BitVector Genealogy
Problem posted at: http://www.itasoftware.com/careers/puzzle_archive.html

A couple terms:

bV_i (or bitVector i) - The bitVector at the zero-based line number *i* in the file of bitVectors.
$P(prog)$ - The probability of bV_{a_0} being created as progenitor.
$P(bV_i)$ - The probability of bitVector i being created from its parent.
$P(bV_i{}^p)$ - The probability of bitVector i's parent being randomly chosen from the pool of bitVectors.

(Almost) Any valid solution to this problem will allow for multiple permutations of birth order to satisfy the parental constraints. Let’s assume that we have a birth order a = [a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 $_{...}$], where a_i represents the zero-based line number of the ith “born” (with a_0 as the progenitor). The probability of any particular birth order permutation can be thought of as the probability of the progenitor being created as bV_{a_0}, then the parent of bV_{a_1} being chosen, then bV_{a_1} being created from its parent, then the parent of bV_{a_2} being chosen, then bV_{a_2} being created from its parent, etc...

$$P(permutation) = P(prog) \times P(bV_{a_1}{}^p)\,P(bV_{a_1}) \times P(bV_{a_2}{}^p)\,P(bV_{a_2}) \ldots$$

$$P(permutation) = P(prog) \times \Pi_{i=1}^{9999}\,[P(bV_{a_i}{}^p)\,P(bV_{a_i})]$$

$$P(permutation) = P(prog) \times \Pi_{i=1}^{9999}\,[P(bV_{a_i}{}^p)] \times \Pi_{i=1}^{9999}\,[P(bV_{a_i})]$$

Looking at the equation, we can make the assumption that the progenitor is equally likely to have any genome, so $P(prog)$ can be considered to be a constant. For $\Pi_{i=1}^{9999}\,[P(bV_{a_i}{}^p)]$, we can look at the first few cases:

For a_1 to be born, its parent is chosen out of a population of 1. $P(bV_{a_1}{}^p) = 1$
For a_2 to be born, its parent is chosen out of a population of 2. $P(bV_{a_2}{}^p) = \frac{1}{2}$
For a_3 to be born, its parent is chosen out of a population of 3. $P(bV_{a_3}{}^p) = \frac{1}{3}$

So, $\Pi_{i=1}^{9999}\,[P(bV_{a_i}{}^p)]$ becomes $\Pi_{i=1}^{9999}\,[\frac{1}{i}]$, which is also a constant.

$P(bV_{a_i})$ is dependent on the number of similar bits between a_i and its parent. If they have h_i dissimilar bits, then:

$$P(bV_{a_i}) = (0.2)^{h_i}\,(0.8)^{10000-h_i}$$
$$P(bV_{a_i}) = (0.2)^{h_i}\,(0.8)^{10000}\,(0.8)^{-h_i}$$
$$P(bV_{a_i}) = (0.8)^{10000}\,(\tfrac{0.2}{0.8})^{h_i}$$
$$P(bV_{a_i}) = (0.8)^{10000}\,(\tfrac{1}{4})^{h_i}$$

This shows that $P(bV_{a_i})$ decreases by a factor of 4 with each additional flipped bit.

> A quick observation here: for this scenario (and any scenario where the probability of a flipped bit is less than 0.5), the probability of two bitVectors having a parent-child relationship is highest where there are zero flipped bits - not where the number of flipped bits matches the expected number of flipped bits. Knowing the probability of a flipped bit only allows us to determine our level of confidence that two bitVectors are related.

Back to our original equation:
$$P(permutation) = P(prog) \times \Pi_{i=1}^{9999}\,[P(bV_{a_i}{}^{p})] \times \Pi_{i=1}^{9999}\,[\,(0.8)^{10000}] \times \Pi_{i=1}^{9999}\,[(\tfrac{1}{4})^{h_i}]$$

Collecting the constants:
$$P(permutation) = K \times \Pi_{i=1}^{9999}\,[(\tfrac{1}{4})^{h_i}]$$

This equation shows that the probability of any permutation (in terms of birth orders) of a particular proposed solution is the same, since all the same parent-child relationships are used. So, the total probability for all permutations of the solution will be:

$$P(Total) = P(permutation) \times Total\ number\ of\ permutations$$

and since the number of flipped bits due to bV_a being created from bV_b is the same as the number of bits flipped due to bV_b being created from bV_a, once we have established a parent-child relationship, it doesn't change $P(permutation)$ if we change the direction of the parent-child relationships. Therefore, once the set of parent-child relationships are established, selecting a new progenitor will not change $P(permutation)$, it will only change the total number of possible permutations.

