Using trees to search a blocks database
Bioinformatics spring 2000, University of Iceland

Logi Viarsson

September 21, 2001
Contents

1 The mission 2

2 Basic idea 2

3 Design 3
   3.1 The DataTree class ......................................................... 4
   3.2 The PxonList class ........................................................ 5
      3.2.1 The pxon file format ............................................... 6
   3.3 BLOCKSLinker .............................................................. 6
      3.3.1 The blocks database format ..................................... 7

4 Implementation and tools used 7
   4.1 The main.cpp file ......................................................... 7

5 The time it takes 8

6 And then what we got out of this stuff. 9

7 Where we go from now 10

8 To probe further 11

A Appendix, search results from BLOCKS search 12

B Appendix, the CDROM 14
   B.1 Installation ............................................................. 14

C Appendix, extract from my diary 15
1 The mission

As sequencing power has increased dramatically over the past years the number of of protein
sequences that have been sequenced is increasing dramatically. As number of sequences grows
people are bound to start wondering if there is anything in common between all those sequences.
A BLOCKs database is an effort to put together a collection of ”well conserved” regions in the
protein code, i.e. common ones. A possible usage might be using the database to find exons, the
key parts of the genetic material that make up proteins.

There are several programs available to search the BLOCKs database but none was found to
be sufficiently fast, in addition we wanted to try out using trees to search huge databases. We used
a Blocks+ Database, created on the 23. of January 2000 by Fred Hutchinson Cancer Research
Center 1100 Fairview AV N, Seattle, WA 98109, USA. Many thanks for creating the database.

That database alone contains more than 250,000 sequences so a fast search algorithm is vital
in order to search efficiently for matches.

2 Basic idea

We want to create a tree where each node is an amino acid. The next amino acid in a given sequence
is then one of the children of the node. For example the sequence ASB would be represented like
in figure 1,

![Figure 1: Tree with sequence ASB](image)

Another example is a tree containing the sequences ASB and ASD. It would be represented like
figure 2 shows. Note that we also need to mark those nodes containing the last amino acid in a
sequence. This makes sure that we can find sequences like ART and ARTA separately in our tree.

![Figure 2: Tree with sequence ASB and ASD](image)

Searching for a given amino acid sequence in the tree is then very easy. All we need to do is to
walk down the tree using the given amino acid sequence as a guide. We then simply check to see
if the ”guide” (the target amino acid sequence) led us to a node containing the stop signal.
For example searching for the sequence $ASB$ in figure 2 starts by seeing if the header node is $A$. Now since the header node is $A$, we see if there is a path to node containing an $S$. We find such a node and move to it. Next we see if this node has a child containing $B$, which we find and there we see that that is the end of the query sequence and also the input sequence. Therefore we found the sequence $ASB$ in the database.

**Why is this a good idea?**

This is a VERY fast way to search for sequences in a large database, since it is independent of the database size. Also for sequences that branch off quickly it is even faster to discard those sequences. For example referring to the tree in figure 2 if we were searching for the sequence $BSB$ we would only need ONE comparison before we would be sure that the sequence is not in the database.

Since most of the DNA code is "garbage", i.e. introns, this is very useful. So we can be extremely quick in searching for a random protein sequence. On the average it would only take about 4.1707 trials to find out that the sequence is not in the database, see in figure 3 for reference.

![Comparison of database size and possible combinations of amino acids](image)

**Figure 3: Comparison of the database size and random combinations**

**Why might this be a bad idea?**

One shortcoming is that you need an EXACT match for your sequence if you want to find it in the tree. But sometimes nature does not make perfect copies and mutants that unfortunately are of interest to the biologist are found. However in upcoming versions one might implement some word system into the algorithm. That is define a fixed word length and chop all the BLOCKs up into fixed words. Then use a tree to search the word based database but on the same time keep a copy in an indexed table. Now that the words give us close matches we need a Smith & Waterman like algorithm to expand the hits found in the tree and evaluate the similarity between the input sequence and the database BLOCK.

3 Design

Lets start by setting forth some specification for our program.
• Memory usage is practically '+' unlimited
• It should be able to find exact matches only in the database.
• It treats all the blocks in the database equally, although some blocks are surely more important than others.
• Should use all reading frames.
• Reports matches in each reading frame in Nucleic acid index relevant to the reference sequence.

