Bio::SearchIO HOWTO

Jason Stajich
Duke University [http://www.duke.edu]
University Program in Genetics [http://upg.duke.edu]Center for Genome Technology [http://cgt.genetics.duke.edu]

Duke University Medical Center
Box 3568
Durham,
North Carolina
27710-3568
USA

<jason-at-bioperl.org>

Brian Osborne
Cognia Corporation [http://www.cognia.com]

NYC, NY 10022
USA

<brian-at-cognia.com>

2002-07-14

Revision History

Revision 0.1	2002-07-14	js
first draft		
Revision 0.2	2002-10-11	js
added info on extending Search objects		
Revision 0.3	2003-02-13	BIO
added table and text to Parsing section		
Revision 0.4	2003-09-10	BIO
updated Parsing section		

This is a HOWTO written in DocBook (SGML) for the reasoning behind the creation of the Bio::SearchIO system, how to use it, and how one goes about writing new adaptors to different output formats. We will also describe how the Bio::SearchIO::Writer modules work for outputting various formats from Bio::Search objects.

Table of Contents

1. Background

One of the most common and necessary tasks in bioinformatics is parsing analysis reports so that one can write programs which can help interpret the sheer volume of data that can be produced by processing many sequences. To this end the Bioperl project has produced a number of parsers for the ubiquitous BLAST report. Steve Chervitz wrote one of the first Bioperl modules for BLAST called Bio::Tools::Blast. Ian Korf allowed us to import and modify his BPlite (*B*last *P*arser) Bio::Tools::BPlite module into Bioperl. This is of course in a sea of BLAST parsers that have been written by numerous people, but we will only cover the ones associated directly with the Bioperl project in this document. One of the reasons for writing yet another BLAST parser in the form of Bio::SearchIO is that even though both Bio::Tools::Blast and Bio::Tools::BPlite did their job correctly, and could parse WU-BLAST and NCBI-BLAST output, they did not adequately genericize what they were doing. By this we mean everything was written around the BLAST format and was not easily applicable to parsing say, FastA alignments or a new alignment format. One of the powerful features of the Object-Oriented framework in Bioperl is the ability to read in say, a sequence file, in different formats or from different data sources like a database or XML-flatfile, and have the program code process the sequences objects in the same manner. We wanted to have this capability in place for analysis reports as well and thus the generic design of the Bio::SearchIO module.

2. Design

The Bio::SearchIO system was designed with the following assumptions: That all reports parsed with it could be separated into a hierarchy of components. The Result which is the entire analysis for a single query sequence. Multiple results can be concatenated together into a single file (i.e. running blastall with a fasta database as the input file rather than a single sequence). Each result is a set of Hits for the query sequence. Hits are sequences in the searched database which could be aligned to the query sequence and met the minimal search parameters such as e-value threshold. Each Hit has one or more High-scoring segment Pairs (HSPs) which are the alignments of the query and hit sequence. Each Result has a set of one or more Hits and each Hit has a set of one or more HSPs, and this relationship can be used to describe results from all pairwise alignment programs including BLAST, FastA, and implementations of the Smith-Waterman and Needleman-Wunsch algorithms.

A design pattern, called Factory, is utilized in object oriented programming to separate the entity which process data from objects which will hold the information produced. In the same manner that the Bio::SeqIO module is used to parse different file formats and produces objects which are Bio::PrimarySeqI compliant, we have written Bio::SearchIO to produce the Bio::Search objects. Sequences are a little less complicated so there is only one primary object (Bio::PrimarySeqI) which Search results need three main components to represent the data processed in a file: Bio::Search::Result::ResultI (top level results), Bio::Search::Hit::HitI (hits) and Bio::Search::HSP::HSPI (HSPs). The Bio::SearchIO object is then a factory which produces Bio::Search::Result::ResultI objects and the Bio::Search::Result::ResultI objects contain information about the query, the database searched, and the full collection of Hits found for the query.

