An Algorithm for Whole Genome Shotgun Sequencing

E. Anson G. Myers

1 Introduction

The push to sequence the entire human genome is gearing up [1]. Recently there has been disagreement within the genome community as to the best approach for sequencing the human genome. While many scientists believe that clone mapping is the best solution, others argue that whole genome shotgun sequencing is a cheaper and faster alternative. Arguments for [2] and against [3] whole genome shotgun sequencing have been printed. The purpose of this paper is not to reargue the merits of whole genome shotgun sequencing, but to outline an algorithm for such sequencing and, through simulated test cases, to demonstrate its practicality.

For our algorithms we assume a large database of randomly selected human DNA fragments is availible. These fragments are dual end sequenced and fall into one of two classes. *Short* fragments are size selected so the sequenced ends overlap. Thus the sequence of the entire fragment (around 0.4-1.2 kb) is known. *Long* fragments are 5-20 kb long and sequencing both ends results in two sequences (reads) whose relative orientation and approximate spacing is known. For most of our trials we assume the number of basepairs sequenced results in a 10-fold coverage of the genome. The goal of whole genome shotgun sequencing is to reconstruct the genome from this database of fragments.

Random sampling will result in portions of the genome being uncovered by reads in the database. These uncovered pieces are called *gaps* and they make the goal of reconstructing the entire genome from the database impossible. Instead we try to reconstruct the represented portion of the genome by forming *contigs*. A contig is a set of overlapping reads. Inclusion of opposite ends of a long fragment in different contigs gives an ordering of those contigs. A set of such ordered contigs is often called a *scaffold*, though in this paper it will usually be refered to as a *walk*. So our revised goal is to find a walk which spans the genome and in which the contigs are as large as possible.

We have some additional information about the genome in the form of *sequene tagged sites* (STSs). An STS is a small sequence (around 300 bps) of the genome which is unique. The order of these STS markers is known (with some errors) as is the approximate distance between them. Due to significant progress in this field [5] [4] it is not unreasonable to assume that such markers will cover the genome and are spaced every 100-200 Kbps. Thus reconstructing the genome breaks down to the problem of reconstucting the stretches of the genome between STS markers. This will sometimes be refered to as the *inter-marker assembly problem.* Our algorithms will focus on this subproblem. One algorithm will attempt to form a walk connecting the STS markers to verify their adjacency. The other will try to reconstruct as much of the genome between the markers as possible.

There are a few obstacles our algorithms must deal with. One is *repeats*, which will erroneously make reads which come from different parts of the genome appear to overlap. Repeats are sequences that, up to some variation, are repeated along the genome. Some, such as Alu's, are relatively short (approximately 300bp) and occuring frequently. Others may be long (up to 10 kbp) and with only 2-5 occurances. Since so little of the genome has actually been sequenced the nature of the repeats is yet unknown. Therefore our algorithms should be prepared to handle a variety of repeat types. Another problem is that sometimes the reads are reported to be opposite ends of a long fragment when they are not. The algorithm should try to ensure that such erroneous information is not used in ordering the contigs.

2 DNA Simulator

To test the effectiveness of our algorithms we built a simulator. The simulator must model the database of reads from a whole human genome in such a way that the overlaps between the reads can be determined quickly. This allows us to try out many variations of our algorithm within a manageable time. Memory conservation was also an issue in the design of the simulator. These concerns had to be balanced against the need to simulate many possible characteristics of the genome.

2.1 What is Simulated

The simulator uses a number of variables to define the nature of the modeled genome. The size of the genome (in basepairs) is determined by the variable `GENOME_LEN`, and the number of types of repeats the genome contains is controled by the variable `NUM_CLASSES`. For each of these types of repeats there are four values to be set, `REPEAT_LEN`, `REPEAT_PERCENTAGE`, `CLUSTER_PROB`, and `CLUSTER_DENSITY`. The first two determine a repeat's size and frequency and the last two deal with how close together occurances are likely to appear on the genome.

REPEAT_LEN[i] is the length (in base pairs) of repeat occurances of type i. REPEAT_PERCENTAGE[i] is the percent of the genome covered by repeats of type i. These two variables together with the genome length determine the number of occurances of a a type of repeat. A repeat type i will have (GENOME_LEN $\times$ REPEAT_PERCENTAGE[i])/REPEAT_LEN[i] occurances. The simulator takes a conservative approach and assumes that each occurance of a repeat of the same type is identical.