To try to do disciplined software development and on the same time not lose focus we use an iterative design. In this publication though all the design is put forth in one section for the benefit of the reader.

We will use three classes in the design one for input output data that for some reasons got the name PexonList one for the Tree itself that has been named Data Tree and finally one to interface with the BLOCKs database that is called BLOCKSLinker. In addition we use a struct for the elements of the tree. A class diagram is found in figure 4.

![Class Diagram](image)

**Figure 4: Class diagram for the project**

### 3.1 The DataTree class

We need to design a tree class that stores our database. It is done via a classic tree design with one exception. Usually when we are designing a B-tree we store all the children in a random "first come first serve" data structure (i.e. linked list). But that is stupid here since we know that any given node will have at most 20 children (because the number of possible amino acids following a given amino acid is at most 20). So instead of storing the children in a linked list or a random array we store them in a vector where each amino acid has a special seat, as illustrated in figure 5.
This saves a considerable amount of time since we know instantly if a given node in the tree has a given child. The cost is however an increase in memory usage. Since the design specifications allow unlimited memory usage we don’t give a dam.

We code the interface in two main functions, AddPath and CheckPath. The AddPath function is used to create a path in the tree and the CheckPath function to compare a given sequence to the tree. The CheckPath function takes as input a sequence of Amino acids and returns an integer. The absolute of that integer depends on how deep that input amino acid sequence got in the tree (how many of the first amino acids match to sequences in the database). If the sequence is a match then a positive number is returned otherwise a negative number is returned.

This might seem a bit odd but it is in fact a clever way to answer two questions with one answer. Did the sequence match the database? And how well does the sequence match the database?

The AddPath function works in a similar way. We take as input a given amino acid sequence, then we compare the sequence to the tree. Sooner or later we ”break off” from the tree. Then it is time to add a new branch to the tree so that the input sequence will be found if searched for. This is illustrated in section 1 in figures 1 and 2.

3.2 The PexonList class

This forms the link between the sequences that are to be compared to our database, stores information on the sequence we are working with, and a convenient way to write our results to an output file. These key functions form the interface of this IO class, they are.

- PexonList(input, output) the constructor that initializes the IO streams in and out of the interface.
- WriteCurrentPexon() writes all known info on the current pexon that have been saved with the AddInterestingSite function
- AddInterestingSite(start, stop) marks the interesting site starting at start and stopping at stop, the information saved this way will be written to the output file next time the user calls the WriteCurrentPexon function.

Figure 5: How the tree in figure 2 would look like in the DataTree class
• GetNameOfPexon() returns the name of the current Nucleic acid sequence, if the user has not specified a name the default Jhondoe is used. Concated to the name is frame information where F means forward and R reverse followed by a number representing the frame. F.ex. if we were in the reverse frame starting one base from the end it would be represented like R 1.

Although you can construct the interface without any parameters it is NOT recommended if you intend to use the IO routines. However you can still use the following translation routine without any trouble.

• WhichAminoAcid(base1, base2, base3) takes as input tree bases as standard character format one of the four letters A,G,C,T representing the corresponding nucleic acid. The function performs lookups in a table and finds the corresponding amino acid, which is returned in a character format. The table was adapted from The cartoon guide to genetics.

3.2.1 The pexon file format

Consists of FASTA-like format with one exception. The sequence is compressed into ONE line and must be so otherwise it is represented as multiple sequences. All empty lines could also cause you problems. A general format is as follows.

> Name1
Sequence1 in FASTA format
> Name2
Sequence2 in FASTA format
...
> NameN
SequenceN in FASTA format

The Sequence code need to comply with the FASTA code as in the following table.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>adenosine</td>
</tr>
<tr>
<td>C</td>
<td>cytidine</td>
</tr>
<tr>
<td>G</td>
<td>guanine</td>
</tr>
<tr>
<td>T</td>
<td>thymidine</td>
</tr>
</tbody>
</table>

Small caps are also accepted by the program, however all other codes cause the stop/any amino acid code to be used for any three letters containing the unknown letter. F.ex. the any code N would cause the ANT triplet to be read as the * amino acid.

