DNA Sequence Analysis Software

Group: Xin Xiong, Yuan Zhang, Hongbo Liu
Supervisor: Henrik Bulskov
# Table of contents

Introduction.......................................................................................................................................2

1 Backgrounds and Motivation .........................................................................................................2
   1.1 Molecular Biology................................................................................................................2
   1.2 Computer Science ...............................................................................................................3
   1.3 Motivation..........................................................................................................................4

2 Theory ....................................................................................................................................4
   2.1 Parsing...............................................................................................................................4
   2.2 JDBC..................................................................................................................................5
   2.3 Oracle SQL loader ...........................................................................................................5

3 Analysis and Design...............................................................................................................6
   3.1 Ideal Model ............................................................................................................6
   3.2 Description of original file ...............................................................................................8
   3.3 Entity Relationship Model ............................................................................................10
   3.4 Design Class Diagram ....................................................................................................11

Implementation ...................................................................................................................................12
   1. Database creation ................................................................................................................12
   2. Parsing ...............................................................................................................................12
   3. Data loading .......................................................................................................................13
   4. Programming .....................................................................................................................14

Discussion .......................................................................................................................................16

Conclusion ......................................................................................................................................19

Acknowledgements .........................................................................................................................20

References.......................................................................................................................................20

Appendix.........................................................................................................................................21
Introduction

1 Backgrounds and Motivation

1.1 Molecular Biology

It is estimated that there are more than 10 million living species on earth (Ref.1). The scientists study the evolution of the species based on the analysis the similarities and differences for the species’ genomes. The precondition is to obtain the data of the genomes that they want to analyze. Nowadays, the data of genomes for many species is available on the websites such as NCBI (the National Center for Biotechnology Information).

A species is a group of living organisms (Ref 2). Every organism has its own unique genome (Ref 3). We mentioned previously that we can download the data of some species’ genome information from the internet. This data of each genome includes the following information, description of the organism, references to the organism, comment, feature, and DNA sequences. The main one is the DNA sequence. DNA is genetic material that represents the characteristic of the species. DNA sequence consists of a set of alphabet or base A, T, C, and G (See Fig. 1.1). The DNA sequence, to be more specific is a double strand and every alphabet on one strand is complemented by exactly one alphabet on the other strand, A by T, C by G and vice versa (See Fig. 1.2) (Ref. 3). Different DNA sequences represent the different characteristics of the species. Based on the analysis of DNA sequence, for example, searching the specific sequence in the complete DNA sequence, we can find out the similarities and differences between the different species. DNA sequence can also be translated into amino acid sequence indirectly. RNA is needed as a messenger between the DNA and amino acids. RNA is also a genetic material and RNA is produced complementarily from one of the DNA strands. The difference between RNA and DNA sequence is the U in RNA instead of T in DNA (See Fig. 1.3).

Several software tools have already been developed to access and analyze for this data. Molecular biologists may use some software tools for their research. So a good software tool will be very helpful for them. But most of them are very expensive and quite complicated to use.

T T A T C C A C A G

Fig. 1.1 DNA single strand

DNA strand is made up the base A, T, C, and G. This single strand has 10 bases. The first base is T and the last base is G.
**Fig. 1.2 DNA double strand**

DNA consists of two strands. Every base on one strand is complemented by one base on the other strand, A by T, C by G and vice versa.

**Fig. 1.3 DNA single strand with a complementary RNA strand and amino acid strand**

One of DNA strands acts as a template for synthesis of a complementary RNA strand. The difference between RNA and DNA sequence is the U in RNA instead of T in DNA. And then, RNA strand is translated into amino acid strand. 3 bases of the RNA strand code for 1 amino acid.

### 1.2 Computer Science

We are working on applying the computer science skills to the field of biology. This is the field of bioinformatics that combining computer science and biology. Our main aim is to make an application that can search the information for some species’ genomes. The information retrieval system is required to achieve it. The principle of information retrieval is to search the information by keywords and ranking of documents on estimated relevance of the documents to the query (Ref 4).

To develop the software, the first thing we need to decide is to choose the programming language. All members of our project had previous experience with Java, so we decided to use Java language. The next thing is to find out the way to develop more stable and more functional software. The original data that we download from the website is a text file. So we decided to store the original data into database, and then we can manipulate the data in the database to implement the functions. The way to export data of text file into database is using the parsing technology.
1.3 Motivation

In order to find out the similarities and differences for many bacterial species, we used a software tool which is called WinSeq\(^1\) to do this analysis. This software tool has implemented several functions that are good for molecular biology researches. But it was unstable, it always crashed. When we were using the WinSeq to search DNA sequence, we could only search DNA fragments in one genome. But sometimes biologists need to compare many different genomes and find out the similarities and differences among these genome sequences. It will be very helpful for biologists to do this kind of research work by using software, which can search DNA fragments in more than one genome at the same time. The aim of the project is to store the biological data in the database, and implement the main functions based on the database. The main function, in our application, a short DNA sequence can be searched by perfect match and some misfits in several genomes. Misfit represents the letter of target sequence is not same with the letter of original sequence. The introduction of misfit leads to a significant amount of possible combinations, which is also an interesting function to implement it.

2 Theory

2.1 Parsing

Parsing is no longer a mystic technique since the early 70’s when Aho, Knuth, Ullman, and many others developed various parsing techniques based on their theory. It is widely used in computer science, linguistics, document conversion and etc. Parsing is the process of analyzing and structuring a sequence of tokens according to a given grammar (Ref. 5). The sequence of tokens can be a computer program, a sentence, a piece of music, and a sequence of biological data in our case. Each of them has their particular grammar. In computer programs, grammar varies from different programming languages. In human languages, grammar is the rules for structuring the words. In biological data, the grammar is how all the biological information is organized and structured. The first task of our project is to parse this biological data to a list of useful information for us.

Generally, parsing can be achieved in two ways:

- **Top-down parsing** is a parsing strategy by analyzing the general structures first, and then breaks them down into smaller parts (Ref.6).
- **Bottom-up parsing**: is a parsing strategy by identifying the most fundamental elements first, and then locating the elements containing these or in a higher abstract level (Ref. 7). The operator-precedence parser is a bottom-up parser.

---

\(^1\) WinSeq is a program for sequence analysis. It was written and maintained by an investigator; Flemming G. Hansen at the Technical University of Denmark.
In our project, the biological original data is parsed into well structured information by using the top-down parsing strategy, because the general structures of the original data is well understand.

### 2.2 JDBC

A program that we want to develop is written in Java language. Java is an object-oriented language. To analyze the data in a database, we need to connect to the database from a java program. Java Database Connectivity (JDBC) is commonly used to connect a java program to a database. JDBC is a Java Application Programming Interface (JavaAPI) for executing SQL statements (Ref 8).