Repeat occurances of the same type may be clustered together. For each occurance of a repeat a random number between 0 and 1 is generated. If this number is less than CLUSTER_PROB[i] then the repeat occurance is added to the current cluster, otherwise a new cluster containing that repeat is started. Notice that if CLUSTER_PROB[i] = 0 then each cluster contains only one repeat occurance (i.e. no clustering). Each cluster which contains more than one repeat occurance is given a size equal to CLUSTER_DENSITY[i] times REPEAT_LEN[i] times the number of occurances of the repeat in the cluster. These repeat occurances are then placed randomly within the cluster using a uniform distribution and assuring they don't overlap. These clusters, together with the non-clustered repeat occurances are distributed randomly across the genome using a uniform distribution and not allowing them to overlap.

The modeled genome also contains STS markers. The length of these markers is given by the variable MARKER_LEN. The lengths of the separations between the markers are calculated using the variables MARKER_SPACING and MARKER_VAR, which give the average spacing and a percentage of variation. Thus for every pair of adjacent markers the spacing is uniformly chosen from the interval MARKER_SPACING $\times$ (1 $\pm$ MARKER_VAR). STS markers are unique bits of sequence so they must be placed so they don't overlap with any repeats.

The simulator models a database which contains reads of two types, LONG and SHORT. Both types represent dual end sequenced fragments. SHORT reads represent fragments small enough so that the ends sequenced overlap, and thus the read is the entire fragment. Reads of type LONG come from the ends of long fragments. These reads do not overlap. Thus each LONG read has a mate whose relative position is known. The lengths of the fragments are determined by the variables SHORT_LEN, LONG_LEN, and LEN_VARIATION. The length of each fragment is chosen uniformly from the interval LEN $\times$ (1 $\pm$ LEN_VARIATION). The length of the read from each end of a long fragment is a constant given by READ_LEN. The ratio of long to short fragments is given by the variable LS_RATIO. The total number of reads generated depends on this and the variable COVERAGE which specifies the redundancy of the coverage of the genome. Note this is the coverage by the reads not the fragments. In other words, generating a long fragment does not cover LONG_LEN basepairs but 2 $\times$ READ_LEN the size of the two ends which are read. The fragments generated are sampled uniformly from the genome. Sometimes two LONG reads are erroneously paired (i.e. they don't actually come from opposite ends of the same fragment). To simulate this we have the variable LONG_ERR which is the probability that such an error has occured. When the simulator generates a LONG read which has an erroneous link, its paired read is taken from a random location on the genome.

2.2 Simulator Actions

There are two basic actions the simulator must perform. First, given a *subread*, the simulator must determine all reads in the database which appear to overlap that subread. A *subread* is just a connected piece of an read. It is specified by the read and its starting and ending location within the read. We will call this first action a *database query*. The second action is determining if and how two subreads are joined. This will be called a *compare*.

When working with real data the database query will involve a pairwise sequence comparison (using some filtering for speed) of the subread with all the reads in the database. This will be by far the most costly operation of the assembly process and thus we monitor how many times this operation is required for different versions of our algorithm. Actually generating a sequence and doing pairwise comparisons would be too costly both in terms of time and memory for our simulation purposes. Instead our simulator specifies all items (markers, repeats, reads) by their location in the genome. Thus we must establish rules for determining whether subreads are *joined* (appear to overlap) which simulate a pairwise comparison of sequences. Since the compare operation also simulates a pairwise comparison of sequences, these rules are used for that operation too.

Two subreads are joined if their locations on the genome overlap by at least MIN_OVERLAP basepairs. Two subreads may also be joined if the left end of one and the right end of the other are in different occurances of the same type of repeat. They must overlap within the repeat by at least MIN_OVERLAP basepairs. This means if the repeat occurances were transposed so that they coincide, then the ends of the fragments would overlap by at least MIN_OVERLAP basepairs. To allow for errors the end of an read is considered to lie within a repeat occurance even if it extends a small distance beyond the repeat. For example, if the right end of an read is at location **a** and the right end of a repeat is at location $b < a$, we say the right end of the read lies within the repeat if $a - b <$ MIN_OVERLAP.