3.3 BLOCKSLinker

This class forms the bridge to the Blocks database. This class is mainly used in the initialization of the program since the database is all read into memory. That means that speed is not such an important factor since this is done only once for unlimited number of searches. The key functions that form the interface are

• BLOCKSLinker(filename) initializes the class, "filename" being a valid BLOCKS database on a format specified below.

• TranslateToData(string) translate the given amino acid character string to the internal amino acid "class". Note that since the int class is in fact all we need we simply use an alias for the AminoAcid tag. Future implementations might contain a AminoAcid class...
• GetNextBlock() Returns the internal code for the next block in the database, if all blocks have been read it returns the NULL pointer.

3.3.1 The blocks database format

The blocks database is simply a text file containing all the blocks. A description that was used is entirely reverse engineered since no official data was available. Each BLOCK has exactly one line in the text file. The line contains the name for the block along with some statistical information that is unknown to us and since we treat all blocks equal they are irrelevant. Then there is the amino acid code, our main interest. For the BLOCK to be recognized as a valid BLOCK line (the line is ignored otherwise) it need to be in the following format.

NAMECODE (NUMBERS ) AMINOACIDCODE NUMBER

Where the AMINOACIDCODE is in the standard FASTA format for amino acids code (see the blast home page http://www.ncbi.nlm.nih.gov/BLAST/fasta.html).

4 Implementation and tools used

C/C++ was chosen for implementation since the author is most experienced in that language. Also this is one of the few languages that supports pointers since we will be working with big data structures it is wise to use pointers instead of copying the data directly.

The classes each got their two standard files that is the *.h file and the *.cpp file and then there is a main.cpp file that stores the main file. Since the author of this publication is strongly against software that comes from a certain firm on the east cost of the US the KDEV ELOP environment was preferred. It colors the code and has an excellent class interface along with a comfortable documentation library that does not require two CD’s.

4.1 The main.cpp file

This file contains two functions. And as the name implies it is the "glue" that holds all the other classes together.

• ShiftPath(AminoAcid *Path,int HowMuch) shifts a amino acid string to the left by HowMuch letters.

• main() where it all comes together.

Indeed the main function is where it all comes together. It contains instances of all the classes that are used. We make a loop and go through it until we get the NULL pointer from the GetNextBLOCK routine or the MAXNOOF BLOCKS has been reached. This second restriction was put into the program for debugging purposes allowing users to put an upper limit on the database size in memory. This does however cripple the program since only a subset of the BLOCKS database is used. A value of 300,000 or greater will cause all the database to be used. ¹

Now the BLOCKS linker has done what it was meant to. And we go on to the searcher routine. We loop through until the PexonList ² tells us that we have read all sequences in the list. However

¹ Using about 390 Megabytes of memory
² pexon is a short for predicted exons
since the nucleic acid that we take into the program has six possible representation as a amino acid code we sixfold each sequence as we translate it to an amino acid sequence. This is done every sixth time the GetNextPeXon() routine is executed.

Then we get the check the peXon candidate and, if it passes our checks we process it. The processing starts by converting it to a AminoAcid vector the tag that is used internally in our program. We treat each amino acid in the string as a possible starting location for a BLOCK. We take the subsequence, that starts at our current location in the string and ends at the end of the string, as a possible BLOCK we then walk down the database tree until we find a mismatch or a STOP code in the case we find a STOP code we have found a match and log it with the AddInterestingSite command. If we find a mismatch we have not found a match and continue to the next letter in the sequence and repeat the processing. This process is then repeated on all letters in the amino acid sequence. This is summarized in figure 6 where a flowchart is drawn for the process.

5 The time it takes

Let's look more closely into the running time. We start by putting a random tree into memory that takes a constant time. We then take the input sequence and translate it into AminoAcid code this is done six times since there are six different reading frames. This work takes linear time. Next we search our BLOCKS database tree for a match. Assuming we are very unlucky, we always find a mismatch in the last letter of the amino acid code. The time function for this phase is then quadratic giving the running time.

\[ T = t_{\text{start}} + t_{\text{processing}} = t_{\text{start}} + aN^2 + LowerOrderTerms \]

This means that the worst case is an \( O(N^2) \) time function. But that is a bit unfair since as we recall from the introduction (figure 3) we rarely need to dig deep into the tree. As the sequences get longer, hence there are more test cases, we expect the time function to become linear since most of the DNA code is random we need rarely to go down deeper than 4 letters to find no matches. But if we are finding matches that time might be sufficiently larger, however about 90% of the human DNA is "garbage" (introns).
That gives us a mix of a linear and a quadratic algorithm that is Dependant upon the test sequence length N. Or mathematically,

\[ T = t_{\text{start}} + t_{\text{processing}} = t_{\text{start}} + a_w(N)N^2 + b_w(N)N \]

Where the \( a_w(N) \) and \( b_w(N) \) are weight functions that decide how much value the quadratic and the linear function have. Let's see what really happens. We time our algorithm and summarize the results on figure 7.

