NCBI BLAST Database Format

Michael S. Farrar

HHMI, Janelia Farm Research Campus

19700 Helix Drive

Ashburn VA 20147

farrarm@janelia.hhmi.org

March 31, 2010

Abstract

The NCBI BLAST programs are arguably one of the most frequently used bioinformatics programs. To use these programs, sequences must first be converted to a binary format. We are are not aware of good documentation of these formats or tools for accessing this data other than those supplied by the NCBI Toolkit. To facilitate the use of these databases in other bioinformatics programs, this paper describes the layout of these files.

Introduction

One small way BLAST achieves its speed is by using sequences in a binary format thus avoiding the overhead of parsing sequences stored in ASCII formats. The program that converts FASTA files to the binary files used by BLAST is *formatdb*. The index file, sequence file and header file are the three files needed to extract sequences from the BLAST database. For protein databases these files end with the extensions ".pin", ".psq" and ".phr" respectively. For DNA databases the extensions are ".nin", ".nsq" and ".nhr" respectively. The index file contains information about the database, i.e. version number, database type, file offsets, etc. The sequence file contains residues for each of the sequences. Finally, the header file contains the header information for each of the sequences. This document describes the structure of the NCBI BLAST database version 4 (the current version as of this writing).

The NCBI C Toolkit warns that internal structure of the BLAST databases can change with little or no notice. They recommend that the readdb API, which is part of the NCBI Toolkit, should be used to extract data from the BLAST databases.

Index File (*.pin, *.nin)

The index file contains format information about the database. The layout of the version 4 index file is below:

Version	Int32	Version number.
Database type	Int32	0-DNA 1-Protein.
Title length	Int32	Length of the title string (T).
Title	Char[T]	Title string.
Timestamp length	Int32	Length of the timestamp string (S).
Timestamp	Char[S]	Time of the database creation. The length of the timestamp S is increased to force 8 byte alignment of the remaining integer fields. The timestamp is padded with NULs to achieve this alignment.
Number of sequences	Int32	Number of sequences in the database (N).
Residue count	Int64	Total number of residues in the database. Note: Unlike other integer fields, this field is stored in little endian.
Maximum sequence	Int32	Length of the longest sequence in the database.
Header offset table	Int32[$N+1$]	Offsets into the sequence's header file (*.phr, *.nhr).
Sequence offset table	Int32[$N+1$]	Offsets into the sequence's residue file (*.psq, *.nsq).
Ambiguity offset table	Int32[$N+1$]	Offset into the sequence's residue file (*.nsq). Note: This table is only in DNA databases. If the sequence does not have any ambiguity residues, then the offset points to the beginning of the next sequence.

The integer fields are stored in big endian format, except for the residue count which is stored in little endian. The two string fields, timestamp and title are preceded by a 32 bit length field. The title string is not NUL terminated. If the end of the timestamp field does not end on an offset that is a multiple of 8 bytes, NUL characters are padded to the end of the string to bring it to a multiple of 8 bytes. This forces all the following integer fields to be aligned on a 4-byte boundary for performance reasons. The length of the

timestamp field reflects the NUL padding if any. The header offset table is a list of offsets to the beginning of each sequence's header. These are offsets into the header file (*.phr, *.nhr). The size of the header can be calculated by subtracting the offset of the next header from the current header. The sequence offset table is a list of offsets to the beginning of each sequence's residue data. These are offsets into the sequence file (*.psq, *.nsq). The size of the sequence can be calculated by subtracting the offset of the next sequence from the current sequence. Since one more offset is stored than the number of sequences in the database, no special code is needed in calculating the header size or sequence size for the last entry in the database.

Protein Sequence File (*.psq)

The sequence file contains the sequences, one after another. The sequences are in a binary format separated by a NUL byte. Each residue is encoded in eight bits.