Suppose in the above case that $a - b \geq$ MIN_OVERLAP, but the repeat occurance did overlap with the read (i.e. the repeat's ending location was at least MIN_OVERLAP greater than the read's starting location). Further suppose the left end of another read was within another occurance of the same type of repeat, but the read's right end extended beyond the repeat occurance. Then when doing a comparison of the two sequences, the two will match well for the portions within the repeats, but then the scores will suddenly start going bad. This case is illustrated in figure 1. Using the correct scoring function it is possible to determine where the scores start going bad with good accuracy. Thus we may locate the place on the read where its overlap with the repeat occurance ends (e.g. location **b** in the example above). Since the simulator does not generate sequence information, we tested our *repeat edge detection* algorithm separately over a large number of cases, varying the sequencing errors of the fragments and the average differences between occurances of the repeats (assuming they were identical here would be unfair). The tests confirmed that the algorithm works, so we added the assumption to our simulator that in these cases repeat edges may be detected.

The information returned from a data base query on a subread is a list of all reads which appear to overlap the subread, how they overlap the subread, and if and where a right and/or left repeat edge is detected. If a subread has an end within an occurance of a type of repeat that has a high frequency, then a very large number of reads will meet the criteria for being joined to that subread. Thus we limit the number of reads a database query can return to `MAX_OVERLAPS`. If this limit is reached then the above case has almost surely occured.

3 Quick Scan

The first algorithm we describe tries to form a walk connecting adjacent STS markers. The object of this algorithm is to quickly confirm that the markers are indeed adjacent. The speed of the algorithm will be measured by the number of database queries it generates. We use this metric since database queries will be by far the most time intensive operation. Putting a limit on the number of database queries allowed, we determine how frequently our algorithm can find a connecting walk between adjacent markers. Knowing this helps us determine the probability of two markers not being adjacent when the algorithm fails.

3.1 Definition of Terms and Moves

A basic data structure we will be using is a *footprint*. A footprint is a contig with some additional information. A contig of course is a list of reads with information on how they overlap. Additionally a footprint records the direction of the walk the footprint is from (left to right or right to left), how far the reads are estimated to be from the originating footprint, a list of footprints one *step* to the right or left, and other things which will be explained as they come up in the description of the algorithm. Two footprints are said to be one step away if they contain corresponding reads of type long. Since two long reads may be thought to correspond when they really don't, in our algorithm footprints must contain two pairs of corresponding reads to be considered one step apart.

The algorithm has several actions it can take. One is called *taking a step*. This action takes one footprint and creates another footprint which is one step away from the first. Since there are errors in the information about corresponding long reads, and since taking a step based on such erroneous information could result in **many** useless database queries, take step will only use *confirmed* steps. Suppose we have a list of reads whose corresponding read is in the footprint and who lie in the direction of the walk. (i.e. If the footprint is part of a walk from left to right, the reads lie to the right of the footprint.) A comparison is done pairwise on these reads. Those that are found to overlap form the base of a possible confirmed step. In order for this step to be bad both of the reads must be paired in error and also overlap on the genome. Assuming the randomness of the location of mismatched pairs, this situtation is practically impossible.

A footprint keeps a list of all the reads which have corresponding reads within the footprint and lie in the direction of the walk. We will call this list the *possible steps*. It also keeps a status of these fragments including which are confirmed and which portions are known to lie within repeats. In addition every read in the database has a field which tells which, if any, footprint it is included in. The take step algorithm then works as follows:

- Find the confirmed read which lies furthest from the footprint and is not part of another footprint
- If no such read exists, return FAILED
- Query the data base using the portion of the read not known to contain a repeat
- If a repeat edge is detected
  - Record that information
  - Use the nonrepetitive portion of the read to update the status (confirmed or no confirmed) of the possible steps
  - Go to top
- Else if `MAX_OVERLAPS` reads returned or reads returned are inconsistant
  - The queried read probably is contained in a repeat. Mark it as such.
  - Update the status of the possible steps. (Those confirmed only by the queried read are no longer confirmed)
  - Go to top
- The estimated distance from start of the walk for the new footprint is the estimated distance of the read which corresponds to the queried read plus `LONG_LEN`.
- The new footprint records the footprint which was sent to the take step procedure as it's parent. (This is used for backtracking)
- The list of footprints one step to the right and left of the new footprint is calculated by looking at the long reads which correspond to reads in the footprint. Any footprint which contains at least two such reads is added to the appropriate list.
- If the new footprint 'bumps' into another footprint, those footprints are joined.

