Mux-searching of inserts

Michael Kleber (and Jon Butler, who hasn't seen this write-up)

December 2, 2008

Abstract

Mux-based insert walking is the current best-performing method (by far) for finding all possible paths between the two end reads of an insert, the operation which underlies the ALLPATHS algorithm. This working document will broadly (and Broad-ly) describe the background and current state of Mux-based searching, and the API for relatively easy experimentation built on the Mux foundation.

1 Background, or, the world before Muxes

The ALLPATHS assembly idea demands two basic abilities:

1. We should be able to process our raw data into a state where we can proceed by only allowing perfect matches (possibly of sequences containing variable-length gaps). This is a prerequisite for our working in kmer space, where we cannot see any other type of alignment.

2. We should be able to find all closed walks between two genomic sequences of approximately known separation (generally, the end reads of an insert). By a closed walk we mean a consensus-style sequence, the "closure," in which

   (a) every base is covered by at least one read, and

   (b) every read used in that covering aligns perfectly with the closure.

   We might want to impose further restrictions — a minimum depth of coverage, a minimum read-read overlap, etc. — but such restrictions tend to cut down the possible solution space and make our job easier.

This second desire, the "insert-walking problem," has proved challenging.

One lesson we learned from our early attempts to solve the insert-walking problem is that gaps (which arise from low-quality bases in Sanger reads) lead to extreme computational difficulty. It is mathematically feasible to define the alignment and merger of reads with gaps, but the operation is uncooperative in many ways, with the result that every walk that encounters a gappy region spends an exorbitant amount of time trying to figure out all possible ways to close the gap.

We currently try to minimize this problem by precomputing all possible closures of every gappy read in the data set, and proceeding as if these "gap-filled reads," or "fills," were the original data. This work is expensive but will clearly happen at some point in the process; it seems better to get it out of the way early, and doing so leaves us the option of capping the computation and simply throwing out or chopping up reads which are too recalcitrant. From here on we assume all work is with fills, but we will lazily refer to them as reads anyway.[1]

The basic search, and its flaws

Once we can safely ignore gaps in reads, we can naively look for closures using a simple heap-based search. Maintain a set of paths to be explored, and repeatedly process the lexicographically minimal path: if it is a closure, remember it; and if it is not too long, add all of its extensions to the heap.[2] Here an extension of a path is any longer path obtained by merging in a read along a perfect proper alignment. The effect of the lexicographic ordering is to effectively act as a depth-first search on paths once they have even a single base difference between them.

This is a loose description of `DepthSearchInsert`. It does a good job in well-behaved areas of the genome, including, for example, an insert crossing over a repeat region with low copy number: the search will emerge from the repeat in many possible spots in the genome, explore each of them in some order, and discard the incorrect ones once it fails to find the closing read of the insert within the allotted distance.

However, this type of search fails badly when it enters a region with very high apparent depth of coverage (that is, a high copy-number, high identity repeat). Suppose the path being explored ends in a promiscuous kmer which

[1] One obvious difficulty is in numbering: since a read may have zero, one, or many fills, we need a way to map between read numbers and the fill numbers which take their place. The `ReadFillDatabase` class encapsulates this translation.

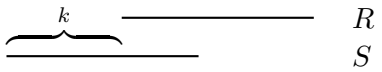
[2] A more efficient version stores some information about paths it has searched, so as to minimize redundant effort along two paths which differ in an early SNP, for example.

appears one million times in the read set. We try to extend the current path by merging with each of these million reads, many of which succeed. Some of these probably extend the read by just a single base, resulting in the next path to process, which then also ends in a kmer which appears about a million times. These million extensions are nearly all redundant. Repeating this huge batch of extensions every base quickly (slowly?) brings us to our knees. Moreover, many inserts in this genome will cross such a region, and we are stuck doing this work each time.

2 The Mux Search

We use a two-step pre-computation to overcome this obstacle:

1. We pre-compute the **graph of (leftward[3]) extensions** of each read (both fw and rc). Specifically, we calculate the directed graph with a vertex corresponding to each oriented read, and an edge $R \to S$ if S aligns perfectly and properly with R and extends R to the left. Each edge of the graph is further labelled with the extension length: $R \xrightarrow{k} S$ indicates that S sticks out past R by k kmers.[4]



2. From this graph, we **remove all transitive edges**. That is, if the graph contains both the two edges $R \xrightarrow{k} S \xrightarrow{\ell} T$ and an edge for the transitively-implied extension $R \xrightarrow{k+\ell} T$, then we delete the edge directly from R to T.