3. New Functionality

The generality of the SearchIO approach is demonstrated by large number of report formats that have appeared since its introduction. These formats include AXT format reports (BLAT, BLASTZ), NCBI tabular output (-m 8 or -m 9 options), NCBI Blast XML, chadosxpr format flat databases, Exonerate output, FASTA output, hmmsearch output (HMMER), megablast output, PSL format output (BLAT), sim4 output, WABA output, and output from Wise.

4. Parsing with Bio::SearchIO

This section is going to describe how to use the SearchIO system to process reports. We'll describe BLAST reports but the idea is that once you understand the methods associated with the objects you won't need to know anything special about other SearchIO parsers.

Before we get into the details we should admit that there is some confusion about the names and functions of the objects for historical reasons. Both Steve Chervitz and Jason Stajich have implemented parsers in this system. Steve created the psiblast parser (which does parse regular BLAST files too) and a host of objects named Bio::Search::XXX::BlastXXX where XXX is HSP, Hit, and Result. These objects are created by his Bio::SearchIO::psiblast implementation. The objects Jason has created are called Bio::Search::XXX::GenericXXX where, again, XXX is HSP, Hit, and Result. Because of some of the assumptions made in Steve's implementation and his utilization of what is known as 'lazy parsing', it is probably not going to be very easy to maintain his system without his help. On the other hand Jason has tried to make his implementations much easier to follow because all the parsing is done in one module.

The important take home message is that you cannot assume that methods in the BlastXXX objects are in fact implemented by the GenericHSP objects. More likely than not the BlastXXX objects will be deprecated and dismantled as their functionality is ported to the GenericHSP objects. For this reason we'll only be discussing the Generic* objects, though we'll use the terms 'hit', 'HSP', and 'result'.

Here's example code which processes a BLAST report finding all the hits where the HSPs are greater than 100 residues and the percent identity is less than 75 percent. This code demonstrates that a result, in this case from a BLAST report, contains one or more hits, and a hit contains one or HSPs.

```
use strict;
use Bio::SearchIO;

my $in = new Bio::SearchIO(-format => 'blast',
                           -file   => 'report.bls');
while( my $result = $in->next_result ) {
  while( my $hit = $result->next_hit ) {
    while( my $hsp = $hit->next_hsp ) {
      if( $hsp->length('total') > 100 ) {
        if ( $hsp->percent_identity >= 75 ) {
          print "Hit= ",       $hit->name,
               ",Length=",     $hsp->length('total'),
               ",Percent_id=", $hsp->percent_identity, "\n";
        }
      }
    }
  }
}
```

The example above shows just a few of the many methods available in SearchIO. In order to display all these methods and what they return let's use a report as input, a simple BLASTX result:

```
BLASTX 2.2.4 [Aug-26-2002]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
```

```
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.

Query= gi|20521485|dbj|AP004641.2 Oryza sativa (japonica
cultivar-group) genomic DNA, chromosome 1, BAC clone:B1147B04, 3785
bases, 977CE9AF checksum.
         (3059 letters)

Database: test.fa
           5 sequences; 1291 total letters



                                                                  Score    E
Sequences producing significant alignments:                       (bits) Value

gb|443893|124775 LaForas sequence                                   92   2e-022

>gb|443893|124775 LaForas sequence
          Length = 331

 Score = 92.0 bits (227), Expect = 2e-022
 Identities = 46/52 (88%), Positives = 48/52 (91%)
 Frame = +1

Query: 2896 DMGRCSSGCNRYPEPMTPDTMIKLYREKEGLGAYIWMPTPDMSTEGRVQMLP 3051
            D+ + SSGCNRYPEPMTPDTMIKLYRE EGL AYIWMPTPDMSTEGRVQMLP
Sbjct: 197  DIVQNSSGCNRYPEPMTPDTMIKLYRE-EGL-AYIWMPTPDMSTEGRVQMLP 246


  Database: test.fa
    Posted date:  Feb 12, 2003  9:51 AM
  Number of letters in database: 1291
  Number of sequences in database:  5

Lambda     K      H
   0.318    0.135    0.401

Gapped
Lambda     K      H
   0.267   0.0410    0.140


Matrix: BLOSUM62
Gap Penalties: Existence: 11, Extension: 1
Number of Hits to DB: 7140
Number of Sequences: 5
Number of extensions: 180
Number of successful extensions: 2
Number of sequences better than 10.0: 2
Number of HSP's better than 10.0 without gapping: 1
Number of HSP's successfully gapped in prelim test: 0
Number of HSP's that attempted gapping in prelim test: 0
Number of HSP's gapped (non-prelim): 1
length of database: 1291
effective HSP length: 46
effective length of database: 1061
effective search space used:  1032353
frameshift window, decay const: 50,  0.1
T: 12
A: 40
X1: 16 ( 7.3 bits)
X2: 38 (14.6 bits)
X3: 64 (24.7 bits)
S1: 32 (17.6 bits)
```