Basically the algorithm takes a confirmed possible step (read with a corresponding read in the footprint), and does a database query on it. If there are no problems, this information is used to build another footprint that is one step further in the direction of the walk. If the data base query reveals a portion of the read to overlap a repeat occurance, this portion of the read must not be used. This will change which possible steps are confirmed since reads which only overlap within a repetitive portion should not be concidered to be confirmed. Note that if this read remains confirmed it will be chosed again (it is still the furthest confirmed possible step) and the nonrepetitive portion will be used to query the data base. If the read used to query the data base is contained completely in an occurance of a repeat, the query will not detect a repeat edge and reads which come from different portions of the genome (where the occurances of the repeat of this type are located) will be returned. If the repeat has a high frequency (large number of occurances), the number of reads returned will be varMAX_OVERLAPS and thus we will know of the problem. On the other hand, if it is a low frequency repeat, then this error could go undetected. There is, however, one detection method for some special types of this case. If the queried read is near the edge of the repeat, then some of reads returned will extend beyond the edge of the repeat. If two of these reads come from different portions of the genome, then they will both overlap on the same side of the queried read, but will not overlap with each other. In this case we say the returned reads are *inconsistant* and know that the queried fragment must be within a repeat. Lastly if the queried read lies close to an existing footprint, some of the reads returned by the data base query might be part of that footprint. We say that the new footprint *bumps* into the existing footprint and the two are then *joined* into a single footprint. We will talk more about joining footprints later. Note that the new footprint can bump into at most two different footprints, one on its left and one on its right.

Another basic action that can be done to a footprint is an *expansion*. The algorithm for expanding has as its arguments a footprint and a direction. It queries the data base with the read in the footprint which extends furthest in the direction of the expansion. The footprint is then updated by adding the new reads returned. The algorithm looks like this:

- query the data base with the read in the footprint which extends furthest in the desired direction
- if a repeat edge is detected
  - requery using the portion of the read not contained in the repeat
- else if the number of reads returned is `MAX_OVERLAPS` or the reads returned are inconsistant
  - The whole footprint might lie within a repeat. Record that.
  - return FAILED
- add the new reads to the footprint, updating the right and left links and the status of possible steps
- if the expansion bumped into another footprint, join the footprints
- if the footprint has not changed (i.e. no new reads where returned) then return FAILED

The main complications here happen if queried read overlaps a repeat. If it's outer (with respect to the footprint) edge is revealed to be in a repeat, then the data base is queried again using the nonrepetitive portion. By doing this a read may be found which spans the repeat, allowing the footprint to be further expanded. At the very least additional reads may be safely added to the footprint increasing its knowledge of links and possible steps. A repeat edge discovered that indicates a repeat which is on the interior side of the fragment, can mean one of two things. First it might be part of the repeat spanned by the footprint (the footprint keeps track of repeats it contains so this is known). In this case the data base should be queried with the outer, nonrepetitve portion of the read. If the interior side of the read lies within the repeat and the other edge of the repeat is not known, then the footprint must have been formed within a repeat. These footprints cannot be safely expanded as the queried read may or may not be from the same area of the genome as the footprint's original read. Also if no repeat edge is detected, but the number of reads returned by the query is `MAX_OVERLAPS` or the reads returned are inconsistant, then the entire queried read lies within a repeat. Since this wasn't previously discovered, the entire footprint probably lies with a repeat. In these cases the expansion fails and the footprint must be marked as 'bad'. If the data base query returns reads that are part of another footprint, then we join that footprint to the expanding footprint. It is possible if a footprint ends in a gap or a repeat that cannot be spanned that the expansion algorithm will add no reads to the footprint. In that case the expansion fails.

Joining footprint B to footprint A, involves added all the reads in B to A. This will change the list of footprints one step away from A and the status of possible steps. There also should be a data base query on an read which links the two footprints to make sure that all reads interior to the new combined footprint are present. Lastly all references to footprint B must be deleted.

