Relatório Técnico – *Geneticz* : Ferramenta para Analise de Expressão Gênica

Pitágoras de A. Alves Sobrinho

Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)
Natal - RN – Brasil

alves.pitagoras@gmail.com

Abstract. *This technical report depicts the final project of the "Linguagem de Programação II" and "Estruturas de Dados Básicas II" courses: a tool for gene expression analysis, Geneticz. This software is able to analyze a file with the level of expression of genes in many samples to, using a statistical process of signature selection and a process of clustering, build a dendrogram about the inserted samples.*

Resumo. *Este relatório técnico descreve o projeto final das matérias Linguagem de Programação II e Estruturas de Dados Básicas II: a ferramenta para análise de expressão gênica, Geneticz. Este software é capaz de analisar um arquivo com o grau de expressão de genes em diversas amostras para, através de um processo estatístico de seleção de assinatura gênica e de um processo de Clustering, construir um dendograma relativo as amostras inseridas.*

1. Introdução

Através da comparação de grupos de amostras de genes (amostras de indivíduos sadios e amostras de indivíduos com um determinado problema, por exemplo) é possível obter uma assinatura de genes mais característicos de uma determinada condição, eliminando os genes que não variam em ambos os grupos. Isso pode ser feito através de testes estatísticos (como o teste de *Wilcoxon* e o *t de Student*) que resultam num *p-Value*. Quando um destes testes é feito entre as expressões gênicas de um mesmo gene num grupo de casos e num grupo de controle, o p-Value representa a significância daquele gene. Assim, estabelecendo um critério de corte máximo para este valor, podemos saber se um gene é significativo o suficiente para ser parte da assinatura gênica ou não.

Comparando as diferenças entre as assinaturas de diferentes amostras é possível, através de algoritmos de *Clustering*, agrupar hierarquicamente estas amostras em grupos e subgrupos. Cada grupo agrupado pode compartilhar de certas características relevantes, como a efetividade de certos tratamentos ou não em seus respectivos indivíduos. Além de evidenciar características, através da aplicação de um teste de validação cruzada, pode-se indicar com tais grupos uma possivelmente errada classificação anterior das amostras entre o grupo de casos e grupo de controle ou até uma assinatura instável.

Atualmente tais técnicas aqui descritas são utilizadas amplamente para diversos fins como selecionar grupos de pacientes para determinados tratamentos (como mencionado anteriormente), experimentos genômicos em animais e vegetais, melhoramento genético animal, resposta imune a doenças e pesquisas sobre metabolismo.

2. Problema Abordado

O trabalho consiste num software capaz de, inicialmente, ler um arquivo com os dados obtidos a partir de experimentos de *microarray* e/ou sequenciamento de segunda geração, para então determinar uma assinatura a partir das amostras descritas em tal arquivo.

Tal assinatura deve então ser processado em um algoritmo de agrupamento (*Clustering*) para classificar as amostras em grupos. O software deve ser capaz de aplicar um teste de validação cruzada sobre estes grupos para determinar se a assinatura é estável ou não.

A partir dos agrupamentos obtidos, deverá ser possível ao usuário gerar um dendograma representando as amostras.

3. Estruturas de Dados Utilizadas

Para que se tornasse possível a produção do *Geneticz* foi necessário implementar diversas estruturas de dados. Todas estas estruturas são baseadas no conceito de árvore binaria. Uma árvore binaria é uma estrutura de dados utilizada para armazenar itens na memória de forma hierárquica. Ela é composta de galhos, cada um contendo uma chave(opcional) e um valor, e cada galho pode ter 0, 1 ou 2 filhos e 1 pai. O pai e tais filhos também são galhos. Toda arvore possui um galho sem pai, este é a raiz, o galho inicial.