We call the result the **Mux graph**, and each edge of it is a Mux. If a read R is in a unique section of the genome, then its associated vertex in the Mux graph will have a single edge out of it, $R \to S$, where S is the unique read which minimally extends R to the left.[5] Equivalently, if you start at the left endpoint of R and move left, the next left endpoint you encounter will be that of S. If the read set supports more than one genomic sequence to the left of R, *and* if the next left endpoint you encounter would be *different*

[3]Because of how kmer paths are stored, searching for leftward extensions is more efficient than rightward. Because of this, all Mux searches progress from right to left (opposite the sense arbitrarily chosen for `DepthSearchInsert`).

[4]Actually we record both the number of kmers and the number of `KmerPathSegment`s by which S extends R.

[5]The Mux is named after this property; it stands for Minimal Unique Extension.

depending on which sequence were genomically correct, then there will be an edge from R to each of its possible minimal leftward extensions.

Details, benefits, drawbacks

To make the Mux graph work well in regions with high apparent depth of coverage, we need to be more specific about what happens when multiple reads have the same left endpoint, which we call "coterminal." Our goal is for as many reads as possible to have a unique Mux.[6] The definition which best achieves this is that if R and S are coterminal, then $R \to S$ is an edge if (a) S is shorter than R, or (b) R and S are also the same length,[7] and R has a smaller read id than S. Part (a) of this definition has a computationally beneficial side-effect: when building the Mux graph, if we ever see a read R with a shorter coterminal read S, we immediately know that R will have a unique mux — the longest coterminal read shorter than R — and we can ignore all other reads.

The intent is to replace the insert-walking problem with the problem of walking in the Mux graph. That is, we wish to find all ways to begin at the Mux graph vertex corresponding to the right end read[8] of an insert, and follow edges of the graph, tracking the running sum of their kmer-length labels, until we find the vertex corresponding to the left end read,[9] at an acceptable total distance.

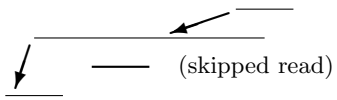
This is computationally enormously beneficial. Having removed transitive edges from the graph means that we can reach each partial walk in the search space in one and only one way. A branch in the Mux graph implies a branch in the underlying genomic data. And returning to our motivating problem, this substantially improves performance in the high apparent

[6] We say $R \to S$ is a Mux of R, so "R has a unique Mux" means that there is one edge leaving vertex R, or equivalently, R has a unique minimal leftward extension.

[7] i.e., they are identical reads; we may not be able to simply delete one of them because of pairing information

[8] Technically, at the vertices corresponding to all of its fills.

[9] Actually, a vertex corresponding to any of its fills, *or* a vertex corresponding to any fill which subsumes the closing read. In some data sets, it is possible to walk past a read without it appearing as a Mux:

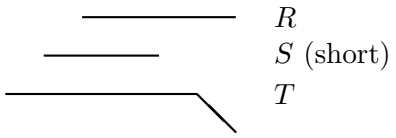


Since we walk in the Mux graph without looking at kmers, we must pre-compute the list of such subsumed reads.

depth of coverage scenario described above. Passing through such a region in the Mux graph is essentially invisible: with the above rules for coterminal reads, every read has a unique Mux, and we pass through the region in linear time, touching each read exactly once. By contrast, the naive heap-based search can take quadratic time.

As in the heap-based search (cf. footnote 2), we can make the search more efficient by maintaining a record of previously-explored states, so as to avoid duplicated effort after going around a SNP. Here again, we benefit from the search living purely in the Mux graph: the exploration state is completely summarized by a pair (V, k), where V is a vertex in the Mux graph (an oriented read), and k is the distance into the search. In particular, once we have explored $\cdots \to R \xrightarrow{k_1} S_1 \xrightarrow{\ell_1} T \to \cdots$, we will spend no time on $R \xrightarrow{k_2} S_2 \xrightarrow{\ell_2} T$ if $k_1 + k_2 = \ell_1 + \ell_2$.