Table 1 shows all the data returned by methods used by the Result, Hit, and HSP objects when the report shown above is used as input. Note that many of the methods shown can be used to either get or set values, but we're just showing what they get.

Object	**Method**	**Example**	**Description**
Result	algorithm	BLASTX	algorithm

Object	Method	Example	Description
Result	algorithm_version	2.2.4 [Aug-26-2002]	algorithm version
Result	query_name	gi|20521485|dbj|AP004641.2	query name
Result	query_accession	AP004641.2	query accession
Result	query_length	3059	query length
Result	query_description	Oryza sativa ... 977CE9AF checksum.	query description
Result	database_name	test.fa	database name
Result	database_letters	1291	number of residues in database
Result	database_entries	5	number of database entries
Result	available_statistics	effectivespaceused ... dbletters	statistics used
Result	available_parameters	gapext matrix allowgaps gapopen	parameters used
Result	num_hits	1	number of hits
Result	hits		Search::Hit::GenericHit object
Hit	name	gb|443893|124775	hit name
Hit	accession	443893	accession
Hit	description	LaForas sequence	hit description
Hit	algorithm	BLASTX	algorithm
Hit	raw_score	92	hit raw score
Hit	significance	2e-022	hit significance
Hit	bits	92.0	hit bits
Hit	hsps		Search::HSP::GenericHSP object
Hit	num_hsps	1	number of HSPs in hit
Hit	locus	124775	locus name
Hit	accession_number	443893	accession number
HSP	algorithm	BLASTX	algorithm
HSP	evalue	2e-022	e-value
HSP	frac_identical	0.884615384615385	Fraction identical
HSP	frac_conserved	0.923076923076923	desc
HSP	gaps	2	number of gaps
HSP	query_string	DMGRCSSG ...	string from alignment
HSP	hit_string	DIVQNSS ...	string from alignment
HSP	homology_string	D+ + SSGCN ...	string from alignment
HSP	length('total')	52	length of HSP
HSP	length('hit')	50	length of hit minus gaps
HSP	length('query')	156	length of query minus gaps

Object	Method	Example	Description
HSP	hsp_length	52	desc
HSP	frame	0	frame, GFF convention
HSP	num_conserved	48	number of conserved residues
HSP	num_identical	46	number of identical residues
HSP	rank	1	rank of HSP
HSP	seq_inds('query','identical')	(966,971,972,973,974,975 ...)	identical positions as array
HSP	seq_inds('query','conserved')	(967,969)	conserved positions as array
HSP	seq_inds('hit','identical')	(197,202,203,204,205 ...)	identical positions as array
HSP	seq_inds('hit','conserved')	(198,200)	conserved positions as array
HSP	score	227	score
HSP	bits	92.0	bits
HSP	range('query')	(2896,3051)	start and end as array
HSP	range('hit')	(197,246)	start and end as array
HSP	percent_identity	88.4615384615385	% identical
HSP	strand('hit')	1	strand of the hit
HSP	strand('query')	1	strand of the query
HSP	start('query')	2896	start position from alignment
HSP	end('query')	3051	end position from alignment
HSP	start('hit')	197	start position from alignment
HSP	end('hit')	246	end position from alignment
HSP	matches('hit')	(46,48)	number of identical and conserved as array
HSP	matches('query')	(46,48)	number of identical and conserved as array
HSP	alignment		Bio::SimpleAlign object