A arvore binaria de pesquisa é uma estrutura de dados derivada da arvore binaria comum. Ela possui as seguintes características adicionais:

- O valor da chave do filho esquerdo de um galho deve ser sempre menor que o valor da chave deste galho;
- O valor da chave do filho direito de um galho deve ser sempre maior que o valor da chave deste galho;

3.1 BinNodeDouble

Esta é a mais essencial de todas as estruturas utilizadas no projeto. O *BinNodeDouble* é um galho de uma Arvore Binaria. Ele possui uma chave identificadora única, um valor de ponto flutuante de dupla precisão armazenado e referências para seu nó pai, para um filho esquerdo e um filho direito. Ele pode ser utilizado tanto para construir arvores binarias convencionais quanto para arvores binarias de pesquisa e arvores AVL. Mesmo que uma instancia de BinNodeDouble não tenha pai, filho esquerdo ou filho direito associados, as referências ainda estarão presentes. Neste caso, elas estarão apontadas para um valor nulo.

3.2 AVLTreeDouble

Um problema da arvore binaria de pesquisa é que dependendo da sequência com que forem inseridos novos galhos, ela pode se tornar completamente desbalanceada. Para evitar este problema nas arvores binarias deste projeto, foi implementada uma árvore AVL. Esta arvore é derivada da arvore binaria de pesquisa, o diferencial é que nela a altura do filho direito e a altura do filho esquerdo de um galho não devem ter uma diferença de mais de 1. Na arvore AVL, sempre que é detectada uma diferença na altura entre os filhos de um galho de mais que 1, são aplicadas rotações nos galhos da arvore para restaurar o balanceamento.

Na *AVLTreeDouble* os galhos são instancias de *BinNodeDouble*. Neste projeto as arvores AVL são utilizadas para representar amostras genéticas, onde cada galho armazena o grau de expressão de um determinado gene.

3.3 ClusterNode

ClusterNode é uma estrutura que, através de herança, herda todas as características do BinNodeDouble. Além de armazenar uma chave e um valor um *ClusterNode* armazena, cada um, uma amostra genética (uma *AVLTreeDouble*).

Além disso é adicionada a funcionalidade de fazer uniões entre instancias de *ClusterNode*. No processo de união dois *ClusterNode* diferentes são unidos como filho direito e filho esquerdo de um novo *ClusterNode*, gerado durante a união. A assinatura genética deste *ClusterNode* pai é definida como a média da assinatura de seu filho direito e seu filho esquerdo. Esta função aproxima o *ClusterNode* do conceito de Conjunto. A principal diferença entre um *ClusterNode* e um Conjunto é que, no primeiro, a hierarquia de pais e filhos é importante.

Neste projeto o esta estrutura é utilizada para o agrupamento de amostras em grupos e então como arvore binaria para a criação do dendograma, onde as folhas são sempre as amostras iniciais.

3.4 HeapMinDouble

Um *HeapMin* é um tipo de arvore de dados binaria e assim como a arvore binaria de buscas possui uma propriedade própria: As chaves do filho esquerdo e filho direito de um galho devem ser sempre maiores que a chave deste galho. Assim é possível garantir que na raiz desta arvore sempre estará o elemento com a menor chave (ou a maior, caso fosse um *HeapMax*). No *HeapMinDouble* as chaves são valores de tipo *Double.*

O principal diferencial na implementação de um *HeapMin* em comparação com uma arvore binaria normal é que, ao invés de se utilizar referencias entre os galhos, o *HeapMin* é implementado ao longo de um vetor. Cada galho é um índice de tal vetor e, partindo do princípio que a arvore é binaria, são utilizados cálculos aritméticos nestes índices para determinar galhos pai, filho direito e filho esquerdo. Nesta estrutura o tipo de dados armazenado em cada nó é genérico, apenas a chave é sempre um *Double*.

Aqui o *HeapMinDouble* foi utilizado para determinar quais genes deveriam compor a assinatura gênica, classificando-os de acordo com seu *p-Value*, de acordo com a quantidade máxima de genes na assinatura informada pelo usuário.

4. Abordagem de Solução do Problema

Foi escolhida como abordagem principal tratar a assinatura genética, a matriz de distancias, os clusters e o dendograma como parte de uma única entidade: A GeneticAnalysis, uma análise genética).