Strategy

My basic strategy for this problem was to arbitrarily select a progenitor, establish parent-child relationships and create a preliminary tree, select the best progenitor based on the preliminary tree, and reverse the direction of certain parent-child relationships to suit. At this genome length (10 000 bits)and reproductive fidelity, there should be almost no ambiguity as to whether or not two bitVectors have a parent-child relationship. One apparent issue with this method is that it may (in theory) be possible to increase $P(Total)$ by breaking a parent-child relationship and assigning the child bitVector to a new parent with a greater Hamming distance, which would cause an increase in the total number of permutations. If the increase in permutations outweighed the decrease in $P(permutation)$ in terms of effect on overall probability, this would be a valid move. However, because the genome is fairly long (10000 bits) and the bit fidelity is fairly high (80%), the effect will almost always lower $P(Total)$. As an example, parent-child relationships usually have about 8000 bits in common (10000*0.8), and the next closest relatives (grandparents, grandchildren, siblings) have about 6800 bits in common (10000*0.8*0.8+10000*0.2*0.2). If we moved a bitVector so that it was shown as being a child of its grandfather, $P(permutation)$ would decrease by a factor of about 4^{1200}, which is far greater than any increase you would see in the total number of permutations. This factor would become more extreme as the bitVector is moved farther away from its original position.

There were a couple conditions I used to save time while comparing bitVectors:

- No closed loops are allowed in the tree. For example, if bV_{a_1} has a parent-child relationship with bV_{a_2} and bV_{a_3}, then the latter two can't share a parent-child relationship. Since all new 'children' bitVectors are added to the current tree upon discovery, there is no need to compare any two bits that are already in the tree, since that would create a loop.
- It is usually not necessary to compare all 10 000 bits to make an assessment as to whether or not two bitVectors are unsuitable. For computational efficiency, my program compared 64 bits at a time, so the program would check the number of flipped bits after various multiples of 64 bits. Using probabilities from binomial distribution to calculate cutoff values, it would then discard any comparison between two bitVectors which have a negligible chance of having a parent-child relationship.

Once the parent/child relationships are established, and a temporary solution is created based on an arbitrary progenitor, we can look for the progenitor that maximizes overall probability. Since any particular permutation of births is equally probable with these child-parent relationships - even if the direction of certain relationships are reversed - the optimal progenitor is the one which allows for the greatest number of birth permutations. To find the number of permutations, we can use the following equation:

$$Total\ Permutations\ for\ any\ solution\ =\ \frac{T!}{\Pi_{i=0}^{(T-1)} s_i}$$

Where *T* is the total number of bitVectors in the tree, and s_i is the subtree size for bitVector *i*. The subtree size is defined as the number of descendants of a bitVector, plus itself. For example, if bV_{10} had, in its subtree, itself, 4 children and 15 grandchildren, then s_{10} would be 1+4+15, or 20.

The equation for total permutations for any solution is based on the fact that there are *T!* possible ways to arrange the birth order of T bitVectors, but the only valid permutations for a particular solution are ones where each bitVector is born before any of its descendants.

Since *T!* is a constant, total permutations are maximized when $\Pi_{i=0}^{(T-1)} s_i$ is minimized. We also don't need to know the actual value $\Pi_{i=0}^{(T-1)} s_i$, just the relative change between different cases.