The following is JDBC architecture (Fig 2).

![JDBC Architecture](image)

**Fig. 2 JDBC architecture (modified the figure from Ref. 8)**

Java program calls the JDBC library. JDBC loads a driver which communicates with the database.

There are four types of JDBC drivers. The driver that we used is the ‘Thin’ driver. This driver has many advantages; it is most suitable for the web, we don’t need to install client software and so forth.

### 2.3 Oracle SQL loader

After parsing, several text files were created, such as author text file, gene text file and so on. Among them, dna_sequence text file has the largest amount of data. Thus, a method to load large amount of data is needed now. And then, we chose Oracle SQL*loader that is a primary method
for quickly loading data from external files into tables in an Oracle database (Ref. 9).

3 Analysis and Design

We want to design an ideal model for our software. This model contains several applications that are required for supporting biological research. Some of them were already implemented by WinSeq tool. The core application is to search more than one short DNA sequences at the same time in the same DNA sequence. This application was not implemented so well by WinSeq. So we would like to implement this hard task.

In the biology field, the short DNA sequence we want to search is DnaA box. DnaA box is a 9-base DNA sequence. It is for DnaA protein binding to. The perfect DnaA box is TTA/TTCCACA. We can limit the number of misfit for the input DnaA box. Misfit represents the base of target sequence is not same with the base of original sequence. We can also limit the distance between the input DnaA boxes if more than one DnaA box as input sequence (See Fig. 3.1).

![DNA Sequence Example](image)

Distance is 11 bases

Fig. 3.1 Search two DnaA boxes in the same DNA sequence

Searching two DnaA boxes in the same DNA sequence, one is 0 misfit and the other is 1 misfit, the distance between them is less than 20 bases

3.1 Ideal Model

Though there are some software tools for DNA sequences analysis, they are still lack of some functions that we think are important for biologists to do their research work. So we are going to develop a more functional software tool. Here we introduce the functions that we are going to implement.

a. View the description of the genome. Every organism has its own unique genome. We can get all information of the organism by imputing the genome number or the genome name of this organism.

b. View the DNA sequence of the genome in a full-length. All DNA sequences of the organism can be showed up when we input the genome number or the genome name of this organism.

c. View the translation sequences. The DNA sequence can be translated into amino acids. Amino acid is a molecule. A protein is made up of a set of amino acids. 3 bases code for 1 amino acid. Amino acid sequence can be showed by inputting a segment of DNA sequence.
d. Capable of editing all information about the genome. After getting the genome information, we can edit and save it.

e. Searching
   Input: DNA sequence, limited misfits, limited distance between the target sequences if the number of input sequences > 1, and the scope of the sequence for searching.
   Output: The best match sequence or a list of sequences that are ranked from best match to the worst match tolerated.

f. Primers design
   Primer is composed of several bases. It is necessary for DNA replication. When we want to amplify certain fragment of DNA, primer is required for starting. Primer is very small fragment sequence containing about 20 bases.
   Input: start base number and end base number
   Output: the sequences of two primers; forward primer and reverse primer.

g. Test primer
   Calculate the temperature melting (Tm)
   Input: primer sequence
   Output: Tm value
   If the length of primer >= 20, Tm = 81.5 + 0.41* (GC %) - 600/L
   GC % represents the percentage of GC content in the target sequence.
   L is the length of primer.

   If the length of primer < 20, Tm = 2*(A+T) + 4*(G+C)
   A+T means the amount of A and T in the primer sequence
   G+C means the amount of G and C in the primer sequence

h. Identify the position of oriC and show the information of oriC. oriC is the initiation site for DNA replication. Normally bacterial DNA are circular strands, before they replicate, they have to find out where to start. oriC is the position to start. Usually oriC contains around 200 bases. The information of oriC includes the length of oriC and sequence of oriC with marked DnaA boxes. DnaA box is a 9-base DNA sequence. It is for DnaA protein binding to.

i. Calculate GC content and AT content. The percentage of GC content and AT content in the target DNA sequence can be showed out.

j. Align two DNA sequences. Comparing two segments of DNA sequence, the similarities and differences can be showed up.

k. Reverse DNA sequence. The DNA sequence can be inverted and showed up.
### 3.2 Description of original file

The biological information, which is our original source, is a set of text files. Each of them is well structured. The text files contain four sections: Header, References, Features and Sequence (Fig. 3.2).

The top line of the header section is the locus, which contains a unique database identifier for a sequence location in the database. The identifier is followed by sequence length and molecule type (e.g., DNA or RNA). This is followed by the molecule shape, and a three-letter code for GenBank divisions. Next to the division is the record date. The following line, “DEFINITION”, provides the summary information for the sequence record. This is followed by an accession number for the sequence, which is a unique number assigned to the genome when it was first submitted to GenBank. In addition to the accession number, there is also a version number and a gene index number (GI). In the “ORGANISM” field, it includes the source of the organism with the scientific name of the species, along with the information of taxonomic classification of the organism.

The “References” section provides the publication citation related to the sequence entry. The “REFERENCE” part includes author, title, journal, PubMed and remark. The “COMMENT” part includes the contact information of the sequence submitter.

The “Features” section includes annotation information about the gene and gene product, as well as regions of biological significance reported in the sequence, with identifiers and qualifiers. The “Source” field provides the length of the sequence, the scientific name of the organism, and the taxonomy identification number. The “gene” field is the information about the base coding sequence and its name. For DNA entries, there is a “CDS” field, which is information about the boundaries of the sequence that can be translated into amino acids.

The Fourth section of the file is the sequence itself starting with the label “ORIGIN”. The sequence ends with two forward slashes (“//”) (it’s not shown in the figure).
Figure 3.2: Sample data (partial)
3.3 Entity Relationship Model

We previously mentioned about the parsing. After parsing, several different data files will be generated. And then, these files should be stored into the database. Before doing this, we need to create an Entity Relationship model which was developed to facilitate database design. It represents a real world as a set of objects. It includes three concepts, entities, relationships and attributes (Ref.4). The first step is to identify entities. In order to identify the entities, we need to know users requirements. In this case, we are developers and also users. We have several requirements, so we identified entities for these requirements.

Figure 3.3 E-R Diagram (it contains five entities and four relationships)
Genome Description entity has primary key – GenomeNo and the other four attributes- Genome Name, Genome Length, GI No and Shape.

Gene, Reference and DNA sequence are weak entities. They are partially dependent on the other entity (Genome Description) for their primary key. The primary key of Gene entity is Start, End and GenomeNo. The primary key of Reference entity is RefNo and GenomeNo. The primary key of DNA sequence entity is BaseNo and GenomeNo.

Reference entity has a multi-valued attribute-Author. We created the new entities “Author” to represent this multi-valued attribute.

These four relationships are all many to many relationship which is represented by M:N in Fig. 3.3.