Há duas formas de instanciar uma GeneticAnalysis. A primeira seria o usuário fornecer os parâmetros e arquivos necessários para ser construída uma análise genética e a segunda seria abrir uma análise pronta e salva num arquivo.

Caso a analise já esteja instanciada o usuário poderá realizar a Validação Cruzada nesta análise. Assim, a solução do problema se divide em duas partes principais: A geração da análise genética e a aplicação de uma validação cruzada sobre ela.

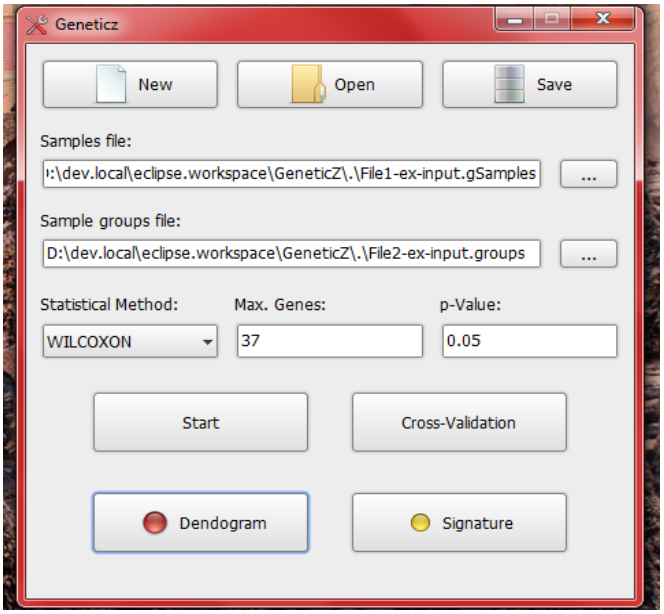

Figura 01: Interface gráfica central do software Geneticz.

4.1 Geração da Análise Genética

A construção de uma análise genética é o resultado de uma sucessão de soluções diferentes. A conclusão de cada uma dessas é vital para que a próxima possa ser iniciada: A matriz de distancias necessita de uma assinatura genética definida para ser elaborada e, para desenhar o dendograma, é necessário ter a acesso ao *ClusterNode* final gerado pela matriz de distancias.

4.1.1 Determinação da assinatura genética.

A determinação da assinatura gênica começa com os parâmetros fornecidos pelo usuário:

- Arquivo de amostras: Arquivo que descreve o grau de expressão dos genes em cada amostra;
- Arquivo de grupos de amostras: Especifica o nome e o grupo a quem pertence cada amostra definida no arquivo de amostras;
- Teste estatístico: Teste utilizado para comparar os grupos de amostras e definir a significância de um gene.
- Máximo de genes: O número máximo de genes que podem ser selecionados como parte da assinatura genética.
- p-Value: O resultado máximo do teste estatístico para que um gene possa ser considerado relevante. Um valor pequeno resulta em menos genes relevantes.

Após checar se os arquivos existem e se os outros dados inseridos são adequados, estes dados são passados a um algoritmo que irá ler o conteúdo dos arquivos e criar uma assinatura gênica. Para criar esta assinatura é necessário determinar quais genes são relevantes ou não. Isso é feito comparando os graus de expressividade de um gene em seus diferentes grupos definidos no arquivo de grupos de amostras. Esta comparação, feita através do teste estatístico t de Student ou Wilcoxon, resulta num p-Value.

É utilizado a estrutura HeapMinDouble com o p-Value de cada gene como e um vetor (com os graus de expressividade do gene em cada amostra associado a cada chave) como chave e valor dos nós, respectivamente, para armazenar os genes. Deste Heap são retirados os genes com os menores p-Value, até que tenha sido retirada a quantidade máxima de genes informada pelo usuário. Caso não haja tantos genes relevantes quanto o máximo que o usuário informou, o máximo de genes inseridos não será atingido.