Amino acid	Value	Amino acid	Value	Amino acid	Value	Amino acid	Value
-	0	G	7	N	13	U	24
A	1	H	8	O	26	V	19
B	2	I	9	P	14	W	20
C	3	J	27	Q	15	X	21
D	4	K	10	R	16	Y	22
E	5	L	11	S	17	Z	23
F	6	M	12	T	18	*	25

DNA Sequence File (*.nsq)

The sequence file contains the sequences, one after another. The sequences are in a binary format but unlike the protein sequence file, the sequences are not separated by a NUL byte. The sequence is first compressed using two bits per residue then followed by an ambiguity correction table if necessary. If the sequence does not have an ambiguity table, the sequence's ambiguity index points to the beginning of the next sequence.

Two-bit encoding

The sequence is encoded first using two bits per nucleotide.

Nucleotide	Value	Binary
A	0	00
C	1	01
G	2	10
T or U	3	11

Any ambiguous residues are replaced by an 'A', 'C', 'G' or 'T' in the two bit encoding. To calculate the number of residues in the sequence, the least significant two bits in the last byte of the sequence needs to be examined. These last two bits indicate how many residues, if any, are encoded in the most significant bits of the last byte.

For the following encoded sequence, `6b148681`, the first three bytes encode 12 residues. Examining the least two significant bits of the last byte, `81`, the last byte encodes a single residue for a total length of 13 residues. The expanded DNA string is `CGGTACCAGACGG`.

Ambiguity Table

To correct a sequence containing any degenerate residues, an ambiguity table follows the two bit encoded string. The beginning of the ambiguity table is pointed to by the ambiguity table index in the index file, "*.nin". The first four bytes contains the count of 32 bit words in the correction table. Each correction contains three pieces of information, the actual encoded nucleotide, how many nucleotides in the sequence are replaced by the correct nucleotide and finally the offset into the sequences to apply the correction.

Nucleotide		Value	Binary	Nucleotide		Value	Binary
-		0	0000	T		8	1000
A		1	0001	W	$(A\|T)$	9	1001
C		2	0010	Y	$(C\|T)$	10	1010
M	$(A\|C)$	3	0011	H	$(A\|C\|T)$	11	1011
G		4	0100	K	$(G\|T)$	12	1100
R	$(A\|G)$	5	0101	D	$(A\|G\|T)$	13	1101
S	$(C\|G)$	6	0110	B	$(C\|G\|T)$	14	1110
V	$(A\|C\|G)$	7	0111	N	$(A\|C\|G\|T)$	15	1111

For 32 bit entries, the first 4 most significant bits encodes the nucleotide. The bit pattern of the ambiguous symbol is equal to bit pattern of the nucleotides or'ed together, i.e. the value of 'H' is equal to ('A' |'T' |'C'). The next field is the repeat count which is four bits wide. One is added to the count giving it the range of $1 - 16$. The last 24 bits is the offset into the sequence where the replacement starts. The first residue start at offset zero, the second at offset one, etc. With a 24 bit size, the offset can only address sequences around 16 million residues long.

To address sequences with more than 16 million residues, 64 bit entries are used. The most significant bit of the table count, the first four bytes of the ambiguity table, is set when 64 bit entries are used. The count is still the number of 32 bit words in the table, so the number of 64 bit entries is the count divided by two. Going from 32 bits to 64 bits, just the size of the fields has changed.. The nucleotide remains the first four bits. This is followed by the repeat count which has been increased to 12 bits giving it the range of $1 - 4096$. This is followed by the offset which is now 48 bits.

Using the following ambiguity string `00000002 32000005 70000009` as an example, the first 32 bits indicates the table has two corrections. The first correction, `32000005`, encodes the symbol 'M' replacing the next three (2+1) nucleotides starting at offset 5. The next correction, `70000009`, encodes the symbol 'V' replacing one nucleotide (0+1) at offset 9. Applying these corrections to the sequence above, it becomes `CGGTAMMMGVCGG`.