We will discuss the E-R model in the discussion.
3.4 Design Class Diagram

It is time to design software when all data store into database. The computer language is Java that we chose. Java is an object-oriented language. So we decided to use an object-oriented modelling language called Unified Modelling Language (UML) to design software (Ref. 10).

In our class design diagram, every class depends on the Database Interface class, which provides the connection to the database. It is a key to the door of the database. If an exception occurred in the process of connecting to the database, it will be captured at the beginning. Each class implements Viewable interface so that the results can be viewed in different ways. In the search class, the SQL query statement is generated according to the inputs, it is executed in the database through the JDBC, and the results can be obtained from it.
Implementation

1. Database creation

According to our E-R model of the relational database, we created the database as following.

```sql
create table genome_description
    (genome_no char(15),
     gi no char(15),
     genome_name char(100),
     length int,
     shape char(10),
     primary key(genome_no));

create table dna_sequence
    (genome_no char(15),
     base_no int,
     base char(1),
     primary key(base_no),
     foreign key(genome_no) references genome_description)

create table gene
    (genome_no char(15),
     gene_name char(15),
     start_base int,
     end_base int,
     primary key(genome_no, start_base),
     foreign key(genome_no) references genome_description)

create table reference
    (genome_no char(15),
     ref_no int,
     title char(100),
     journal char(100),
     pubmed char(10),
     primary key(genome_no, ref_no),
     foreign key(genome_no) references genome_description)

create table author
    (genome_no char(15),
     ref_no int,
     author_name char(20),
     primary key(genome_no, ref_no, name),
     foreign key(genome_no, ref_no) references reference)
```

For further searching, an index is created:

```
CREATE INDEX DNA_SEQ ON "DNA_SEQUENCE" ("BASE", "GENOME_NO", "BASE_NO")
```

A B+ tree will be created inside the Oracle database, which is very efficient for the searching. The order of the attributes is compatible with the SQL statement created in “Search” class.

2. Parsing

As mentioned in the Theory, we top-down parsed the biological data into a set of relational information in several text files. “genome_description.txt” contains the genome number, the gi (genome identification) number, the genome name, the length, and the shape of the genome. “gene.txt” includes the genome number, the starting base number, the ending base number, and the gene name. “DNA_sequence.txt” contains the genome number, the base number, and the base. “reference.txt” includes the genome number, the reference number, the title, the journal, and the PubMed number. “author.txt” contains the genome number, the reference number, and the author name.
The parser reads in one original biological file at one time, and writes the above information into the five files. The scanner runs through the original file by reading from the beginning, returning tokens (strings) delimited by space, moving to the next token until the end of the file. If the returning token matches “LOCUS”, the Header part is going to be parsed. Likewise, the label “REFERENCE” represents the starting point for the Reference part, “FEATURE” appears at beginning of the Feature part, and the entire sequence follows the label “ORIGIN”.

In the Reference part, there are usually more than one author name in one reference. Therefore, the author names are stored in an ArrayList of an ArrayList ($\text{ArrayList< ArrayList<String>}>$). The length of the outer ArrayList is the same of the number of references. Some of the inner ArrayList is null, because sometimes a reference does not have author information. This is handled by constructing an ArrayList of the String type in each reference, adding to the outer ArrayList ($\text{authors.add(new ArrayList<String>()})$), and adding author names into the inner ArrayList only if the “AUTHOR” label occurs ($\text{authors.get(x).add(in.next())}$). Similarly, title, journal, and other publication information also do not appear regularly in references, which boolean flags are introduced to deal with. See full codes in appendix.

In the Feature part, the format of the starting and the ending base number of a gene varies in different original biological files. Some of the formats are like: “190..255”, “complement(1123..3322)”, “(432..565)join(654..875)”, and etc. We considered as many conditions as possible, and left others unknown. We extract the digits by split the string into several arrays of characters by the delimiter “.”. Because $\text{String.split}$ uses regular expressions and “.” have a special meaning, we wrapped “.” in [] and do $\text{location.split("[.]\[.\]")}$. In order to reduce the redundancy of the relational database design, we did not write the translation sequence into the “gene.txt” file. It could derive from the DNA sequence, and be a function of the programming. See full codes in appendix.

In the sequence part, it is relatively straightforward because it only contains the sequence information. The length of every token is fixed (10), except the number index in each line and the last token (Fig. 5.2). A local variable “a” is introduced to keep track of the number (index) of each base. It increases by 10 (the length of the token) when the scanner reads the next token ($a = a + \text{charArrat.length}$). See full codes in appendix.

### 3. Data loading

We used the SQL Loader (a tool in the Oracle database client) to load the data into the Oracle database. In the first place, we modified the parser to make the output files fulfill the format of the SQL loader and compatible with the control file. The control file determines the location of the source file and the delimiter in the file.
Then load the file into the database using the SQL Loader in the command line.

```sql
load data
INFILE 'C:\Documents and Settings\Xin\workspace\Parser\dna_sequence.txt'
truncates
INTO TABLE dna_sequence
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(genome_no, base_no, base)
```

4. Programming

JDBC is a bridge connecting JAVA and database. In JAVA, data stored in the memory, which is fast for manipulating, is flexible, and can be built in different structures. In Oracle database, data can be stored in the hard disk as a B+ tree, which is very efficient for data management (e.g. searching) (Ref. 4). The amount of the biological data is usually tremendous, and the data need to be retrieved frequently, so we decided to implement most and the core functions (searching) inside the database. In the constructor of the DBInterface, the “thin” driver is used for connecting to the Oracle database. A public method getConn provides the access to the connection of the database.

The Genome class is the center of all classes. In the database, genome number is or is part of the primary key(s) for each table. Usually the name of the genome is the input for users, so it is parameterized in the constructor of the Genome class. The information about this genome is get from single queries based on the genome name in the database, including genome number, GI number, length and shape of the genome, all genes, and all references in the genome. The classes Gene, Reference, and Author are all represent a single copy of them. A gene class contains the information about the gene name, the starting base number, and the ending base number. A Reference class includes the publication information like tile, journal, PubMed, and etc. An Author class contains the name of the authors. All these queries are implemented in SQL statement, for example, “select author_name from author where genome_no = 'genome_no' and ref_no = ref_no”.

Each class has the methods that provide the public accessing to the information above.

Search class is the core class of the application. The input of the searching is a DNA sequence which must have a length shorter than the genome, and the number of misfits tolerated. The output of the searching will be a list of possible matched sequences represented by their starting and ending base number. The main process is to get the SQL statement that can be used for querying in the database. The complexity suddenly increased when the misfit is introduced. For instance, a sequence “GA TCG” is the input for searching matches sequences in all genomes, and 2 misfits are tolerated. Possible matches are “GATCG”, “AATCG”, “TATCG”, “CATCG”, “AGTCG”, “GGCCG”, and etc. This implicates the power set of all combinations of misfits is generating. Therefore, in order to create the SQL statement from the input, the power set of all combinations of misfits has to be found.