![Time function for some cases](image)

Figure 7: Time function for the algorithm, real results

As one can see in the figure for \( N \) in the range \( 1,000 \leq N \leq 20,000 \) we have like expected a mix of linear and quadratic time function. For the lower values it is more quadratic but as \( N \) goes higher we see the time function becoming more linear.

6 And then what we got out of this stuff.

As mentioned above the BLOCKS database is a collection of common motives in proteins. If we find a lot of matches in a BLOCKS database I can be pretty sure that I have an exon. But the eucariot gene system is very sensitive, f.ex. if we insert only ONE nucleic acid into the sequence at the begining, the protein translation is sure to give odd results since we have shifted all the nucleici acids a seat back but we are still refering to the begining of the sequence when we translate to amino acids. This is refered to destroying the reading frame.

Let's make a test case on a sequence given to us by Daniel skarssson our instructor. This sequence was compared to the BLOCKS database and hits were found at the following locations in the Forward 0 reading frame (that is a direct translation of the sequence).

<table>
<thead>
<tr>
<th>BLOCK found</th>
<th>Location in nucleici acid code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FHVAPC**QLSLVLLFSSGIALWSLAVISWERWVVVCKP</td>
<td>1806 to 1926</td>
</tr>
<tr>
<td>CFPLAVILLCY</td>
<td>2742 to 2778</td>
</tr>
<tr>
<td>KAEKEVSRMVVMIIAYCFCWGPYTVF</td>
<td>3594 to 3705</td>
</tr>
<tr>
<td>ATIYNPHYVFMRQFR</td>
<td>3753 to 3804</td>
</tr>
</tbody>
</table>

Daniel skarsson had already found these exons at the following location.
As we can see from the tables our BLOCKs do not necessarily occur in an exon, the first exon for example is not to be found in the BLOCKS database and only the exact matches in the BLOCKS database are within an exon as a matter of fact they are in the same exon. This is perhaps because the genetic code was taken from RED-SENSITIVE OPSIN (RED CONE PHOTORECEPTOR PIGMENT) which is a highly specialized protein. Perhaps we would have had better luck if something more commonly found in nature would have been used. Neither the less this result tells us that not all the code for exons in nature is a like. Before we pass our judgement however lets review another case. Lets see what happens where we enter a protein into the BLOCKS database. We use the HUMAN BAND 3 ANION TRANSPORT PROTEIN borrowed from swissprot which is one of the proteins in the Hemoglobin complex.

When running against BLOCKS we find a LOT of matches which is a good sign. However there are also lot of holes, but we could tell with sufficient sureness that this is a protein from our results. The system was therefore able to recognize a protein with low error rate but it was amazing to see how many holes were in the results. (The full report from the BLOCKS searcher is in appendix A).

It is my belief that a BLOCKS search is not good for finding exons, since it has such a hard time handling introns that might come in the middle of a BLOCK hence destroying our hopes of finding it. A wiser aproach would be to use the BLOCKS database for analysis on proteins and to link them to a protein family and study its origin, its function. We might thong be able to lower the acceptance threshold although that might flood us with false positives.

7 Where we go from now

The next person working on BLOCKS will probably try to lower the acceptance threshold. Then some way of making a word like structure could be the key to fast functionality. We would then have to store each BLOCK twice, in a tree with fixed maximum depth and accurately in a indexed table. A given index might then be attached in the tree so a given word would correspond to a very limited number of BLOCKS to search from. We would then use a slower but more sensitive algorithm like Smith-Waterman to expand our word hits.

