AN ABSTRACT OF THE DISSERTATION OF

John Smith for the degree of Doctor of Philosophy in Computer Science presented on September 23, 2011.

Title: The Meaning of Life

Abstract approved: _____

Joan Smythe

This is an abstract statement.

The Meaning of Life

by

John Smith

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented September 23, 2011
Commencement June 2012

Doctor of Philosophy dissertation of John Smith presented on September 23, 2011.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

John Smith, Author

ACKNOWLEDGEMENTS

I would like to acknowledge the Starting State and the Transition Function.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1: Introduction

1.1 Introduction to the Variant Annotation

1.2 Definition

1.2.1 Gene Structure

1.2.2 Potential Variant Effects

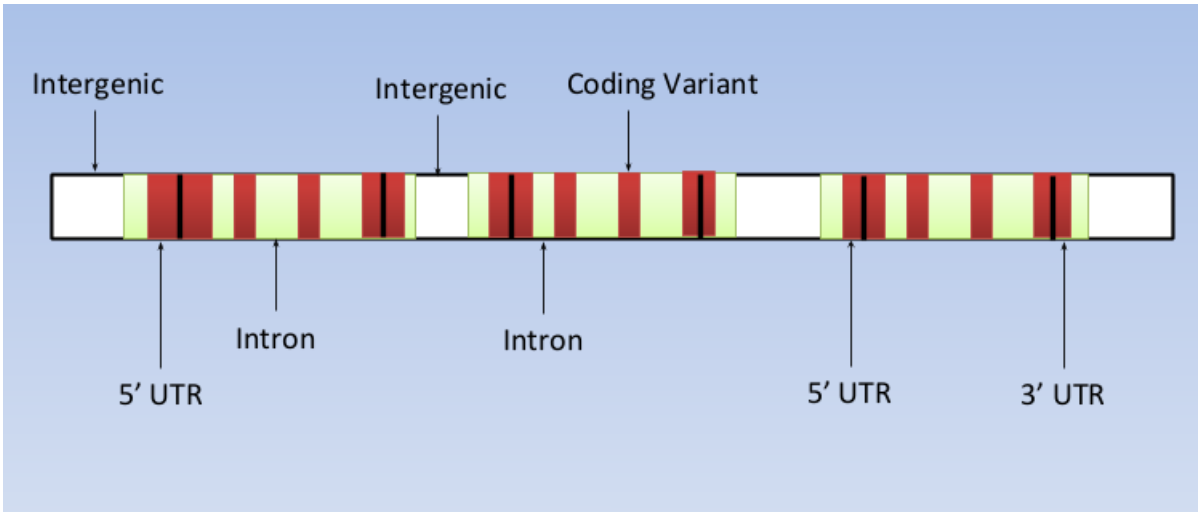

Figure 1.1: potential variant functional regions

In this project, we annotate 7 potential variant effects. The corresponding functional regions are show in Figure 1.1

- Intergenic variant
  Intergenic variant does not reside in any genes, resulting in that it does not alter any amino acid sequences. So its putative impact is low.

- Intron variant
  Intron variant resides in a gene, but not in an exon. Thus, it does not alter any amino acid either. Its putative impact is still low but a little higher than Intergenic variant.

- 5' UTR variant
  5' UTR variant resides in an exon, but is prior to the start of coding sequence. Although it still does not alter any amino acid sequences, it is relatively more damaging than intergenic and intron variant.

- 3' UTR variant
  3' UTR variant resides in an exon, but after the stop codon of coding sequences. The putative impact is similar to 5' UTR variant.

- Non-coding exon variant
  Non-coding exon variant resides in an exon. But the corresponding gene has no start or stop codon, resulting in no coding sequences. Thus, no amino acid sequences would be altered.

- Synonymous variant
  Synonymous variant resides in an exon and coding sequences, leading to altering a coding sequence. However, since several DNA codons correspond to one amino acid, the resulting amino acid sequence does not change.

- Non-synonymous variant
  Non-synonymous variant resides in coding sequences and alters an amino acid sequence. Its putative impact is high.

The following table shows the notation and putative impact for each variant effect.

Variant effect	Putative impact
intergenic_region	MODIFIER
intron_variant	MODIFIER
5_prime_UTR_variant	MODIFIER
3_prime_UTR_variant	MODIFIER
non_coding_exon_variant	MODIFIER
synonymous_variant	LOW
non_synonymous_variant	MODERATE

For non-synonymous variant, we annotate the reference amino acid, alternate amino acid and the position of the amino acid instead of the term.