Let's assume we are looking at our arbitrary progenitor (bV_a) in our tree of *T* bitVectors, and want to know if any of its children (bV_b, bV_c, bV_d, etc.) would be a more appropriate progenitor. If we selected one of its children as new progenitor (bV_b, for example), and s_b had an initial value of *j*, s_b would get a value of *T*, s_a would get a value of *T-j*, and no other family subtree sizes would be affected. Since the net effect on our equation $\Pi_{i=0}^{(T-1)} s_i$ is the replacement of *j* with *T-j* for one of the s_i values, we can see that if any child of the progenitor has a subtree size that is greater than $\frac{T}{2}$, then *T-j* will be smaller than *j*, so the substitution will increase the number of valid permutations and make a more probable tree.

> (A couple clarifications: I'd like to point out that only one child of a progenitor can have a subtree size greater than $\frac{T}{2}$, since the total of the progenitor's children's subtree sizes will be *T-1*. Also, since the subtree size of a child must be smaller than that of its parent, all descendents of bitVectors with a subtree size less than $\frac{T}{2}$ will also have a subtree size less than $\frac{T}{2}$, so there is no possibility of these descendents being a more likely progenitor. Finally, if the progenitor is replaced, the former progenitor will then have a subtree size which is less than $\frac{T}{2}$)

We can then assign our new progenitor, see if any of its children are more suitable, assign that child as the new progenitor, and repeat the process until we have a progenitor which has no children with subtree sizes greater than $\frac{T}{2}$, at which point we have the ideal progenitor for our tree.

Side note: The fact that, for any bitVector, there is a maximum of one child who has a subtree size greater than $\frac{T}{2}$, means that all bitVectors with a subtree size greater than $\frac{T}{2}$ are related through a series of parent-child relationships, with birth order from largest subtree size to smallest (since if there was a second series of bitVectors with subtree sizes greater than $\frac{T}{2}$, it would eventually converge with the first). Therefore, we can say that assigning the bitVector with the smallest subtree size above $\frac{T}{2}$ as the progenitor will result in the most probable tree (I found this relevant primarily because my bitVector objects were set up to identify parents, but not children, so it was more efficient to identify the real progenitor and then propagate up through its parents / grandparents and recursively reassign relationships that way).

In terms of tradeoffs, the most obvious one is the probability of mistakes versus performance. Comparing samples of two genomes instead of the entire genomes reduces run time but increases the probability of mistakes. Since there are 9999 parent-child relationships, I set my cutoff values to incorrectly remove a parent-child relationship with a probability of approximately 10^{-6} (for each of the 9999 relationships). This process leaves a small chance (about 1%) that the program will run and fail to identify a valid relationship, in which case multiple roots would be found. Obviously the program can be modified to check for the proper number of roots/relationships and dynamically revise the cutoff parameters if needed, but I thought it would be much simpler to manually tweak the cutoff parameters as needed, if needed. In the event that a valid link is discarded incorrectly, multiple roots will be created and the individual trees will be optimized. This will create a few errors, but most of the identified parents (about 99.9%) will be correct.

Also, Java may not have been the most efficient programming language to use. Run time could likely be improved by using another programming language, but I found the runtime (under 5 seconds) to be acceptable.

Note: Testing Data

When creating simulated data for testing, it is likely that the relationship tree which is created is not the most probable. For example, after the first child is created, the resulting tree consists of two bitVectors, and from this point going forward both are statistically equally likely to have been the progenitor. When testing, a replication of the test genealogy is not the goal; the goal should be to choose the most probable genealogy, even if it differs from the actual genealogy.

Note: 500 x 500 set

At only 500 bits per bitVector, the solution strategy should be different. With that few bits, the distinction between parent-child relationships (EV = 100 flipped bits) and the next closest relatives (EV = 160 flipped bits) is small enough that there will be a number of relationships where it is unclear as to whether or not two bitVectors have a parent-child relationship, as any cutoff value will likely exclude legitimate relationships and/or include illegitimate ones. I won't start talking about a strategy for the 500 x 500 problem, as I think that it's much less straightforward than the 10000 x 10000 solution, but I thought I'd note the difference since the 500 x 500 set was given as a way to "test" the program.

Any questions or comments, contact me - seanxmurray@gmail.com