After this an expanded score would be evaluated based on how well the input sequence matches to a given BLOCK. An adjustable threshhold could then be used to select which matches are good enough. The graph on figure 3 tells us that to push the random hits out of the equation a minimum of 5 letters must be set. Currently this minimum is much higher an EXACT match is required, setting the threshold at least 7 letters to as high as 20 letters!

The molecular biologist might be interested in a hit on a BLOCKS database in an intron. It tells the scientist that this intron was once a part of a protein (he might even get a good idea on what family). Or even better if this intron is very well conserved our scientist might even want to reconsider on what is called an intron and what is an exon.

In our program we also note that all BLOCKS are treated equally, this might not be such a good idea since statistical information is readily available. In the future this might be improved greatly by implementing a better BLOCKS structure that might be added to those elements that contain
an end of a BLOCKS word bad hits on a solid BLOCK would then always been logged but not necessarily a good hit on a bad BLOCK.

8 To probe further

1. An excellent BLOCKS site can be found at Fred Hutchinson Cancer Research Center in Seattle, Washington, USA. (http://blocks.fhcrc.org/) All my thanks for making it.

2. Site full of classic sequence analysis site at Brigham young university (http://dnasc.byu.edu/seqtools.htm).

3. BLAST is always a good idea for reference (http://www.ncbi.nlm.nih.gov/blast/blast.cgi?Jform=0)

4. KDEVELOP an excellent development environment designed for the X-11 window system (http://www.kdevelop.org).


6. Larry Golnick and Mark Wheelis have written an excellent guide to genetics the Carton guide to genetics is simple and illustrative for the untrained mind.
A  Appendix, search results from BLOCKS search

>BL00219  Anion exchangers family proteins.
  Length = 919

Score = 1268 bits (3246), Expect = 0.0
Identities = 538/922 (58%), Positives = 632/922 (68%), Gaps = 70/922 (7%)

Query: 57  VYVELQELVMDEKQELRWMEEARWVLQENLGENAGGCRPHLSHITFWSLLELRVFT 115
         V+VEL EL+D KNQE +WME AR++ EENL E+GA WG PH++ L+F SLLELRR
Sbjct: 1  VFVELNELLLD-KNQEPQWETARWKFEENLEEGA+WGPVH+ASLSRSLELRRTLA 59

Query: 116  KGTVLDDQTS_LAVGANNLDLRVIFIPDFQDREELLRALLKHSH-------- 163
          VLLDL + +L GVA+Q+++ + DQI+ +DR +LRRALLKHSH
Sbjct: 60  HGAVLDDQQTLPVGAVQVVMVISDQIKAEDRNAVLRAALLKHSHPSDEKDFSFRN 119

Query: 164  __--AGELEALGKVPAVLTRS--GDPSQPLLQPQHSSHETQFLFCEQGDGTEHSPSGL--- 216
          AG L +L G S ++PL+ ET+L E+ P+GI
Sbjct: 120  ISAGSGLGSLVHHHGQAEDPHVTEPLIG--GIPETRLDVEREDVPPSAPAGITRSK 177

Query: 217  --------LEKIPPDSEATLVLVGRADFLEQPVLGFVRLQ--XXXXXQXXXIRFLFLV 268
          LEKIP D+EA+T+VLVG DFLEQP L VFRL +RFLFLV
Sbjct: 178  SKHELKLLEKIPDAEATLVLVGCDFLEQPALAFVRLGAAPOLVEAPPPVRFLFLV 237

Query: 269  LGPEAPHDITYTQLGRAAATLMSERVFRIDAYMAQSRGELLHSLGRFLDCSLVLPPTDAPS 328
          LGP +PH+DY +LGRA ATLM+R VRFAYA+ L R ELL+++ FDCLS+VLPT+P
Sbjct: 238  LGPSSPHVDYHELGRAIDTMLSDRVIDHEAAYLADDREELLNAINSFLDCLSIVLPTEVP 297

Query: 329  EQALLSLVPVQRELLRRRQSSPAK--------DSSFYKGLDLNGGP-DDPLQQTG 376
          E+ L S+V QR++L++R + P D + ++ G DDPL++TG
Sbjct: 298  EELLRSIVHFQROMKLREEQGRLLPPGLGEPKSAQDKALLQMVEAGAAEDDPLRRTG 357