Table 1. SearchIO Methods

Table 1 shows that a method can return a string, an array, an array or a string, or an object. When an object is returned some additional code will probably be needed to get the data of interest. For example, if you wanted a printable alignment after you'd parsed BLAST output you could use the get_aln() method, retrieve a Bio::SimpleAlign object and use it like this:

```
use Bio::AlignIO;
# $aln will be a Bio::SimpleAlign object
my $aln = $hsp->get_aln;
my $alnIO = Bio::AlignIO->new(-format=>"msf");
my $alignment_as_string = $alnIO->write_aln($aln);
```

On one hand it appears to be a complication, but by entering the worlds of the AlignIO and SimpleAlign objects you now have access to their functionality and flexibility. This is the beauty of Bioperl!

Some of these methods deserve a bit more explanation since they do more than simply extract data directly from the output. For example, the ambiguous_aln() method is designed to tell us whether two or more HSPs from a given hit overlap, and whether the overlap refers to the queries or the hits, or both. One situation where overlaps would be found in one but not the other arises where there are repeats in the query or hit. The ambiguous_aln() method will return one of these values: q - query sequence contains overlapping sub-sequences while hit sequence does not. s - hit sequence contains overlapping sub-sequences while query does not. qs - query and hit sequences contain overlapping sub-sequences relative to each other. - - query and hit sequence do not contain multiple domains relative to each other OR both contain the same distribution of similar domains.

Another method that's useful in dissecting an HSP is the seq_inds() method of the HSP object. What this method does is tell us what the positions are of all the identical or conserved residues in an alignment, query or hit. It could be used like this:

```
# put all the conserved matches in query strand into an array
my @str_array = split "",$hsp->query_strand;
foreach ( $hsp->seq_inds('query','conserved') ){
   push @conserved,$str_array[$_ - 1];
}
```

In most cases the SearchIO methods extract data directly from output but there's one important exception, the frame() method of the HSP object. Instead of using the values in the BLAST report it converts them to values according to the GFF specification, which is a format used by many Bioperl modules involved in gene annotation (for more on GFF see http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml).

Specifically, the frame() method returns 0, 1, or 2 instead of the expected -3, -2, -1, +1, +2, or +3 in BLAST. GFF frame values are meaningful relative to the strand of the hit or query sequence so in order to reconstruct the BLAST frame you need to both the strand, 1 or -1, and the GFF frame value:

```
my $blast_frame = ($hsp->query->frame + 1) * $hsp->query->strand;
```

Our simple table of methods does not show all available arguments or returned values for all the SearchIO methods. The best place to explore any method in detail is http://doc.bioperl.org which provides the HTML versions of the Perl POD (Plain Old Documentation) that is embedded in every well-written Perl module. Another useful source of code is the examples/searchio/ directory in the Bioperl package.

5. Implementation

This section is going to describe how the SearchIO system was implemented, it is probably not necessary to understand all of this unless you are curious or want to implement your own Bio::SearchIO parser. We have utilized an event-based system to process these reports. This is analagous to the SAX (Simple API for XML) system used to process XML documents. Event based parsing can be simply thought of as simple start and end events. When you hit the beginning of a report a start event is thrown, when you hit the end of the report an end event is thrown. So the report events are paired, and everything else that is thrown in between the paired start and end events is related to that report. Another way to think of it is as if you pick a number and color for a card in a standard deck. Let's say you pick red and 2. The you start dealing cards from our deck and pile them one on top of each other. When you see your first red 2 you start a new pile, and start dealing cards onto that pile until

you see the next red 2. Everything in your pile that happened between when you saw the beginning red 2 and ending red 2 is data you'll want to keep and process. In the same way all the events you see between a pair of start and end events (like 'report' or 'hsp') are data associated with object or child object in its hierarchy. A listener object processes all of these events, in our example the listener is the table where the stack of cards is sitting, and later it is the hand which moves the pile of cards when a new stack is started. The listener will take the events and process them. We've neglected to tell you of a third event that is thrown and caught. This is the characters event in SAX terminology which is simply data. So one sends a start event, then some data, then an end event. This process is analagous to a finite state machine in computer science (and I'm sure the computer scientists reading this right are already yawning) where what we do with data received is dependent on the state we're in. The state that the listener is in is affected by the events that are processed.