Após isso as árvores AVL com os genes (cada uma representando uma amostra) serão utilizadas como ponto inicial para a construção da *GeneticAnalysis*.

4.1.2 Clustering

Para o agrupamento das amostras foi utilizado um algoritmo de *clustering* hierárquico. A comparação entre os agrupamentos é baseada em centroide, ou seja, o valor de cada agrupamento é a média entre seus sub-agrupamentos. O método para calcular a diferença é a Distancia Euclidiana.

Cada amostra é inserida num *cluster* separado, assim inicialmente cada cluster tem apenas um elemento e o valor de cada um é exatamente a amostra genética correspondente. O processo de *Clustering* se trata do agrupamento de *clusters* em grupos de acordo com suas semelhanças. É um ciclo de criar matrizes de diferenças entre cada um deles e unir os *clusters* com as menores diferenças, até que só reste um *cluster*.

Para a implementação dos *clusters*, se utiliza a classe *ClusterNode*. Note que, como a cada repetição há menos *clusters* para serem comparados, o tamanho da matriz de distancias vai diminuindo até que acabe quando há apenas 1x1 elementos (apenas um cluster).

4.1.3 Desenho do dendograma

A classe *Dendogram* irá receber o *ClusterNode* final gerado pelo processo de *Clustering* e então irá interpreta-lo como a raiz de uma arvore binaria. Os galhos seguintes à raiz são os grupos em que as amostras foram organizadas e as folhas de tal arvore serão, sempre, as próprias amostras. A classe *Dendogram* tem um algoritmo que, recursivamente, desenha cada galho numa imagem, que poderá ser posteriormente exibida na para o usuário e/ou salva num arquivo.

4.2 Aplicação de Técnica de Validação Cruzada

Para verificar a consistência dos agrupamentos gerados, utilizamos uma técnica de validação cruzada chamada *leave-one-out*. No *leave-one-out* a consistência do conjunto de dados é testada na ausência de cada uma dos dados. Neste caso, o conjunto de dados é o conjunto de amostras. Dada uma amostra especifica, o dendograma é construído do zero sem ela e então é verificada a consistência dos agrupamentos gerados em relação aos agrupamentos originais, de quando aquela amostra está presente.

O critério de consistência é que, dado que o grupo direito de amostras são as folhas do filho direito da raiz do cluster e o grupo esquerdo de amostras são as folhas do filho esquerdo da raiz, quando uma amostra é removida nenhum elemento do grupo direito se move para o grupo esquerdo e vice-versa.

5. Projeto Orientado a Objetos

Aqui será apresentado o diagrama de classes de todo o projeto e a descrição dos Padrões de Projeto aplicados.