Header File (*.phr, *.nhr)

The header file contains the headers for each sequence, one after another. The sequences are in a binary encoded ASN.1 format. The size of a header can be calculated by subtracting the offset of the next sequence from the current sequence offset. The ASN.1 definition for the headers can be found in the NCBI Toolkit in the following files: asn.all and fastadl.asn.

The parsing of the header can be done with a simple recursive descent parser. The five basic types defined in the header are:

- Integer – a variable length integer value.

- VisibleString – a variable length string.

- Choice – a union of one or more alternatives.

- Sequence – an ordered collection of one or more types.

- SequenceOf – an ordered collection of zero or more occurrences of a given type.

Integer

The first byte of an encoded integer is a hex `02`. The next byte is the number of bytes used to encode the integer value. The remaining bytes are the actual value. The value is encoded most significant byte first. The following hex string, `02020123`, encodes the decimal value 291.

VisibleString

The first byte of a visible string is a hex `1A`. The next byte starts encoding the length of the string. If the most significant bit is off, then the lower seven bits encode the length of the string, i.e. the string has a length less than 128. If the most significant bit is on, then the lower seven bits is the number of bytes that hold the length of the string, then the bytes encoding the string length, most significant bytes first. Following the length are the actual string characters. The strings are not NUL terminated. The following string "HHMI" would be encoded as `1A0448484D49`. A string "HHMI" repeated 100 times, would be

encoded as `1A82019048484D49 48484D4948484D49` ... The hex `82` with the most significant bit on uses the next two bytes `0190` to encode the length of the string, 400. This is then followed by the string.

Choice

The first byte indicates which selection of the choice. The choices start with a hex value `A0` for the first item, `A1` for the second, etc. The selection is followed by a hex `80`. Two NUL bytes follow the choice. For example, an *Object-id*[1] consisting of the second item where *str* is set to "HHMI" would be encoded as `A1801A0448484D49 0000`. The first two bytes `A180` codes for the second item, *str*, is the choice followed by the string "HHMI" followed by `0000` indicating the end of the choice.

Sequence

The first two bytes are a hex `3080`. The header is then followed by the encoded sequence types. The first two bytes indicates which type of the sequence is encoded. This index starts with the hex value `A080` for the first item, `A180` for the second, etc. then followed by the encoded item and finally two NUL bytes, `0000`, to indicate the end of that type. The next type in the sequence is then encoded. If an item is optional and is not defined, then none of it is encoded including the index and NUL bytes. This is repeated until the entire sequence has been encoded. Two NUL bytes then mark the end of the sequence. For example, if the "Fall" of 2010 were encoded to the *Date-std*[2] format it would be `3080A080020207DA 0000A3801A044661` `6C6C00000000`. The sequence starts with two bytes `3080`. The first type encoded is `A080` is for the type *year*. This is followed by *year's* type and value, `020207DA`, which is an integer encoded `2010`, ending with `0000`. The *season* is encoded next. First is *season's* type `A380` followed by the string "Fall", `1A0446616C6` , ending with `0000`. Since there are no values for *month* and *day* (since they are optional) no types are encoded for their values. The final `0000` encodes the end of the sequence.

SequenceOf

The first two bytes are a hex `3080`. Then the lists of objects are encoded. Two NUL bytes encode the end of the list. For example, a sequence of two integers 16 and 32 would be encoded as `3080020110020120`

---

[1]See the *Header Definition* section for the ASN.1 definition.
[2]See the *Header Definition* section for the ASN.1 definition.

`0000`. The first two bytes `3080` codes for the start of the list, then the two integers `020110` and `020120` and finally `0000` indicating the end of the list.