3.2 The Algorithm

The basic idea of the algorithm is simple enough. The goal, as we stated earlier, is to create a walk connecting two adjacent markers using as few data base queries as possible. We start by creating a footprint that contains each marker. We then start walking from each of these footprints toward the other one. We walk from both directions since it is much more likely that these two walks will meet somewhere than that a single walk will find the target marker. We walk by taking a step if possible, if not, expand the footprint until a step is possible. If this expansion fails, take a step back and try again. This is a form of depth first search. Each walk continues until the walks meet or until the estimated distance traveled is at least as far as the average distance between markers, or the walk has run out of option (becomes stuck). While both directions are still active, we alternate steps between them.

If the initial described above has not resulted in the walks meeting, further action must be taken. There are two actions that can be taken on each footprint to increase the chance the walks will meet. The footprints can be expanded or a new step can be taken from them. The order each of these actions can be applied can be varied and we tried several different schemes to see which would be fastest. The schemes tried were:

1. For each footprint, do an exapansion if possible, otherwise take a step
2. For each footprint, take a step if possible, otherwise do an expansion
3. Loop through the footprints alternating a step and expansion round
4. For each footprint take a step, expanding if necessary until a step is possible (or can no longer expand)
5. For each footprint, expand as much as possible and then take all steps

Notice that with all these approaches we loop through the set of footprints. In the first three one action is taken on each footprint per iteration. In the last approach all actions that can be to a footprint are, so each footprint is only acted on once. Expansion on a footprint is always done first in the direction of the walk if possible. If expansion is no longer possible in that direction, it is then done in the other direction. One exception to this is that if a footprint's estimated distance from the walk's starting point is larger than the distance betweeen markers, expansion is not done in the direction of the walk. Likewise a step is never taken from such a footprint. To avoid doing unnecessary data base queries each footprint records when it is no longer possible to be expanded in a given direction.

The tests resulted in a clear win for approach 4, perhaps because it saturated one area of the genome. We also ran tests to see the best order in which to examine the footprints. These did not result in large differences, but there was a slight advantage by examining the footprints in the middle of the walks first, and working our way to the outside.

Lastly notice that our algorithm for taking a step will not create a footprint in the middle of a repeat if it can detect it. A footprint in the middle of a repeat is bad because it will contain reads which overlap with occurances of the repeat from all over the genome. Thus the step out of that footprint could be into a completely unrelated portion of the genome. It would take many extra data base queries to deal with walk that wandered outside of the markers in this way. Low frequency repeats (2-3 occurances) may easily be stepped into without being detected. However, the coverage of such footprints might be suspeciously high. We thus include with each footprint a status variable that can have the values: `GOOD`, `BAD`, `UNKNOWN`, or `SUSPECIOUS`. A footprint that is known not to be within a repeat is marked `GOOD`. This can be determined if a repeat edge is detected where the repeat lies (partially) outside the footprint. A footprint that is known to lie within a repeat is marked `BAD`. How such a footprint is determined and marked was included in the description of the algorithms for taking a step and expanding. No steps are taken from or expansions done on a `BAD` footprint. When the operation take step creates a new footprint, it calculates the average coverage of the queried subread. If this coverage exceeds a certain amount (tests show $1.5 \times$ `COVERAGE` to be a good number), the footprint is marked as `SUSPICIOUS`. A `SUSPICIOUS` footprint is not used during the first part of the algorithm (the depth first search), and if the walks have not met and further expansion of the footprints are needed, `SUSPICIOUS` footprints are only operated on after all other footprints have been tried. If there is more than one `SUSPICIOUS` footprint to be operated on, the ones with the smallest coverage are done first.

4 Inter Marker Assembly

The goal of this algorithm is to fill in as much of the genome as possible between two adjacent STS markers. The eventual output of this algorithm should therefore be a set of ordered, maximal contigs (or footprints). By maximal we mean that no reads from the database can be added to the contigs.