Chapter 2: Methods

2.1 Input

2.1.1 Genome data

The genome data contains the whole nucleotide sequences of each chromosome. It is often stored in a faidx-indexed reference file in the FASTA format. We will use this file to get the amino acid codon of a given variant. Example:

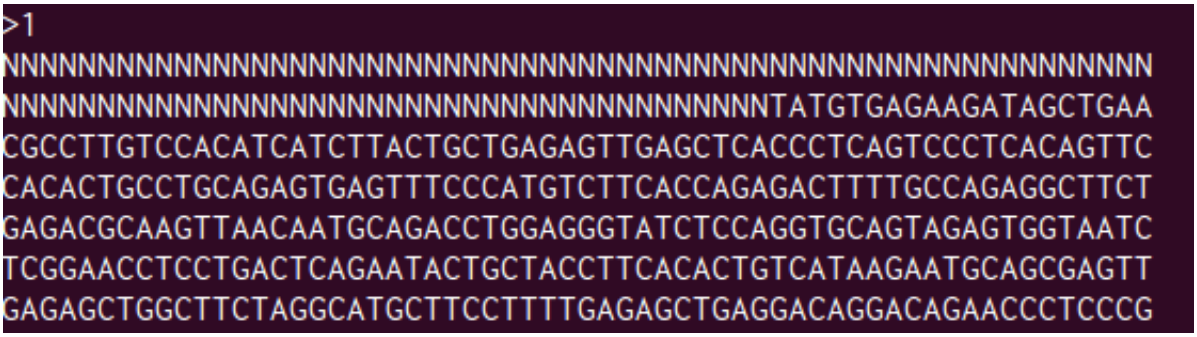

Figure 2.1: Genome data example

2.1.2 Gene structure data

The gene structure data contains information about gene structure, such as start codon, stop codon, exon, strand and gene id. This is the key data to determine the functional region of a given variant. Example:

2.1.3 Variant data

This is the data to be annotated. The key information includes chromosome, position, reference base and alternate variant alleles of the variant. The data is typically stored in VCF file. We will annotate the variant in the INFO field. Example:

```
chr1    canFam3_ensGene exon    8350657 8351219 0.000000        +       .       gene_id "E
chr1    canFam3_ensGene exon    8359233 8359447 0.000000        +       .       gene_id "E
chr1    canFam3_ensGene start_codon     8365671 8365673 0.000000        +       .       ge
chr1    canFam3_ensGene CDS     8365671 8365716 0.000000        +       0       gene_id "E
chr1    canFam3_ensGene exon    8365645 8365716 0.000000        +       .       gene_id "E
chr1    canFam3_ensGene CDS     8366024 8366362 0.000000        +       2       gene_id "E
chr1    canFam3_ensGene exon    8366024 8366362 0.000000        +       .       gene_id "E
chr1    canFam3_ensGene CDS     8410380 8410727 0.000000        +       2       gene_id "E
chr1    canFam3_ensGene exon    8410380 8410727 0.000000        +       .       gene_id "E
chr1    canFam3_ensGene CDS     8427874 8427976 0.000000        +       2       gene_id "E
chr1    canFam3_ensGene exon    8427874 8427976 0.000000        +       .       gene_id "E
chr1    canFam3_ensGene CDS     8453827 8453842 0.000000        +       1       gene_id "E
```

Figure 2.2: GTF file example

```
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  C       sample1
1       366459  .       G       T       .       PASS    SC=0;KS=NOVEL;CV=COVERED;PW=0.9999
1;n_ref_sum=40;n_alt_sum=KEEP      RC:AC:RS:AS     1.954141:91:4:3398      0:GG:23.515183:
1       724795  .       G       T       .       PASS    SC=0;KS=NOVEL;CV=COVERED;PW=0.9992
n_ref_sum=0;n_alt_sum=KEEP RC:AC:RS:AS     2.1856:76:4:2866        0:GG:24.594482:82
1       1244743 .       C       A       .       PASS    SC=0;KS=NOVEL;CV=COVERED;PW=0.9999
=3163;n_ref_sum=0;n_alt_sum=KEEP   RC:AC:RS:AS     2.027738:94:4:3575      0:CC:24.982382:
1       1244775 .       G       A       .       PASS    SC=0;KS=NOVEL;CV=COVERED;PW=0.9998
```

Figure 2.3: VCF file example

2.2 Output

Genes may overlap. Hence, one variant may have several effects. For each variant, the output is a list of effects. The potential effects are listed in section 1.1. In the project, the output is stored in a VCF file in the INFO field. To be specific, a new subfield, named ANN, will be appended to INFO field. The most important information is the effect of variants and the putative impact of the effect.

Besides the effect of variants, the corresponding Ensemble transcript id and common gene name are also annotated. All the annotation formats follow the standard variant annotation in VCF format. Example:

```
ANN=A|intron_variant|MODIFIER|ENSCAFT00000046825|ENSCAFG00000000012|ATP9B,
```

Figure 2.4: VCF file example

2.3 How it works

There are 3 datasets to be kept track of. If we search on the whole datasets for each variants, the total running time complexity is O(m*n*k), where m, n and k is the size of each file. In addition, the size of the datasets may be too large to load them all into RAM. For instance, the size of a typical human genome file is about 3.0 GB. To solve the difficulties in this problem, we develop several algorithms to annotate variants.

The whole problem can be split into 2 smaller subproblems. The first one is to obtain a list of genes in which a variant resides. The second one is to annotate each functional region. And if the variant resides in a coding sequence, get the corresponding DNA sequence around the variant and translate it into amino acids.

