line number of that individual's parent, or -1 if it is the progenitor. Balance performance against probability of mistakes as you see fit.

2 Problem analysis

Firstly, any genealogy can be valid solution to the above problem. However, it may not be a highly probable genealogy, given the bit vectors and the procedure used to used to clone them. So, the problem at hand is to find a highly probable genealogy for the set of bit vectors given to us. The problem can be treated as *minimum spanning tree* problemwith every node being a bit vector and a link between nodes indicating a parent-child relationship. The weight of the edge is just equal to *hamming distance* between the two bit vectors. The shorter the hamming distance, the higher the probability that there is a parent-child relationship between the nodes under consideration.

2.1 Probability of parent-child relationship

Let us say that there are two bit vectos A and B, and they are d hamming distance apart, with the length of the bitvector being n and the probability of flipping the bit during the cloning process be p, then the probability that A was the parent of B is given by $p^r(1-p)^{n-r}$. From this expression, it is clear that, smaller the hamming distance, higher is the probability there is parent child relationship among them. One of the key thing to note is that, just because the probability is p, the parent child need not be np bits apart to have a highly probable parent child relation ship. In fact, as long as p is less than 0.5, the closer the bit vectors are higher the probability of parent child relationship. However, if the p exceeds 0.5, then farther the bit vectors, the more probable will be their relationship. As you may have noticed, I have not mentioned as to who would be the parent and who would be the child. The reason for doing so is that both of them can be parent or child with equal probability. So, we can frame the problem as one of just calculating the minimum spanning tree as I mentioned before. However, this would still not resolve as to who was the parent or the child.

2.2 Identifying the root of the minimum spanning tree

As mentioned before, any genealogy can be true, it however may not be highly probable. Along the same lines, once the MST is calculated, any one of the nodes can be used as the root node, however, it may not lead to a highly probable genealogy to the given problem. Again to make the genealogy highly probable, we have to chose an appropriate root. This clearly implies that choice of root should maximize the number of ways in which cloning of a given pair of parent and child can happen, to increase the probability of the genealogy hence obtained.

3 Solution process

The solution process involves two things, first the calculation of the Minimum spanning tree and secondly finding the root of the tree which maximizes the number of ways it can be build the trees.

3.1 calculation of the minimum spanning tree

The problem at hand can be considered as the calculation of the minimum spanning tree where the n nodes are present in n dimensional space. Since, this is a very large dimensional problem, and the calcution of the distance can be very expensive. After solving a smaller problem by brute force, and extrapolating it to dimension of the current problem gave me an estimate of about 8 hrs, so this clearly needed an approximate solution. I have used *locality sensitive hashing* proposed by some of the work by Dr. Indyk Piotr. Locality sensitive hashing hashes nodes which hashes nodes close by to the same bucket with a high probability. Multiple such hashing functions are used. The edges that are formed by taking all combinations of two from any given bucket are considered as highly probable edges. The edges joining two points which collide in the most number of hashed buckets are considered very likely candidates. It is assumed that edges joining two points with larger number of collisions when hashed are always better than those points that collide in fewer hash buckets. This assumption can be false but will more likely be true and is taken for granted to make the problem more tractable.

The hashing function used is described here. For any bit vector x , $h(x)$ is defined as the binary number formed by taking the ${a_i}^{th}$ bit of x for i in $[0, k)$. L such hashing functions are chosen. Both, k and L determine the error involved in finding the best edge for our MST. For further details on how k and L were chosen please look at Indyk's publication. The probability of a given set of links in tree with n nodes and $n-1$ links with corresponding distances being d_i is given by $prob_{links}$

$$Probability_{links} = \prod_{i_{link}=0}^{n-1} p^{d_i}(1-p)^{(n-d_i)} \tag{1}$$

3.2 Finding the root id of the MST

As described before, in order to be able to find the most probable solution, we need to find the root id such that the number ways in which cloning can be done is maximized. The number of ways in which any tree can be created by cloning can be represented as a recursive expression.