A small caveat, in an ideal situation a processor would throw events and not need to keep any state about where it is, it would just be processing data and the listener would manage the information and state. However, a lot of the parsing of these human readable reports requires contextual information to apply the correct regular expressions. So in fact the event thrower has to know what state it is in and apply different methods based on this. In contrast the XML parsers simply keep track of what state they are in, but can process all the data with the same system of reading the tag and sending the data that is inbetween the XML start and end tags.

All of this framework has been built up so to implement a new parser one only needs to write a module that produces the appropriate start and end events and the existing framework will do the work of creating the objects for you. Here's how we've implemented event-based parsing for Bio::SearchIO. The Bio::SearchIO is just the front-end to this process, in fact the processing of these reports is done by different modules in the Bio/SearchIO directory. So if you look at your bioperl distribution at the modules in Bio/SearchIO you'll see modules in there like: blast.pm, fasta.pm, blastxml.pm, SearchResultEventBuilder.pm, EventHandlerI.pm (depending on what version of the toolkit there may be more modules in there). There is also a SearchWriterI.pm and Writer directory in there but we'll save that for later. If you don't have the distribution handy you can navigate this at the bioperl CVSweb page [http://cvs.open-bio.org/cgi-bin/viewcvs/viewcvs.cgi/bioperl-live/Bio/SearchIO/?cvsroot=bioperl].

Let's use the blast.pm module as an example to describe the relationship of the modules in this dir (could have subsituted any of the other format parsers like fasta.pm or blastxml.pm - these are always lowercase for historical reasons). The module has some features you should look for - the first is the hash in the BEGIN block called %MAPPING. This key value pairs here are the shorthand for how we map events from this module to general event names. This is only necessary because if we have an XML processor (see the blastxml.pm module) the event names will be the same as the XML tag names (like <Hsp_bit-score> in the NCBI BLAST XML DTD). So to make this general we'll make sure all of the events inside our parser map to the values in the %MAPPING hash - we can call them whatever we want inside this module. Some of the events map to hash references (like Statistics_db-len) this is so we can map multiple values to the same top-level attribute field but we know they will be stored as a hash value in the subsequent object (in this example, keyed by the name 'dbentries'). The capital "RESULT", "HSP", or "HIT" in the value name allow us to encode the event state in the event so we don't have to pass in two values. It also easy for someone to quickly read the list of events and know which ones are related to Hits and which ones are related to HSPs. The listener in our architecture is the Bio::SearchIO::SearchResultEventBuilder. This object is attached as a listener through the Bio::SearchIO method add_EventListener. In fact you could have multiple event listeners and they could do different things. In our case we want to create Bio::Search objects, but an event listener could just as easily be propagating data directly into a database based on the events. The SearchResultEventBuilder takes the events thrown by the SearchIO classes and builds the appropriate Bio::Search::HSP:: object from it.

Sometimes special objects are needed that are extensions beyond what the GenericHSP or GenericHit objects are meant to represent. For this case we have implemented Bio::SearchIO::SearchResultEventBuilder so that it can use factories for creating its resulting Bio::Search objects - see Bio::SearchIO::hmmer _initialize method for an example of how this can be set.

6. Writing and formatting output

Often people want to write back out a BLAST report for users who are most comfortable with that output or if you want to visualize the context of a weakly aligned region to use human intuition to score the confidence of a putative homologue. Bio::SearchIO is for parsing in the data and Bio::SearchIO::Writer is for outputting the information. The simpliest way to output data as a pseudo-BLAST HTML format is as follows.

```
my $writerhtml = new Bio::SearchIO::Writer::HTMLResultWriter();
my $outhtml = new Bio::SearchIO(-writer => $writerhtml,
                                -file   => ">searchio.html");
# get a result from Bio::SearchIO parsing or build it up in memory
$outhtml->write_result($result);
```

If you wanted to get the output as a string rather than write it out to a file, simply use the following.

```
$writerhtml->to_string($result);
```

The HTMLResultWriter supports setting your own remote database url for the sequence links in the event you'd like to point to your own SRS or local HTTP based connection to the sequence data, simply use the remote_database_url method which accepts a sequence type as input (protein or nucleotide).