The most challenging part is to get the gene list since genes may overlap. The naive solution is to search on the whole genome for each gene, resulting in a time complexity of $O(N * K)$, where N is the number of genes and K is the number of variants. To speed up the annotation, 2 algorithms were proposed. One algorithm is array based and efficient for sorted variants while the other is tree based and efficient for unsorted variants.

2.3.1 Outline of the algorithm

2.3.2 Searching algorithm

2 algorithms are proposed for searching gene list. One is array based and most efficient for sorted variants while the other is tree based and most efficient for unsorted variants.

2.3.3 Array based searching

Array based search algorithm is most efficient for sorted variants. Genes are stored in sorted order in an array. While iterate though the variants, the algorithm keep track of the last visited gene. Each variant start searching from the last visited gene, not from the beginning. At the end, all the variants and genes are visited only once, resulting in a complexity of $O(N + K)$, where N is the number of genes and K is the number of variants.

Algorithm 1 VARIANT ANNOTATION

input : gene structure data G. reference genome data R. variant data V.
output: The annotated variant data V

for *each variant v in V at chromosome chr and position pos* **do**
    $gene\_list$ = get_gene_list(G, chr, pos)
    $A = \varnothing$
    if *gene_list is empty* **then**
        A = A + "intergenic_region"
    end
    for *each gene in gene_list* **do**
        if *gene has neither start codon nor stop codon* **then**
            A = A + "non_coding_exon_variant"
        end
        else if *v does not reside in any exon of gene* **then**
            A = A + "intron_variant"
        end
        else if *v resides prior to start codon of gene* **then**
            A = A + "5˙prime_UTR_variant"
        end
        else if *v resides after stop codon of gene* **then**
            A = A + "3˙prime_UTR_variant"
        end
        else
            get reference amino acid ra and alternative amino acid aa
            if $ra == aa$ **then**
                A = A + "synonymous_variant"
            end
            else
                A = A + "ra + pos + aa"
            end
        end
    end
    Annotate v with A
end

return The annotated variants V

2.3.4 Tree based searching

Tree based algorithm is most efficient for unsorted variants. In this algorithm, genes are stored in an interval tree. Although each variant searching on the whole gene, it takes only $O(logN)$

Algorithm 2 ARRAY BASED SEARCHING

input : gene structure data G, chromosome chr and position pos
output: a list of genes which reside in chr and overlap pos

get gene array $gene\_arr$ and last visited index i from G on chromosome chr

while $i <$ *length of* $gene\_arr$ *and pos* $>$ *the end of* $gene\_arr[i]$ **do**
    $i = i + 1$
end

update last visited index i to G
$gene\_list = \varnothing$
$j = i$

while $j <$ *length of* $gene\_arr$ *and* $gene\_arr[j]$ *overlap* pos **do**
    add $gene\_arr[j]$ to $gene\_list$
    $j = j + 1$
end

return $gene\_list$

time to get the gene list. Thus, the total time complexity is $O(K * (logN))$. If sort the variants and apply array based algorithm, the total time complexity would be $O(K * (logK) + N)$, which is larger than $O(K * (logN))$.

Interval tree is very similar to binary search tree. Standard binary search tree usually maintains a single value for each node while interval tree maintains an interval for each node. When searching genes, it makes a preorder traversal on the interval tree until the position is completely outside of the current node.

The key to keep binary search tree efficient is maintaining it balanced. Since genes rarely change, we don't need a balanced binary search tree algorithm here, such as Red Black tree. We could sort the genes by their starting position first and then build a balanced binary search tree with it.

2.4 Other issues

2.4.1 how to get amino acid

The whole genome sequence data is very large. For instance, the size of a typical human genome file is about 3.0 GB. However, since coding sequences amount to only 1.5% of whole genome,

Algorithm 3 TREE BASED SEARCHING

input : gene structure data G, chromosome chr and position pos
output: a list of genes which reside in chr and overlap pos

Function *get_gene_list(G, chr, pos)*
    get gene tree root $gene\_root$ from G on chromosome chr
    $gene\_list = \varnothing$
    $binary\_search(gene\_root, pos, gene\_list)$
    return $gene\_list$

Function *$binary\_search$(root, pos, gene_list)*
    if *$root$ is empty* **then**
        return
    end
    if *pos resides outside region of $root$* **then**
        return
    end
    g = the gene in $root$
    if *pos resides in g* **then**
        add g to $gene\_list$
    end
    $binary\_search(root.leftchild, pos, gene\_list)$
    $binary\_search(root.rightchild, pos, gene\_list)$

we could preload the coding sequence to the gene structure data. In this project, the final gene structure data is about 100 MB, which is smaller enough to load in most machines' memory.

2.4.2 strand

Another annoying problem is that genes have different strand. Different from positive strand, negative strand starts from a bigger position and ends at a smaller position. In addition, the nucleotide sequence must be reversed according to the format A-T, C-G.

Chapter 3: Result

3.1 Running time

3.1.1 Sorted variants

The following table shows the running time for sorted variants. There are 8777 variants in this experiment. The variants come from a real variant project. The time unit is second.

naive	array	tree
12.64	0.75	0.93

3.2 Memory cost

3.3 Success rate

APPENDICES

Appendix A: Redundancy

This appendix is inoperable.