Let P_T represent the number of ways tree T can be created and N_T rep-

resent the number of nodes in it. Let T_c^i represent the i^{th} child of tree T. Let $P_{T_c}^i$ represent the number of ways the children can be created and $N_{T_c}^i$ be the number of nodes in each of those trees. Let N_c be the number of children of the given tree T. The recursive formula to evaluate the P_T and N_T are given by

$$N_T = 1 + \sum_{i=0}^{N_c} N_{T_c}^i \tag{2}$$

The number of ways a given tree can be built is equal to the product of number of ways its child trees can be created multiplied by the number of ways in which $N_T - 1$ positions can be grouped into k groups of size $N_{T_c}^i$.

$$P_T = \prod_{i=0}^{N_c} P_{T_c}^i \left(\frac{(N_T - 1)!}{\prod_{i=0}^{N_c} N_{T_c}^i!} \right) \tag{3}$$

3.3 Calculating the quality of a given solution

As discussed before, any viable solution is a correct solution, however may not be the most probable solution. So, we need to be able to calculate the probability of a given solution at hand. It is easy to see that probability of a given solution being the actual solution is proprotional to the product of the number of ways a given tree can be built with the product of the probabilities of all the links present in the current tree. This can be calculated as

$$probability\ of\ current\ solution \propto P_T Probability_{links} \tag{4}$$

$$probability\ of\ current\ solution \propto \left[\prod_{i=0}^{N_c} P_{T_c}^i \left(\frac{(N_T - 1)!}{\prod_{i=0}^{N_c} N_{T_c}^i!} \right) \right] \prod_{i_{link}=0}^{n-1} p^{d_i} (1-p)^{(n-d_i)} \tag{5}$$

4 Implementation Details

The solution to this problem is implemented in Clojure. The implementation consists of the hashing function, the modified minimum spanning and the evaluation of the cost function described. The implementation has no additional detail that needs to be added to what is already said for the first

two topics. However, the third topic of the calculation of the cost function needs a little more attention.

4.1 Implementation of the cost function

The estimation of the cost function involve two pieces, the evaluation of $Probability_{links}$ and finding the root that maximizes the number of ways in which the tree can be built. I initially set out to optimizing the cost function as a whole, however it turned out that the contribution from the $Probability_{links}$ was more significant and had more variation in it when you chose a different set of links. This helped me to decouple the problem into two pieces, first calculate the MST independently and then just find the optimum root id in the second step. On further thought, I realized that if $p- > 0.5$ we cannot make this simplifying decoupling of the problem. In this situation, root-id will play a more significant role in improving or reduction the probability of the given tree.
In the process of calculating the optimium root id, one needs to be carefull. We would have to calculate the number of ways the tree can be with each node of the tree as root and pick the one that has the highest number of ways to be cloned. However, there are a couple of problems with the above proposition.
Firstly, there would be a lot of repetition of computation when the number of ways for a given root is calculated, that are repeated when compared to the evaluation with respect to another root. However, this can be solved using memoization of the values corresponding the sub-trees.
Secondly, even though we are using memoization, if not calculated in the right order, the stack depth can blow due to large number of recursive calls. In order to avoid this, we can use order of the subtrees obtained by progressively removing the edges in their *Prufer Order* and putting them back in the reverse order.
Thirdly, the numbers obtained during the calculation are very huge. In order to get over this problem, I do all the calculations in terms of the logarithms only.

4.2 incremental update of the free tree

The hashing function serves as a filter that picks a short edge with high probability. So, we consume the edges in the order of highest to lowest and

progressively add them to the tree. If the new edge does not create any cycles it is simply added else the cycle that gets created is examined and the worst edge in the cycle is eliminated. This process is repeated as long as we have not run out of time as specified by the user. Then the optimum root is calculated and returned.