You can also override the id_parser() method to define what are the unique IDs from these sequence ids in the event you would like to use something other than accession number that is gleaned from the sequence string.

If your data is instead stored in a database you could build the Bio::Search objects up in memory directly from your database and then use the Writer object to output the data. Currently there is also a Bio::SearchIO::Writer::TextResultWriter which supports writing BLAST textfile output.

7. Extending SearchIO

The framework for Bio::SearchIO is just a starting point for parsing these reports and creating objects which represent the information. If you would like to create your own set of objects which extend the current functionality we have built the system so that it will support this. For example, if you've built your own HSP object which supports a special operation like, realign_with_sw which might realign the HSP via a Smith-Waterman algorithm pulling extra bases from the flanking sequence. You might call your module Bio::Search::HSP::RealignHSP and put it in a file called Bio/Search/HSP/RealignHSP.pm. Note that you don't have to put this file directly in the bioperl source directory - you can create your own local directory structure that is in parallel to the bioperl release src code as long as you have updated your PERL5LIB to contain your local directory or use the 'use lib' directive in your script. Also, you don't have to use the namespace Bio::Search::HSP as namespaces don't mean anything about object inheritance in perl, but we recommend you name things in a logical manner so that others might read your code and if you feel encouraged to donate your code to the project it might easily integrated with existing modules.

So, you're going to write your new special module, you do need to make sure it inherits from the base Bio::Search::HSP::HSPI object. Additionally unless you want to reimplement all the initialization state in the current Bio::Search::HSP::GenericHSP you should just plan to extend that object. You need to follow the chained constructor system that we have setup so that the arguments are properly processed. Here is a sample of what your code might look like (don't forget to write your own POD so that it will be documented, I've left it off here to keep things simple).

```
package Bio::Search::HSP::RealignHSP;
use strict;
use Bio::Search::HSP::GenericHSP;
use vars qw(@ISA); # for inheritance
@ISA = qw(Bio::Search::HSP::GenericHSP); # RealignHSP inherits from GenericHSP
```

```
   sub new {
      my ($class,@args) = @_;
      my $self = $class->SUPER::new(@args); # chained contructor

      # process the 1 additional argument this object supports
      my ($ownarg1) = $self->_rearrange([OWNARG1],@args);

      return $self; # remember to pass the object reference back out
   }

   sub realign_hsp {
      my ($self) = @_;
      # implement my special realign method here
   }
```

The above code gives you a skeleton of how to start to implement your object. To register it so that it is used when the SearchIO systems makes HSPs you just need to call a couple of functions. The code below outlines them.

```
use Bio::SearchIO;
use Bio::Search::HSP::HSPFactory;
use Bio::Search::Hit::HitFactory;

# setup the blast parser, you can do this with and SearchIO parser however
my $searchio = new Bio::SearchIO(-file => $blastfile,
                                 -format =>'blast');
# build HSP factory with a certain type of HSPs to make
# the default is Bio::Search::HSP::GenericHSP
my $hspfact = new Bio::Search::HSP::HSPFactory(-type =>
                'Bio::Search::HSP::RealignHSP');
# if you wanted to replace the Hit factory you can do this as well
# additionally there is an analagous
# Bio::Search::Result::ResultFactory for setting custom Result objects
my $hitfact = new Bio::Search::Hit::HitFactory(-type =>
                  'Bio::Search::Hit::SUPERDUPER_Hit');
$searchio->_eventHandler->register_factory('hsp', $hspfact);
$searchio->_eventHandler->register_factory('hit', $hitfact);
```

We have to register the HSPFactory which is the object which will create HSPI objects, by allowing this to be built by a factory rather than a hardcoded Bio::Search::HSP::GenericHSP->new(...) call we are permitting a user from taking advantage of the whole parsing structure and the ability to slot their own object into the process rather than reimplementing very much. We think this is very powerful and is worth the system overhead which may not permit this to be as efficient in parsing as we would like. Future work will hopefully address speed and memory issues with this parser. Volunteers and improvement code is always welcome.