The possible actions and datastructures are that same as for the quick scan algorithm. However, there the goal was a speedy confirmation of the STS markers, so the algorithm attempted to cross the distance between the markers as quickly as possible. Here we're trying to assemble the whole genome between the markers so the goal is to move as completely and safely as possible. Thus in the above algorithm the basic idea was step when you can to quickly cover the distance, while the basic idea here will be expand while you can to completely cover the distance. Since we are filling in everything as we move the markers can be linked they will be when enough distance is traveled and so there is no longer a need to walk simultaneously from both directions. Thus each footprint need not record the direction of the walk. Also we add a direction to the algorithm to take a step so that a step from a footprint can be taken in either direction.

4.1 The Algorithm

The algorithm for inter marker assembly

- Create a footprint containing the left STS
- Create a target footprint containing the right STS
- Expand this footprint to the right as much as possible. The expansion will end for one of the following reasons
  1. A repeat which cannot be spanned is encountered
  2. A gap is found
  3. The target STS is reached
  4. The distance covered is more than `DIST`
- If case 3 or 4 causes the expansion to end, our algorithm is finished. Otherwise do the following
- Repeat while able to take new steps
  - Take a step to the right if new footprint is estimated to be less than `DIST` from the originating STS marker
  - Expand the new footprint to the left while possible
  - The expansion will end for one of the following reasons:
    1. A gap or repeat is encountered (hopefully the same gap or repeat which ended the right expansion of the previous footprint.)
    2. It is discovered the footprint was created in a low frequency repeat (by the discovery of the repeat's left edge).
  - If case 2 occurs, and the footprint links to the creating footprint, they will be joined, noting the repeat edge
  - Expand the footprint to the right as much as possible
  - If at any time during a footprint's creation or expansion the target STS is reached, `DIST` is set to be the estimated distance between the markers based on the connecting walk
  - Add footprint to que
- while the cue is nonempty, for each footprint in the cue repeat the above loop. Adding taking a step to the left if such a step does not result in a fooprint which appears to be to the left of the starting footprint. (i.e. estimated distance less than zero)

`DIST` is a cap on how far to try filling in the genome before giving up. It should start out with a value which is large enough to guarentee reaching the target marker if possible. This value will depend largely on `LEN_VARIATION` the variation in the marker spacing and in the long fragment length. Notice that the estimated distance from the starting marker should be more accurate here than in the quick scan algorithm since we are covering the distance using fewer steps. (Distance between fragments in the same footprint is known exactly, it's only the distance between the footprints that is estimated.) If the target STS marker is found, then `DIST` is reset to the estimated distance to this marker. The idea being that we want to fill in all of the genome possible *between* markers, so we do not take a step or expand outside of these boundaries.

It is possible for the above algorithm to end without finding the target marker. If this happens, we repeat the algorithm, this time starting at the target marker. There are a couple ways this could result in a connecting walk. A footprint in this walk might have two long reads whose corresponding reads both belong to the same footprint in the previous walk. Thus these two footprints would be linked. The previous walk did not take a step into this location because the two reads did not overlap and thus were not confirmed. Another way the walks may connect is if a footprint in the previous walk was created in a low frequncy repeat and its expansion thus ends at the edge of the repeat. If the new walk has a footprint on the other side of the repeat, those two footprints can be joined, thus connecting the walk. Note that some caution should be used here since if the repeat type has more than one occurance between the markers, this join may not be correct.

This algorithm can incorporate information from the quick scan algorithm, and thus both can be run without doing redundant data base queries. When taking a step, if the chosen read is part of a footprint formed in the quickscan algorithm, one incorporates that footprint, changing it's estimated distance accordingly, rather than doing a database query and forming a new footprint.

5 Empirical Results

This section will give the simulated test results of these algorithms.

5.1 Quick Scan Results

The first results here should be used to justify our chosen algorithm. The pieces that need to be decided on are:

- How to expand walks which have traveled a sufficient distance, but not yet met
- At what level of coverage should a potential step be considered 'suspect'
- After how many queries should the algorithm give up

Then we will want to show the success rate for our chosen algorithm under varying physical conditions. Success is the rate at which the algorithm finds connecting walks. Also of interest might be the average time it takes to find these walks. Some of the conditions that might be altered are:

- The Coverage
- The Types and Frequency of Repeats
- The Clustering of Repeats (not tried)
- Error rates in the pairing of reads (not tried)
- Potential differences in small read sizes, large read sizes, and marker spacing (I suspect this might not be of interest)
- Average large read size (not tried)
- Long/Short ratio (not tried)

Another thing which came up but is not currently possible in the simulator is to allow several different types (average size) of long reads in one simulation. Would this be somethnig I want to try to take the time to do?