Query: 377  QLFGGLVRDIRRPPYYLSIHDITDAFSPQVQLAADVIFIYALSAAITPGGGLGEGKTRNMGM 436
          + FGGL+R+RRYP+YLSD DA P Q LAVIFYIALSAPITPGGLGEGKT + +G
Sbjct: 358  RPFGGGLRDIRRPPYYLSDFDRDLPQPCLAAVIFYIALSAPITPGGGLGEGKTEDLIG 417

Query: 437  VSELLISTAVQGILFALLGAAQPLVVGFSGPLLVFEEAFFSFETNGLEYIVGRWIGFW 496
          VSEL++STA+QG++FLGAQPLVVVFEAFFSFCE+N LEY+VGRWIGFW
Sbjct: 418  VS ELMSTALQGIVFCLLLQAPQPLLIFSSPVLLFEAFFSFCESDNLEYLVGRWIGFW 477

Query: 497  LILLVLVVFAPAGSFVLVFIRSTQETXXXHXXHXYETFSLIKIFQDHPLQ-------- 550
          L+LL +L+EA GFSFLVR+ISR+TQEO YET KLI KIF+HPL
Sbjct: 478  LVLLALLMALEGFSFLVIRYISRTQEIFAFLSLIFIFIETFYKLIKIFQEHPLHGCVSNS 537

Query: 551  --------KTYNYNLVMVPKQPQPL------------------PNTALLSLVMLAGTFFFAMMLR 589
          + N GP PNTALLSLVMLAGTFFF A LR
Sbjct: 538 SSETDSSENNATWAGAGSTLGAPNRRSSAGQAQGRPRGQPNTALLSLVLMAGTFIAPFLR 597
Query: 590 KFKNSYFPGKLRRVIGDFGVPISILMLVDFIIQDITYTQKLSPDGFKVSNSSARGWV 649
KFKNS +FPG+RRVIGDFGVPISILMLVLD+ I+DTYTKLSVP G V+N RGW
Sbjct: 598 KFKNSRFFPGKIRRVIGDFGVPISILMLVDYTEDTYTQKLSPGSLVTPNPDARKWG 657
Query: 650 IPHLGLRSEFIWMMFASALPPALLVFILIFLESQTTLIVSKPERMKVSGFGHLDXXXX 709
IPHLG + FF+WMM ASALPALLVFILIF+E+QITTIVSK ERK+VKGSGFHL
Sbjct: 658 IPHLGEKRPPFVWWMVASALPALLVFILIFMETQITTLIVSKKRKLVKGSGFHLDDLII 717
Query: 710 XXXXXXXXXFPWSATTVRSTVTHANALTVMKASTPAGAAQIQVEKQXXXXXXX 769
PWLSA TVRSVTHANALTVMK PGA QIQVEKQR
Sbjct: 718 VAMGGICALFGLPWLSATVRSVTHANALTVMKAVAPDGKPQIQVEKQRITGLLVAVL 777
Query: 770 XXXXXXXMEPIIRPLAVLFIFLYMGVTSDLGQLFDRILLLFKPPKYHDPYVKVRK 829
M PIL RIPLAVLFIIFLYMGVTDSGQL+DR+LLL PPK+HPDPYVKVRK
Sbjct: 778 VGSLILMGPIIRPLAVLFIFLYMGVTSDLGQLFYDRLLLLMPKHPDPYVKVRK 837
Query: 830 TWRMLHFTGQIICLAVLWVKSTPALPFVXXXXXXXFRNVELQCLDAD 889
TWRMLHT IQ+CLA+LVWVKSTASLAPFV F + ELQCLDAD
Sbjct: 838 TWRMLHTAIQQILLCLALLWVKSTASLAPFPVLTILTVLLCRLLPRIFTDRELQCLDAD 897
Query: 890 DAKATFDEEEGRDEYDEVMPV 911
DA+ TFDEEG DEYDE+ MPV
Sbjct: 898 DAEPTFDEEEGEDEYDEMPMPV 919
B Appendix, the CDROM