In the power set algorithm, there are two for loops and a recursive call. The outside loop, “for(int i = 1; i <= misfits; i ++)”, iterates from 1 to the number of misfits. For each step, the method
getPowerSets(int m, int n, int p) is invoked. “m” represents the starting point to examine, starting with 0. “n” represents the number of possible combinations minus 1 starting with the length of the string minus the number of the misfits. “p” represents the steps in generating a combination, starting with the number of the misfits. In this method, the inner loop “for(int i = m; i <= n; i++)” generates all combinations in a step, recursively call the method getPowerSets with m + 1, n + 1, and p – 1. Prior to this inner loop, the value of p is examined. Every time when the value of p decreased to 0, it means a combination is generated, then the SQL statement with this combination of the misfits can be generated by the method generateStmt. An example of the output is shown below.

```
select t1.genome_no, t1.base_no, t5.base_no
from (select genome_no,base_no from dna_sequence) t1,
(select genome_no,base_no from dna_sequence where base = 'g') t2,
(select genome_no,base_no from dna_sequence where base = 'a') t3,
(select genome_no,base_no from dna_sequence where base = 't') t5
where t1.genome_no = t2.genome_no and t1.base_no = t2.base_no - 1 and
t1.genome_no = t3.genome_no and t1.base_no = t3.base_no - 1 and
t1.genome_no = t5.genome_no and t1.base_no = t5.base_no - 1
union all
select t1.genome_no, t1.base_no, t5.base_no
from (select genome_no,base_no from dna_sequence where base = 'c') t1,
(select genome_no,base_no from dna_sequence where base = 'a') t3,
(select genome_no,base_no from dna_sequence where base = 't') t5
where t1.genome_no = t3.genome_no and t1.base_no = t3.base_no - 1 and
t1.genome_no = t5.genome_no and t1.base_no = t5.base_no - 1
union all
select t1.genome_no, t1.base_no, t5.base_no
from (select genome_no,base_no from dna_sequence where base = 'c') t1,
(select genome_no,base_no from dna_sequence where base = 'a') t3,
(select genome_no,base_no from dna_sequence where base = 't') t5
where t1.genome_no = t3.genome_no and t1.base_no = t3.base_no - 1 and
t1.genome_no = t5.genome_no and t1.base_no = t5.base_no - 1
union all
select t1.genome_no, t1.base_no, t5.base_no
from (select genome_no,base_no from dna_sequence where base = 'c') t1,
(select genome_no,base_no from dna_sequence where base = 'g') t2,
(select genome_no,base_no from dna_sequence where base = 'a') t3,
(select genome_no,base_no from dna_sequence where base = 't') t5
where t1.genome_no = t2.genome_no and t1.base_no = t2.base_no - 1 and
t1.genome_no = t3.genome_no and t1.base_no = t3.base_no - 1 and
t1.genome_no = t5.genome_no and t1.base_no = t5.base_no - 1
union all
select t1.genome_no, t1.base_no, t5.base_no
from (select genome_no,base_no from dna_sequence where base = 'c') t1,
(select genome_no,base_no from dna_sequence where base = 'g') t2,
(select genome_no,base_no from dna_sequence where base = 'a') t3,
(select genome_no,base_no from dna_sequence where base = 't') t5
where t1.genome_no = t2.genome_no and t1.base_no = t2.base_no - 1 and
t1.genome_no = t3.genome_no and t1.base_no = t3.base_no - 1 and
t1.genome_no = t5.genome_no and t1.base_no = t5.base_no - 1
```

Discussion

In the beginning, we made the decision whether or not to parse the original files which are GeneBank formatted files into database. This decision depends on many different aspects. Based on using WinSeq tool, we found that it can only open one genome file at the time. That means it can only analyze a genome sequence each time. For the purpose of analyzing the similarities and differences for many genomes, this tool might not be good enough. The data we need to analyze is quite large. Average size of genome file is about 10 MB. If we choose to store data in flat files, all the data have to be loaded into the memory during the searching. The performance might be very slow because of loading millions even billions of characters into the memory. If we choose database, it can not only integrate data, but also increase the speed and efficiency of the program by structuring the data in a B+ tree. The algorithm could be much easier to be implemented in the database than in the flat files, more complex queries on the data can be used in the database. Besides, it is much easier to search in several genomes in the database than in several flat files.

In the design E-R model section, we determined to connect Gene entity with Genome Description entity directly instead of DNA sequence entity. Actually a DNA sequence contains many genes; it seems that they should have a relationship directly between them. If Gene and DNA sequence have weak entity relationship, the primary key of Gene should partly depend on the primary key of DNA sequence. The primary key of DNA sequence is the base number. Each base of DNA sequence will be recorded in a tuple in database. So in the Gene table, each gene name will be corresponding to the whole bases of DNA sequence around 5 million lines. To solve this redundancy, we made the decision to make the relationship between Gene entity and Genome Description. We also made the decision about the weak entity in E-R model section. We found that Gene, Reference and DNA sequence entities can not point out the unique value by their own key. They have to partially use the primary key of Genome Description entity to point out the unique value, so we identified Gene, Reference and DNA sequence as weak entities.

In the parser, there are two fundamentally different ways to read and write data from and into the files. One is by using the Streams; the other is by using the Readers and Writers. Streams access sequences by bytes, while Readers and Writers access sequences by characters. In our case, the files are composed of strings and characters, so the FileReader and FileWriter, which are the subclasses of the Reader and Writer, are applied to the parser. Java.util.Scanner is used for extracting strings delimited by space (default), because in the original files, most information is stored as strings. It reduced the complexity of the parser. However, it lowered the flexibility at the same time. Sometimes the tokens need to be decomposed, such as strings of DNA sequences, starting and ending base number of genes. subString, charAt and toCharArray are used in these cases in order to get the characters. Read in by characters could be a direct solution.

When we was trieing to load the data into the database, 5 files (genome description.txt, gene.txt, DNA sequence.txt, reference.txt, author .txt) are generated in the parser, and the format the parser wrote in is like: insert into tableName values (‘’, ‘’, ‘’). This SQL statement inserts one tuple into
The problem was that “DNA sequence.txt” file with 5 genomes contains nearly 25 million lines, means that 25 million tuples need to be added into the database using the `insert into` clause, and it was a never ending story through the network. Then the tool, SQL Loader, was introduced to our project. The parser was modified to generate output files without the insert `into` clause. With the control file, the SQL Loader efficiently loaded those files into the tables in the database (approximately 6 minutes for nearly 25 million tuples).