Walking in the Mux graph, however, is not a perfect proxy for walking with reads. Just because the graph's edges permit us to walk $R \to S \to T$, there is no guarantee that R and T perfectly align with each other: they do align on the part where they both match S (perfect alignment is transitive!), but if S is short, T may disagree with R to the right of S:



This can cause us to search regions of the Mux graph which are actually impossible to reach at the path level, leading to two potential problems: finding closures in terms of Muxes that are illegal as paths (like $R \to S \to T$ above), and spending time in useless areas of the search space (everything of the form $R \to S \to T \to \cdots$). Neither of these proves serious. We can filter out path-illegal Mux closures at the end of the search, when we need to convert from Mux space to path space anyway. And empirically, the extra time spent during the search is more than made up for by the fact that searching in the Mux graph, without any reference to the underlying paths, is far faster than any search which constantly checks for full consistency.

A second apparent drawback to the Mux approach is that it has a large setup cost, the construction of the Mux graph. If we are going to walk a substantial fraction of the genome, this is time well spent. However, the method just described forces us to choose which reads we make available for walking at the Mux graph construction stage. In particular, it is not well-suited to walking different inserts with different sets of permissible reads

(e.g., those believed to live on one contig according to an existing assembly). The solution to this problem is presented in section 3, Extensibility.

The Mux-based search described above is implemented by the class `KmerPathMuxSearcher`, which can be called either programmatically or from the command line via the executable `MuxSearchInsert`.

Result of the search

The closures found by a Mux-based insert walk are returned in a graph structure (an object of class `MuxSearchWalk`, one field of the return structure `MuxSearchResult`).

In the simplest case, the returned graph is a representation of the subgraph of the Mux graph containing exactly those vertices (oriented reads) which appear in successful walks of the insert. In this representation, though, each vertex carries an extra datum: the distance (in kmers) between the left endpoint of its associated read and the left end of the insert. If there are two paths in the Mux graph from a read to the end of the insert, they may be different lengths; in such a case, the returned `MuxSearchWalk` would have two entirely independent vertices, with the same associated read but different distance-to-end fields.

All caveats about the Mux graph apply to this return graph as well. Note in particular that (1) it may well encode illegal closures which appear good in Mux space but not in kmer space, and (2) it has vertices corresponding to most reads that appear in successful closures of the insert, but not to subsumed, skipped ones (see footnote 9).

The method `MuxSearchResult::CalculateAllClosures` instructs the `MuxSearchResult` to convert the results from Mux-space to kmer space; the results are stored in another field, the `vec<KmerPath> all_closures`, which is otherwise unfilled. This might be a very time-consuming operation. The graph representation efficiently represents parallel paths which disagree at individual bases, e.g. the two-haplotype case in the presence of SNPs. Conversion to kmer space may, in the worst case, produce a number of closures which is exponential in the size of the graph.

3 Extensibility

The recently-completed redesign of Mux-based searching is built with two layers of parameterization which should make it easy to implement new insert walking ideas. This design document will describe both layers.

The inner layer makes it easy to impose extra criteria on the reads used in walking inserts. For example, it was used to implement "paired pairs" insert walking, where two reads are only allowed to align if their partners do as well. Trying new things involves writing only a minimal amount of code, and certainly does not require deep knowledge of the innards of Mux searching. It is covered in section 3.1.

The outer layer allows broad access to how the search is carried out; messing with this does require understanding the basics of the Mux search engine. It was used to implement the inner layer just described, the idea "Walk an insert as described in Section 2, with the added requirement that paths are covered by 'good' reads (whatever 'good' means)." Some useful information about this is given in section 3.2.

Much thanks to Will and Pablo for engineering consultation.

3.1 Coverage by good reads

In the course of exploring insert walking, we have wanted to try various strategies which restrict the set of available reads. Examples include:

1. The original ALLPATHS proposal mentioned the possibility of walking some inserts take into account compatibility with other inserts walked earlier.

2. When comparing insert-walking with a preexisting assembly, it is helpful to walk an insert using only the reads that the assembly believes fall in the area.

3. Adapting to new types of data sets, e.g. high-coverage/short read/tight insert length libraries where we want to required both end reads to align ("paired pairs").

Older insert-walking methodologies adapted well to such demands: just write a filter which restricts the permissible ways to extend a walk. The Mux method is harder to adapt: the read S may not meet whatever criterion you impose on reads, but we cannot reject it out of hand, as it is a necessary link in the Mux walk $R \to S \to T$, even when R and T overlap.

Mux-based insert walking now has an easy programming interface which allows the addition and run-time selection of these sorts of criteria.