5.2 Inter Marker Assembly Results

Here what we care about is how much of the genome can be sequenced and how many errors are made (both corrected and not) while doing it. These things should be examined under some of the same conditions as in the section above.

5.3 Edge Detection Results

I'm not sure if we want this here or not. What would be of interest here is how accurately the edge of a repeat is detected. The conditions that might vary here are:

- Error rate in sequencing the fragments
- Rate of difference between repeat occurences
- Coverage

References

[1] E. Marshall and E. Pennisi. NIH Launches the Final Push To Sequence the Genome. *Science* 272 (1996), 188-189

[2] J. Weber and G. Myers. Human Whole Genome Shotgun Sequencing. *Genome Research* 7 (1997), 401-409

[3] P. Green. Against a Whole-Genome Shotgun. *Genome Research* 7 (1997), 410-417

[4] T.J. Hudson, L.D. Stein, S.S. Gerety, J. Ma, A.B. Castle, J. Silva, D. Slonim, R. Baptista, L. Kruglyak, S. Xu, X. Hu, A.M.E. Colbert, C. Rosenberg, M.P. Reeve-Daly, S. Rozen, L. Hui, X. Wu, C. Vestergaard, K.M. Wilson, J.S. Bae, S. Maitra, S. Ganiatsas, C.A. Evans, M.M. DeAngelis, K.A. Ingalls, R.W. Nahf, L.T. Horton, M.O. Anderson, A.J. Collymore, W. Ye, V. Kouyoumjian, I.S. Zemsteva, J. Tam, R. Devine, D.F. Courtney, M.T. Renaud, H. Nguyen, T.J. O'Connor, C. Fizames, S. Faure', G. Gyapay, C. Dib, J. Morissette, J.B. Orlin, B.W. Birren, N. Goodman, J. Weissenbach, T.L. Hawkins, S. Foote, D.C. Page, and E.S. Lander. An STS-Based Map of the Human Genome. *Science* 270 (1995), 1945-1954

[5] M. Olson, L. Hood, C. Cantor, and D. Botstein. A Common Language for Physical Mapping of the Human Genome. *Science* 245 (1989), 1434-1435

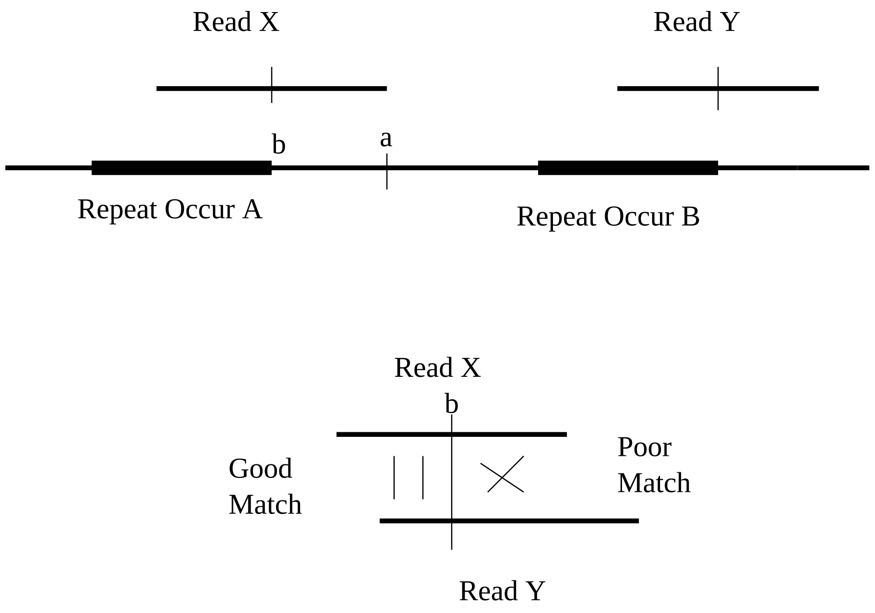

Figure 1: Repeat Edge Detection.