There are some requirements for implementing algorithms for sequence database searching (Ref.11). The first is sensitivity, which refers to the ability to find as many correct matches as possible. The second is selectivity, or specificity, which refers to the percentage of the incorrect matches in the searching results. The third is speed, which is the time it takes to get results from database searches. Ideally, we want to have the maximum sensitivity, selectivity, and speed in database searches. However, it’s not possible in reality. An increase in sensitivity usually leads to a decrease in selectivity. An improvement of speed always comes at the cost of lowered sensitivity and selectivity. Therefore, a compromise often has to be made. In our project, two experiments were carried to compare the time that the searching took.
Experiments

<table>
<thead>
<tr>
<th>Input sequence: cgaat</th>
<th>Max misfits: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Searching from 25 million bases</td>
<td></td>
</tr>
</tbody>
</table>

It takes around 2 minutes to search 5 bases in 25 million for perfect matches, and around 4 minutes to get the results with the same input bases but tolerating 2 misfits. The efficiency is acceptable, but it is not good enough for sequence analysis and genetic research. If the length of the input sequence reaches 9, it will take more than 60 minutes to get the results. As the number of misfits increase, the combinations grow exponentially. However, we can still get the result in this complex situation with 5 genomes in the database. On the other hand, in biological research, usually the searching is focused on only one genome, which could be relatively fast in our application (in around 15 seconds). Additionally in our implementations, a ranking system could be built later, so that the best result (perfect matches with 0 misfits) can be viewed at the top of the screen.
We did some optimizations in our implementations. We added an index to the table DNA_SEQUENCE which will be created as a B+ tree in the database. This made the searching much more efficiency (We can’t get the results in 30 minutes before introducing this index). Additionally, we changed the set operator “union” to “union all” (see Programming part), which also made the searching time from 20 minutes to 2. The operator “union” in Oracle database does not allow duplicates, so each time an ordering operation might be processed, which could be rather expensive.

Of course, many similar algorithms can efficiently do the search in only one genome. One suggestion to implement the searching in flat files is building a scoring matrix (inspired from the BLAST). The horizontal and vertical axes of the matrix represent the input sequence and the genome sequence. The matrix is built by scanning each base of one sequence with all bases in the other sequence. If a match is found, the value of the position is assigned to be 1. Otherwise, the value of the position is 0. Lines linking the position with value of 1 in diagonals indicate the perfect matches (0 misfits). Misfits can be represented by the nicks in the diagonal lines. The scoring can be weighted if necessary. This model requests loading the matrix into the memory, and the size of the matrix can be very large. The limitation of it is that only one genome can be searched at a time.

In our project, the algorithm for searching is exhaustive, which means the aim is to find the best or exact solution for the problem by examining all mathematical combinations. Another type of algorithm is heuristic, which uses computational strategy to find a near optimal solution by reducing the searching space according to some criteria. Our application has good sensitivity and selectivity, but the speed is relatively slow. If the number of the genomes in the database increases, the results can not be obtained within a realistic time. On the other hand, sometimes high accurate results are required in sequence analysis and genetic researches. In most case, searching a large sequence database using heuristic method is the better choice. Currently, BLAST and FASTA are the two major heuristic algorithms for performing database searches (Ref. 11). In our further work, we are going to try to implement some searching functions by using the heuristic algorithms.

**Conclusion**

As the genetic information increasing exponentially, the analysis and researches more and more rely on computational methods. Database is widely used in data management and data retrieval. Using relational database to store and retrieve biological data is one of the solutions to analyze rapidly increasing biological information. Our project implements the functions of collecting information from original biological file, building a relational database to store the data, and
“DNA boxes” Searching. Some functions described in our model, such as sequence translation and primer design, are not implemented in our project due to limited time. We applied one possible solution to the problem that searching the sequence fragment in several genomes with optional misfits allowed. The database provides the efficient way to implement it. It is good to have the core function worked, but the efficiency need to be improved when the input sequence and number of misfits increase. This application could be applied to some research field which needs accurate results of the alignment, and tolerate the time limit in a certain extent. We will continue working on modifying the database design, implementing more functions, and developing more efficient algorithms for various searching demands in the future.

Acknowledgements

First we would like to thank our supervisor, Henrik Bulskov, for all help he gave to us. It is our first computer project, in the beginning; we even didn’t know where we should start. He guided us to develop DNA sequence analysis software step by step. He also gave us a lot of good suggestions. Furthermore, we are grateful to Flemming G. Hansen at the Technical University of Denmark. We could be free to use the software, WinSeq, which is developed by him. We got some ideas from WinSeq for our project.

References

8. JDBC tutorial website: http://www.jdbc-tutorial.com/
Appendix

```java
public interface Parser {
    public void parse();
}
```

Paser.java
import java.util.*;
import java.io.*;

public class BioParser implements Parser {
    private String id;
    private String length;
    private String type;
    private String shape;
    private String division;
    private String date;
    private StringBuffer definition;
    private String accessionNum;
    private String versionNum;
    private String geneIndexNum;
    private StringBuffer keywords;
    private String source;
    private ArrayList<String> classification;
    private StringBuffer comment;
    private ArrayList<String> title;
    private ArrayList<String> journal;
    private ArrayList<ArrayList<String>> authors;
    private ArrayList<String> pubmed;
    Scanner in;
    PrintWriter gOut, gOut, dsOut, rOut, aOut;

    // constructor
    public BioParser(Reader reader, PrintWriter gOut, PrintWriter gOut,
                     PrintWriter dsOut, PrintWriter rOut, PrintWriter aOut) {
        in = new Scanner(reader);
        this.gOut = gOut;
        this.gOut = gOut;
        this.dsOut = dsOut;
        this.rOut = rOut;
        this.aOut = aOut;

        keywords = new StringBuffer();
        definition = new StringBuffer();
        classification = new ArrayList<String>();
        title = new ArrayList<String>();
        journal = new ArrayList<String>();
        authors = new ArrayList<ArrayList<String>>();
        pubmed = new ArrayList<String>();
    }

    // parse entire file
    public void parse() {
        if (in.next().equals("LOCUS")) // looking for "LOCUS"
            parseReader(in);
        else System.out.println("error");

        if (in.hasNext("REFERENCE")) // looking for "REFERENCE"
            parseReference(in);
        else System.out.println("error");

        if (in.hasNext("COMMENT")) // looking for "COMMENT"
            parseComment(in);
        else System.out.println("error");

        if (in.hasNext("FEATURES")) // looking for "FEATURE"
            parseFeatures(in);
        else System.out.println("error");
    }
}

BioParser.java
if (in.next().equals("ORIGIN")) // looking for entire sequence ...... can not delete number......:-(
    parseSequence(in);
    else System.out.println("error");
}