We adapt Mux searching to dynamically changing notions of which reads are good to use as follows. The underlying Mux-based search engine remains unchanged; we still search by looking for paths in the entire Mux graph. As we search, we keep track of which reads in the path we are currently

exploring should be considered "good." Mux searching always tracks the left endpoints of reads; if we also note their lengths, then we can easily verify that any potential closure is in fact covered by good reads. Moreover, if the longest read has length ℓ, then we can prune any search which has gone ℓ kmers since the end of its last good read. (This paragraph's ideas are implemented at the outer level of abstraction, as described in section 3.2.)

Definitions of good

The mechanics of checking for coverage by good reads is common to all the strategies we want to support; the only difference is the notion of what makes a read good. The `MuxSearchPolicy` is a virtual base class; classes derived from it can act as judges of which reads are considered good. We will call these MSPs.

The fundamental job of an MSP is to repeatedly answer the question "do you consider this Mux a good way to extend the current search?" To help it answer, the MSP is handed both the Mux in question and information about the path in the Mux graph leading from the opening read to the proposed extention. Specifically, it can find out which Muxes were used, at what distance into the search they appeared, and whether or not they were considered good when they were added.

This query is posed by a call to `PushAndGiveOpinion`, a purely virtual function of the base class `MuxSearchPolicy` which any MSP must override; returning true indicates a good read. For clarity, Figure 1 contains the complete code for a sample MSP to walk using a given subset of the reads. It ignores all the available information about the state of the search, and bases its decision entirely on the id of the read to be added.

Figure 1: A sample MSP to walk using a given subset of the reads.

```
class MSP_TheseReadsOnly : public MuxSearchPolicy {
  hash_set<int> m_good_reads;

public:
  MSP_TheseReadsOnly( const vec<int>& reads )
    : m_good_reads( reads.begin(), reads.end(),
                    2*reads.size() ) {}  // keep hash load under 50%
  bool PushAndGiveOpinion( const Mux& final_mux,
                           const MuxSearchState& search,
                           const vec<Bool>& is_good ) {
    return( m_good_reads.count( final_mux.GetPathId().GetId() ) );
  }
};
```

The speed of an MSP's `PushAndGiveOpinion` has a large impact on search performance. For example, in Figure 1 we use a hash set for constant expected time lookup; a binary search would be too slow if the set of reads is large.

As of this writing, the files `MuxSearchPolicy.{h,cc}` define:

1. `MSP_TheseReadsOnly`, shown above.

2. `MSP_PairedPairs`, which implements paired pairs searching. Note that this class holds pointers to the kmer path and pairing data of the assembly, but that fact is invisible to the surrounding search.

Future policies should go in the same place.

An MSP might want to maintain some internal state, instead of just responding to queries and forgetting what it has been asked. This requires a little more knowledge of how MSPs are called.

Querying derived Mux search policy classes

At any time during a Mux search, the search state is primarily a stack of Muxes, representing a path in the Mux graph joining the search start note to the node currently being explored.[10] The state being searched changes by pushing and popping this stack, i.e., by adding/removing one final edge to/from the path.

Recall that the Mux-based search is run by a `KmerPathMuxSearcher`. This searcher can be given any number of MSPs, by repeated calls to its member function `AddPolicy( MuxSearchPolicy* MSP )`. Before pushing a new Mux on to the stack, the searcher queries each of these policies, and it declares a read good if and only if all of the classes agree that it is good.

The record of which reads are "good by unanimous agreement" is what is made available to MSPs when they are asked for their opinion of future Muxes, as the `vec<Bool>& is_good` argument to `PushAndGiveOpinion`. If an MSP wants access to its own opinion of earlier reads, not the consensus, it must maintain the data itself; read on.

MSPs with internal state; a technical detail revealed

An MSP might want to maintain additional internal state. It might, for example, consider reads to be of several different types, and want to keep

[10] The `MuxSearchState` structure, seen in Figure 1, also contains two parallel stacks which do not matter here: the length of this path in kmers, and the set of still-unexplored outgoing edges from each vertex along this path.

track of which type each read is. It can do so by keeping a stack of information, which it then needs to push and pop as the search adds and removes reads from the stack representing the state of the search. The function `PushAndGiveOpinion` is so named precisely because this is the MSP's opportunity to push any state information. There is a corresponding function `Pop()`, which takes no arguments, but which lets the MSP know that the last read on the search stack is being removed.