Header Definition

The following is the ASN.1 BLAST header definition. This information was found in two files, asn.all and fastadl.asn, in the NCBI C Toolkit.

```
Blast-def-line-set ::= SEQUENCE OF Blast-def-line  -- all deflines for an entry

Blast-def-line ::= SEQUENCE {
    title VisibleString OPTIONAL,          -- simple title
    seqid SEQUENCE OF Seq-id OPTIONAL,     -- Regular NCBI Seq-Id
    taxid INTEGER OPTIONAL,                -- taxonomy id
    memberships SEQUENCE OF INTEGER OPTIONAL, -- bit arrays
    links SEQUENCE OF INTEGER OPTIONAL,    -- bit arrays
    other-info SEQUENCE OF INTEGER OPTIONAL } -- future use

Seq-id ::= CHOICE {
    local Object-id,              -- local use
    gibbsq INTEGER,               -- Geninfo backbone seqid
    gibbmt INTEGER,               -- Geninfo backbone moltype
    giim Giimport-id,             -- Geninfo import id
    genbank Textseq-id,
    embl Textseq-id,
    pir Textseq-id,
    swissprot Textseq-id,
    patent Patent-seq-id,
    other Textseq-id,             -- for historical reasons, 'other' = 'refseq'
    general Dbtag,                -- for other databases
    gi INTEGER,                   -- GenInfo Integrated Database
    ddbj Textseq-id,              -- DDBJ
    prf Textseq-id,               -- PRF SEQDB
    pdb PDB-seq-id,               -- PDB sequence
    tpg Textseq-id,               -- Third Party Annot/Seq Genbank
    tpe Textseq-id,               -- Third Party Annot/Seq EMBL
    tpd Textseq-id,               -- Third Party Annot/Seq DDBJ
    gpipe Textseq-id,             -- Internal NCBI genome pipeline processing ID
    named-annot-track Textseq-id } -- Internal named annotation tracking ID

Dbtag ::= SEQUENCE {
    db VisibleString,     -- name of database or system
    tag Object-id }       -- appropriate tag

-- Object-id can tag or name anything
Object-id ::= CHOICE {
    id INTEGER,
    str VisibleString }

Patent-seq-id ::= SEQUENCE {
    seqid INTEGER,        -- number of sequence in patent
    cit Id-pat }          -- patent citation
```

```
Textseq-id ::= SEQUENCE {
    name VisibleString OPTIONAL,
    accession VisibleString OPTIONAL,
    release VisibleString OPTIONAL,
    version INTEGER OPTIONAL }

Giimport-id ::= SEQUENCE {
    id INTEGER,                       -- the id to use here
    db VisibleString OPTIONAL,        -- dbase used in
    release VisibleString OPTIONAL } -- the release

PDB-seq-id ::= SEQUENCE {
    mol PDB-mol-id,           -- the molecule name
    chain INTEGER DEFAULT 32, -- a single ASCII character, chain id
    rel Date OPTIONAL }       -- release date, month and year

PDB-mol-id ::= VisibleString   -- name of mol, 4 chars

Id-pat ::= SEQUENCE {                     -- just to identify a patent
    country VisibleString,                -- Patent Document Country
    id CHOICE {
        number VisibleString,             -- Patent Document Number
        app-number VisibleString },       -- Patent Doc Appl Number
    doc-type VisibleString OPTIONAL }  -- Patent Doc Type

Date ::= CHOICE {
    str VisibleString,        -- for those unparsed dates
    std Date-std }            -- use this if you can

Date-std ::= SEQUENCE {                -- NOTE: this is NOT a unix tm struct
    year INTEGER,                      -- full year (including 1900)
    month INTEGER OPTIONAL,            -- month (1-12)
    day INTEGER OPTIONAL,              -- day of month (1-31)
    season VisibleString OPTIONAL,     -- for "spring", "may-june", etc
    hour INTEGER OPTIONAL,             -- hour of day (0-23)
    minute INTEGER OPTIONAL,           -- minute of hour (0-59)
    second INTEGER OPTIONAL }          -- second of minute (0-59)
```