// parse LOCUS
void parseHeader(Scanner in)
{
    while (!in.hasNext("DEFINITION")
    {
        id = in.next();
        length = in.next();

        if (in.hasNext("bp")
            { in.next();
                type = in.next();
            }
        else
            { in.next();
                type = "protein";
            }

        shape = in.next();
        division = in.next();
        date = in.next();
    }

    if (in.hasNext("DEFINITION")
    {
        in.next();
        while (!in.hasNext("ACCESSION")
        {
            definition = definition.append(in.next()).append(" ");
        }
    }

    if (in.hasNext("ACCESSION")
    {
        while (!in.hasNext("VERSION")
            { accessionNum = in.next();
            }
    }

    if (in.hasNext("VERSION")
    {
        in.next();
        while (!in.hasNext("KEYWORDS")
            { versionNum = in.next();
                geneIndexNum = in.next().substring(3);
            }
    }

    if (in.hasNext("KEYWORDS")
    {
        in.next();
        while (!in.hasNext("SOURCE")
            { keywords = keywords.append(in.next()).append(" ");
            }
}
}}

    while(!in.hasNext("ORGANISM")) { //skip "SOURCE", because "ORGANISM" has the same information
        in.next();
    }

    if(in.hasNext("ORGANISM")) {
        in.next();
        source = in.nextLine().trim();
        while(!in.hasNext("REFERENCE")) {
            String temp = in.next();
            classification.add(temp.substring(0, temp.length()-1));
        }
    }

    //gOut.println("insert into genome_description" + "("\'GENOMEDN\'\," + "\'GINO\', " + "\'NAME\', " + "\'LENGTH\', " + "\'SHAPE\'\")");
    //gOut.println("insert into genome_description values" + "("\'" + id + "]", " + "\" + geneIndexNum + "]", " + "\" + source + "]", " + length + ", " + "\" + shape + "");
}

//parse REFERENCE (publication information)
void parseReference(Scanner in) {
    int x = 0;
    while(!in.hasNext("COMMENT")) {
        if(in.next().equals("REFERENCE")) {
            boolean a = false;
            boolean t = false;
            boolean j = false;
            boolean p = false;
            authors.add(new ArrayList<String>());
            while(!in.hasNext("REFERENCE") || in.hasNext("COMMENT")) {
                String ref = in.next();
                if(ref.equals("AUTHORS")) {
                    while(in.hasNext("TITLE")) {
                        if(in.hasNext("III,")) { 66 authors.get(x) = null 66
                            authors.get(x).add(in.next());
                        } else in.next();
                        a = true;
                        x++;
                    }
                } else {
                    StringBuffer sbt = new StringBuffer();
                    while(in.hasNext("JOURNAL")) {
                        String s = in.next();
                        if(s.contains("\""))
                            s = s.replace("\"", "prime");
                    } //
                }
            }
        }
    }
}

BioParser.java
```java
    sbt.append(s).append(" ");
    } title.add(sbt.toString());
    t = true;
    }
    if(ref.equals("JOURNAL")
    { StringBuffer sbj = new StringBuffer();
        while({!in.hasNext("PUBMED") || in.hasNext("REFERENCE") || in.hasNext("COMMENT")})
            { sbj.append(in.next()).append(" ");
        ) journal.add(sbj.toString());
    j = true;
    }
    if(ref.equals("PUBMED")
    { while({!in.hasNext("REFERENCE") || in.hasNext("COMMENT")})
            { pubmed.add(in.next());
        } p = true;
    }
    }
    if(!a)
    { x++;
    }
    if(!t) title.add("unknown");
    if(!j) journal.add("unknown");
    if(!p) pubmed.add("unknown");
}
}

//rOut.println("";GENOMENO" " + ""REFNO" " + ""TITLE" " + ""JOURNAL" " + 
""PUBMED"");
for(int i = 0; i < title.size(); i++)
    { rOut.println("insert into reference values" + "(\"" + id + ",", " + (i+1) + "," + "\"" + title.get(i) + "," + "\"" + journal.get(i) + "," + "\"" + pubmed.get(i) + ")");
}

//aOut.println("";GENOMENO" " + ""REFNO" " + ""NAME"");
for(int i = 0; i < authors.size(); i++)
    { if(authors.get(i) != null)
        for(int j = 0; j < authors.get(i).size(); j++)
            { String s = (String)authors.get(i).get();
                if(s.contains("\""))
                    s = s.replace("\", " ");
                aOut.println("insert into author values" + "(\"" + id + ",", " + (i+1) + "," + "\"" + s + "\"");">
```

BioParser.java
```java
// parse COMMENT
void parseComment(Scanner in)
{
    in.next();
    String comment = new StringBuffer();
    while (!in.hasNext("FEATURES"))
    {
        comment = comment.append(in.next()).append(" ");
    }
}

// parse gene FEATURES
void parseFeatures(Scanner in)
{
    String s1 = in.next();
    String s2 = in.next();
    if (s1.equals("CDS") && s2.contains(".."))
    {
        String [] split = location.split("[]");
        gOut.print("insert into gene values" + "(\"" + i + "\", ");
        if (Character.isDigit(location.charAt(0)) && location.contains("\", "))
        {
            String[] a = location.split("[]");
            gOut.print(a[0] + ", " + a[1] + ", ");
        } else if (location.charAt(0) == 'c' && location.contains("..") && location.contains("join"))
        {
            String[] c = location.substring(1, location.length()-1).split("[]");
            gOut.print(c[0] + ", " + c[1] + ", ");
        } else
        {
            gOut.print("null, " + "null, ");
        }
    } else
    {
        gOut.print("null, " + "null, ");
    }
    String s = in.next();
    while (!s.contains("/gene") && !s.contains("/locus_tag"))
    {
        s = in.next();
    }
    if (s.contains("/gene") || s.contains("/locus_tag"))
    {
        gOut.println("\"" + s.substring(7,s.length()-1) + "\"));
        else gOut.println("\"" + s.substring(7,s.length()-1) + "\"));
        boolean b = false;
        while (!in.hasNext("CDS") || in.hasNext("ORIGIN"))
        {
            String temp = in.next();
            String start;
            if (temp.contains("/translation="))
            {
                start = temp.substring(14);
                StringBuffer sb = new StringBuffer();
                sb.append(start);
                while (!in.hasNext("gene") || in.hasNext("ORIGIN"))
                { sb.append(in.next());
            }
        }
    }
}
```
//parse sequence
void parseSequence(Scanner in) {
    int a = 0;
    while (in.hasNext()) {
        String x = in.next();
        if (x.length() > 0 && x.charAt(0) == '/') {
            char[] charArray = x.toCharArray();
            for (int i = 0; i < charArray.length; i++) {
                System.out.println((i + a) + " number " + charArray[i]);
                if (i < charArray.length - 1) {
                    System.out.println("  ");
                }
            }
        }
        else {
            String s = new String(charArray);
            a = a + charArray.length;
        }
    }
}