There is one other consideration for MSPs that maintain internal state, and it exposes a detail which has been swept under the rug for the entire discussion of walking using only good reads. Recall that we avoid duplicated effort by pruning the search if we ever reach a previously-explored state. In the basic Mux search, it is enough to remember the pair (V, k), where V is a vertex in the Mux graph (an oriented read) and k is a distance into the insert in kmers.

In searching using only good reads, this no longer suffices: two searches that reach the same (V, k), but with different histories of which reads are considered good, may have different futures.

If you search using coverage by good reads, this issue is dealt with automatically: the search instead records (V, k, h), where h is a hash of the good reads as far back as the longest read can reach. In the presence of an MSP which maintains internal information,[11] this hash is insufficient.

Therefore any derived Mux searching policy may also override the virtual function `HashOfPosition`, which returns a `longlong` that feeds into the hash h. The hash used ought to be fast, but it doesn't have to be good from the cryptographic point of view — it just needs to protect against random collisions, not deliberate attacks.

3.2 Changing what happens during a search

In this section, we very briefly discuss how the `KmerPathMuxSearcher` works internally. Its architecture involves a layer of abstraction which was designed to allow easy switching between the plain Mux-based search described in Section 2 and the good read coverage version described in Section 3.1. As discussed above, this required the search to track extra information, prune the search according to a new criterion, and modify the notion of when a search state had already been seen.

[11] Or one which makes use of the bad reads in its decision-making process — which seems like a bad idea right now, but maybe there is a good reason to do so that I haven't thought of.

The fundamental logic of Mux-based searching is contained in the function `KmerPathMuxSearcher::SearchDirector`. This function is templatized, and its first argument is some kind of Mux Search Agent, or MSA. An MSA is a class which implements the various primitive operations of Mux-based searching, and the `SearchDirector` runs the search by calling the MSA's member functions in appropriate sequence.

Figure 2 contains an only slightly abbreviated copy of the code for the `SearchDirector`. Conceptually, there is a stack of all the Muxes which might extend the current path; the call `msa.HasNextMux()` checks whether this is empty, and `msa.GetNextMux()` pops that stack and prepares the MSA to explore the popped Mux.

The files `MuxSearchAgent.{h,cc}` define:

1. A class `MuxSearchAgentBase`, which implements all of the operations called by the search director, sometimes trivially: `MayIPush()` and `IsStateNewRecordIfSo()` always return true.

2. The derived class `MuxSearchAgentSimple`, which implements the Mux search of Section 2. This just changes `IsStateNewRecordIfSo()` to recognize states (V, k) the second time it sees them.

   Thanks to templates, the trivial call to `MayIPush()` can be inlined and optimized away at compile time. Hooray!

3. The derived class `MuxSearchAgentGoodReads`, which implements the "coverage by good reads" search described in Section 3.1. `MayIPush()` now queries all mux search policies, records the result, and returns false (i.e., prunes the search) if it has been too long since the last good read. `IsStateNewRecordIfSo()` implements the hash function scheme described above.

   If a derived class overrides some action central to the search, it must call the base class's version of the action in addition to doing whatever it wants to do. For instance, `MuxSearchAgentGoodReads::Pop()` calls `MuxSearchAgentBase::Pop()`, and then pops its own stack (of which reads are good) and calls the `Pop()` methods of all of its MSPs.

I hope that the director/agent scheme described here will be robust enough to accommodate new ideas about how Mux-based searching ought to work.

```
template<typename MuxSearchAgent>
void KmerPathMuxSearcher::SearchDirector( MuxSearchAgent msa, ... ) {
  msa.Setup( ... );

  while( ! msa.Done() ) {
    // If the current final mux has no more extensions to explore,
    // then we are done with it: pop one level on the mux stack.
    if( ! msa.HasNextMux() ) {
      msa.Pop();
      continue;
    }

    msa.GetNextMux();  // What might we explore next?

    if( msa.NextMuxGoesTooFar() )
      continue;

    if( ! msa.IsStateNewRecordIfSo() )
      // Check for convergence with previously-saved closures
      msa.SaveIfConverged();
    else {
      // If this will be our first visit to this state:
      if( HitSearchLimit() )
        break;
      if( msa.MayIPush() ) {
        msa.PushMux();
        msa.SaveIfClosed();
        // Note that we keep exploring deeper even if this was a closure:
        // we might be in a repeat region where we'll close again later.
      }
    }
  }
}
```

Figure 2: The basic logic of the Mux-based search. The committed version improves on this just a little... but this is too long already, so I'll stop here.