The cdrom enclosed with this document contains the following files

<table>
<thead>
<tr>
<th>INSTALLATION</th>
<th>Contains documentation on how to install the software on the CDROM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>blldb</td>
<td>Directory containing the BLOCKS database</td>
</tr>
<tr>
<td>blocks.dat</td>
<td>The database itself</td>
</tr>
<tr>
<td>blockslinker.cpp</td>
<td>Implementation file for the BLOCKSLinker class</td>
</tr>
<tr>
<td>blockslinker.h</td>
<td>Declaration file for the BLOCKSLinker class</td>
</tr>
<tr>
<td>datatree.cpp</td>
<td>Implementation file for the DataTree class</td>
</tr>
<tr>
<td>datatree.h</td>
<td>Declaration file for the DataTree class</td>
</tr>
<tr>
<td>documentation.ps</td>
<td>This file</td>
</tr>
<tr>
<td>dbs300k</td>
<td>Text file containing timing data for database size set for 300,000</td>
</tr>
<tr>
<td></td>
<td>BLOCKS i.e. the full database</td>
</tr>
<tr>
<td>dbs30k</td>
<td>Text file containing timing data for database size set for 30,000</td>
</tr>
<tr>
<td></td>
<td>BLOCKS</td>
</tr>
<tr>
<td>dbs3k</td>
<td>Text file containing timing data for database size set for 3,000</td>
</tr>
<tr>
<td></td>
<td>BLOCKS</td>
</tr>
<tr>
<td>files</td>
<td>Directory containing the input query files.</td>
</tr>
<tr>
<td>PexonList</td>
<td>File containing the input sequences used in project 3.</td>
</tr>
<tr>
<td>main.cpp</td>
<td>Includes the main function.</td>
</tr>
<tr>
<td>pexionlist.cpp</td>
<td>Implementation file for the PexionList class</td>
</tr>
<tr>
<td>pexionlist.h</td>
<td>Declaration file for the PexionList class</td>
</tr>
<tr>
<td>snapper</td>
<td>Compiled program file on a Red Hat 6.0 system.</td>
</tr>
<tr>
<td>snapper.cpp</td>
<td>Pseudo make file i.e. contains just include statements.</td>
</tr>
<tr>
<td>timer.cpp</td>
<td>Main function for timing the algorithm.</td>
</tr>
</tbody>
</table>

B.1 Installation

First we need to mount the cdrom successfully. Often (in over 99% of cases) one would use

`mount /mnt/cdrom`

And then changing the current directory to the cdrom drive.

`cd /mnt/cdrom`

Then copying the contents of this CDROM INCLUDING subdirectories to a location on your hard drive e.g. if that location was `/home/logi/blocks/` I would execute the following command.

`cp -R * /home/logi/blocks/`

Most likely you need to compile the program to run it. Compile using g++ compiler but of course first changing the current directory to the site where you copied the directory by

`cd /home/logi/blocks/`

And then compiling with g++.

`g++ snapper.cpp -o snapper`

At last you can start to run this thing. Tee off by executing snapper with

`./snapper`

Enjoy . . .

---

5In case your system is not quite with 500 MEgs of RAM you can limit the memory usage, see the INSTALLATION file on the cdrom for further instructions.
C Appendix, extract from my diary

In order to cast some light into the wast amount of time put on this project I cast forth "selected" chapters from my diary.

March 4th I chose BLOCKS as my project in bioinformatics.
March 11th Internet session, gathering interesting pages
and reading material took about 3.5 hours.
March 14th First design session, found out that the original hash table
approach was uneffective, switched to a tree approach 4 hours.
March 17th Programing session modified tree build and tested, time
aprox 3 hours.
March 21th Second desing session, more designed about interface to the
BLOCKS database which was downloaded at that time. 4 hours.
March 24th Programing session, started work on the interfaces needed to
refine my design . . . 3 hours.
March 28th Reading and design session, interfaces designed properly and
a draft for a processflow. 4 hours.
March 31th Design and final implementations done for a quick test. 2 hours
April 4th IT WORKS but only after a few hours of "debuging" aprox 3 hours
April 5th FINALLY got into some HARDWARE at Decode genetics (a 1GB memory
machine :) spend 5 hours finetuning and debugging.
April 6th Time measurements done on diffrent database size's spend 3 hours
playing on biolinux.decode.is
April 10th Started on documentation and work related to returning the
project. aprox 1 hour
April 13th Documentation and CDROM creation mania taking an "allnighter"
to FINISH BLOCKS time 10 hours.
TOTAL time 45.5 hours.