BioParser.java
```java
import java.io.*;

public class BioParserTester
{
    public static void main(String[] args)
    {
        try
        {
            FileReader reader1 = new FileReader("NC_000913.gbk");
            FileReader reader2 = new FileReader("NC_000800.gbk");
            FileReader reader3 = new FileReader("AC_000091.gbk");
            FileReader reader4 = new FileReader("NC_008253.gbk");
            FileReader reader5 = new FileReader("NC_004431.gbk");

            PrintWriter gOut = new PrintWriter("genome_description.txt");
            PrintWriter qOut = new PrintWriter("gene.txt");
            PrintWriter dsOut = new PrintWriter("dna_sequence.txt");
            PrintWriter rOut = new PrintWriter("reference.txt");
            PrintWriter aOut = new PrintWriter("author.txt");

            Parser p1 = new BioParser(reader1, gOut, qOut, dsOut, rOut, aOut);
            Parser p2 = new BioParser(reader2, gOut, qOut, dsOut, rOut, aOut);
            Parser p3 = new BioParser(reader3, gOut, qOut, dsOut, rOut, aOut);
            Parser p4 = new BioParser(reader4, gOut, qOut, dsOut, rOut, aOut);
            Parser p5 = new BioParser(reader5, gOut, qOut, dsOut, rOut, aOut);

            p1.parse();
            p2.parse();
            p3.parse();
            p4.parse();
            p5.parse();

            gOut.close();
            qOut.close();
            dsOut.close();
            rOut.close();
            aOut.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

BioParserTester.java
```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBInterface {
    private Connection conn;

    public DBInterface() {
        try {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            conn = DriverManager.getConnection("jdbc:oracle:thin:8dat-db.ruc.dk:1521:datalogi", "XXIONG", "N021PXZISW");
        } catch (SQLException e) {
            System.out.println("error");
        }
    }

    public Connection getConnection() {
        return conn;
    }
}
```
import java.sql.*;

public class Genome implements Viewable {
    private String genome_name;
    private String genome_no;
    private String gi_no;
    private String shape;
    private int length;
    private ArrayList<Gene> genes;
    private ArrayList<Reference> refs;
    private Connection conn;

    public Genome(String genomeName, Connection conn) {
        genes = new ArrayList<Gene>();
        refs = new ArrayList<Reference>();
        try {
            this.conn = conn;
            Statement stmt = conn.createStatement();
            String queryStr = "SELECT * FROM genome_description WHERE genome_name = "+ genomeName + ";"
            ResultSet rSet = stmt.executeQuery(queryStr);
            while(rSet.next()) {
                genome_no = rSet.getString(1);
                gi_no = rSet.getString(2);
                genome_name = rSet.getString(3);
                length = rSet.getInt(4);
                shape = rSet.getString(5);
            }
            stmt.close();
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }

    public String getGenome_name() {
        return genome_name;
    }

    public String getGenome_no() {
        return genome_no;
    }

    public String getGl_no() {
        return gi_no;
    }

    public String getShape() {
        return shape;
    }

    public int getLength() {
        return length;
    }
}

Genome.java
```java
public ArrayList<Gene> getGenes()
{
    try
    {
        Statement stmt = conn.createStatement();
        String queryStr = "select gene_name from gene where genome_no = '" +
        genome_no + "'";
        ResultSet rSet = stmt.executeQuery(queryStr);
        ArrayList<Gene> g = new ArrayList<Gene>();
        while(rSet.next())
        {
            g.add(new Gene(genome_no, rSet.getString(1), conn));
        }
        genes = g;
    }
    catch(SQLException e)
    {
        e.printStackTrace();
    }
    return genes;
}

public Gene findGene(String gene_name)
{
    Gene g = new Gene(genome_no, gene_name, conn);
    return g;
}

public ArrayList<Reference> getRefs()
{
    try
    {
        Statement stmt = conn.createStatement();
        String queryStr = "select ref_no from reference where genome_no = '" +
        genome_no + "'";
        ResultSet rSet = stmt.executeQuery(queryStr);
        ArrayList<Reference> g = new ArrayList<Reference>();
        while(rSet.next())
        {
            g.add(new Reference(genome_no, rSet.getInt(1), conn));
        }
        refs = g;
    }
    catch(SQLException e)
    {
        e.printStackTrace();
    }
    return refs;
}

public String toString()
{
    return "gene name = " + gene_name + ", genome number = " + genome_no;
}

public void view()
{
    System.out.println(this);
}
```

Genome.java
import java.sql.*;

public class Gene implements Viewable
{
    private String genome_no;
    private String gene_name;
    private int start_base;
    private int end_base;

    public Gene(String genomeNo, String geneName, Connection conn)
    {
        try
        {
            Statement stmt = conn.createStatement();
            String queryStr = "select * from gene where genome_no = " + genomeNo + " and gene_name = " + geneName + " ";
            ResultSet rSet = stmt.executeQuery(queryStr);
            while(rSet.next())
            {
                genome_no = rSet.getString(1);
                start_base = rSet.getInt(2);
                end_base = rSet.getInt(3);
                gene_name = rSet.getString(4);
            }
            stmt.close();
        }
        catch(SQLException e)
        {
            e.printStackTrace();
        }
    }

    public String getGenome_no()
    {
        return genome_no;
    }

    public String getGene_name()
    {
        return gene_name;
    }

    public int getStart_base()
    {
        return start_base;
    }

    public int getEnd_base()
    {
        return end_base;
    }

    public String toString()
    {
        return "gene name = " + gene_name + ", start base = " + start_base + ", end base = " + end_base;
    }

    public void view()
    {
        System.out.println(this);
    }
}
import java.sql.*;

public class Reference implements Viewable {
    private String genome_no;
    private int ref_no;
    private String title;
    private String journal;
    private String pubmed;
    private ArrayList<Author> authors;
    private Connection conn;

    public Reference(String genomeNo, int refNo, Connection conn) {
        authors = new ArrayList<Author>();
        try {
            this.conn = conn;
            Statement stmt = conn.createStatement();
            String queryStr = "select * from reference where genome_no = " + genomeNo + " and ref_no = " + refNo + "";
            ResultSet rSet = stmt.executeQuery(queryStr);
            while(rSet.next()) {
                genome_no = rSet.getString(1);
                ref_no = rSet.getInt(2);
                title = rSet.getString(3);
                journal = rSet.getString(4);
                pubmed = rSet.getString(5);
                stmt.close();
            }
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }

    public String getGenome_no() {
        return genome_no;
    }

    public int getRef_no() {
        return ref_no;
    }

    public String getTitle() {
        return title;
    }

    public String getJournal() {
        return journal;
    }

    public String getPubmed() {
        return pubmed;
    }

    public ArrayList<Author> getAuthors() {
        return authors;
    }
}

Reference.java
```java
{ try
    
    String queryStr = "select author_name from author where genome_no = " + genome_no + 
    " and ref_no = " + ref_no;
    
    ResultSet rSet = stmt.executeQuery(queryStr);
    
    ArrayList<Author> g = new ArrayList<Author>();
    
    while(rSet.next())
    
    g.add(new Author(genome_no, ref_no, rSet.getString(1), conn));
    
    authors = g;
    
} catch(SQLException e)
    
    e.printStackTrace();

    return authors;

public String toString()
{
    
    return "ref number = " + ref_no + "\n" + "authors = " + authors + "\n" + "title = " + title + "\n" + "journal = " + journal + "\n" + "pubmed = " + pubmed;
}

public void view()
{
    
    System.out.println("this");
}

```
import java.sql.*;

public class Author implements Viewable
{
    private String genome_no;
    private int ref_no;
    private String author_name;

    public Author(String genomeNo, int ref_no, String name, Connection conn)
    {
        try
        {
            Statement stmt = conn.createStatement();
            String queryStr = "select * from author where genome_name = " + genomeNo + " and ref_no = " + ref_no + " and author_name = " + name + ";"
            ResultSet rSet = stmt.executeQuery(queryStr);
            while(rSet.next())
            {
                genome_no = rSet.getString(1);
                ref_no = rSet.getInt(2);
                author_name = rSet.getString(3);

                stmt.close();
            }
        }
        catch(SQLException e)
        {
            e.printStackTrace();
        }
    }

    public String getGenome_No()
    {
        return genome_no;
    }

    public int getRef_no()
    {
        return ref_no;
    }

    public String getAuthor_name()
    {
        return author_name;
    }

    public String toString()
    {
        return "genome number = " + genome_no + ", ref_no = " + ref_no + ", author name = " + author_name;
    }

    public void view()
    {
        System.out.println(this);
    }
}
import java.sql.*;

public class Search {

    private ArrayList<String> result; // the searching results list
    private String input; // searching input
    public StringBuffer queryStr; // the query string will be generated in the
    constructor
    private String[] x; // the array of strings used as artificial inner table names
    private ArrayList<String> s; // store the partial SQL statements that allow all
    possible misfit combinations
    private Connection conn; // connection

    public Search(String inputStr, int misFits, Connection conn) {
        // initializations
        result = new ArrayList<String>();
        input = inputStr;
        queryStr = new StringBuffer();
        s = new ArrayList<String>();
        this.conn = conn;
        x = new String[input.length()];
        for (int i = 0; i < x.length; i++)
            x[i] = "t" + (i + 1);
        queryStr = getQueryStr(misFits); // get query string
    }

    // get query string
    private StringBuffer getQueryStr(int misFits) {
        if (misFits > x.length) System.out.println("error"); // number of misfits can
        not be larger than the length of the input sequence
        else if (misFits == 0) { // perfect match
            String str1, str2, str3;
            str1 = "select " + x[0] + ", genome_no, " + x[0] + ", base_no, " + x[x.length
            - 1] + ", genome_no and " + x[i] + ", base_no - 1 and " + "\n",
            str2 = " from ";
            str3 = " where ";
            String temp;
            for (int i = 0; i < x.length; i++)
                temp = "(select genome_no, base_no from dna_sequence where base = " +
                input.charAt(i) + ") | " + x[i] + ", base_no and " + "\n",
                str2 = str2 + temp;
            str2 = str2.substring(0, str2.length() - 2); // cut the last comma
            str2 = str2 + "\n"
            for (int i = 0; i < x.length - 1; i++)
                temp = x[0] + " genome_no = " + x[i + 1] + "\n",
                str3 = str3 + " genome_no and " + x[i] + ", base_no = " + x[i + 1] + ", base_no - 1 and " + "\n",
                str3 = str3 + temp;
            str3 = str3.substring(0, str3.length() - 5); // cut the last "and"
            queryStr.append(str1 + str2 + str3);
        } else {
            for (int i = 1; i <= misFits; i++)
        }
    }
```java
{  
    getPowerSets(0, x.length - i, i); // get all possible combinations of misfits in the input sequence
}
queryStr.delete(queryStr.length() - 6, queryStr.length()); // omit the last "union"
return queryStr;
}

// get all possible combinations of misfits in the input sequence
private void getPowerSets(int m, int n, int p)
{
    if (p == 0) {
        generateStmt(s); // generate the partial statement for each combination
        s.remove(s.size() - 1);
        return;
    }
    for (int i = m; i <= n; i++) {
        s.add("(select genome_no,base_no from dna_sequence) " + x[i] + ","," + "; // with the misfit allowed
        getPowerSets(i + 1, n + 1, p - 1); // recursion
    }
    if (s.size() > 0) {
        s.remove(s.size() - 1);
    }
}

// generate the partial statement for each combination
private void generateStmt(ArrayList<String> s)
{
    String str1, str2, str3;
    str1 = " select " + x[0] + ", genome_no, ", x[0] + ", base_no, ", x[x.length - 1] + ", base_no" + "\n";
    str2 = " from ";
    str3 = " where ";
    String temp;
    for (int i = 0; i < x.length; i++) {
        boolean b = false;
        for (String st: s) {
            if (st.contains((CharSequence)x[i])) // find the misfit positions
            {
                b = true;
                temp = st;
                str2 = str2 + temp;
            }
        }
        if (!b) // if it is not a misfit position
        {
            temp = "(select genome_no,base_no from dna_sequence where base = "; + input.charAt(i) + ",") + x[i] + ",", + ";\n";
            str2 = str2 + temp;
        }
    }
    str2 = str2.substring(0, str2.length() - 2); // omit the last comma
    str2 = str2 + "\n";
}
```

Search.java
for(int i = 0; i < x.length - 1; i++)
{
    temp = x[0] + ".genome_no = " + x[i + 1] + ".genome_no and " + x[i] + ".base_no = " + x[i + 1] + ".base_no = 1 and " + "\n";
    str3 = str3 + temp;
}
str3 = str3.substring(0, str3.length() - 5); // omit the last "and"
queryStr.append(str1 + str2 + str3 + "\n" + " union" + "\n");

// do the searching in the database and return the results
public ArrayList<String> sequenceSearch()
{
    try
    {
        Statement stmt = conn.createStatement();
        ResultSet rSet = stmt.executeQuery(queryStr.toString());
        ArrayList<String> r = new ArrayList<String>();
        while(rSet.next())
        { 
            r.add(rSet.getString(1).trim() + " " + rSet.getString(2) + " " + rSet.getString(3));
            result = r;
        }
    }
    catch(SQLException e)
    {
        e.printStackTrace();
    }
    return result;
}
public interface Viewable
{
    public void view();
}