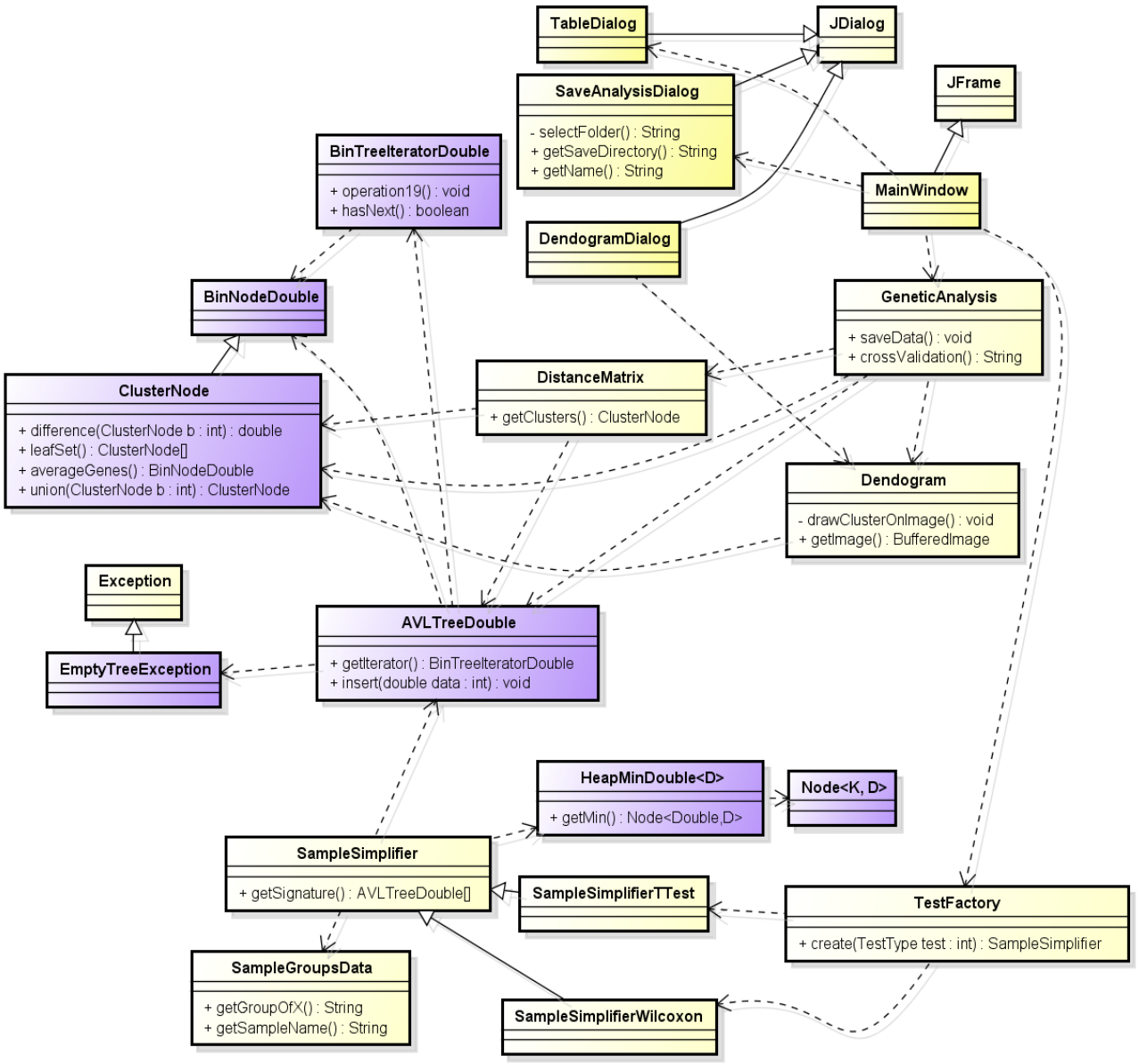

Figura 02: Diagrama de classes do software Geneticz. Classes relacionadas a estruturas de dados estão em azul, enquanto classes que representam janelas de interface com o usuário estão em amarelo.

5.1 Iterator

Iterator (ou iterador) é um padrão de projeto utilizado para permitir acesso aos elementos de um *container* (como uma lista ou uma arvore de dados) sem expor a estrutura interna do container e permitindo atravessar todos os elementos múltiplas vezes, ou mesmo ter iteradores diferentes atravessando os elementos simultaneamente.

Neste projeto este padrão foi utilizado para criar um iterador chamado BinNodeDoubleIterator que pode iterar por qualquer estrutura que use BinNodeDouble. A classe AVLTreeDouble tem um método “getIterator()” que retorna um iterador para seus galhos.

5.2 Template Method

No padrão de projeto *Template Method* a responsabilidade da implementação de algum ou alguns dos sub-passos de um algoritmo é deixada para as classes filhas da classe responsável por tal algoritmo. Dessa forma a implementação de um mesmo algoritmo pode variar de acordo com as classes filhas, sem que a parte do algoritmo que não deve variar seja reescrita.

Neste projeto o padrão *Template Method* foi utilizado na classe SampleSimplifier, responsável por transformar um arquivo de amostras e um arquivo de grupos de amostras num vetor de amostras com apenas genes significativos. No algoritmo para gerar tal vetor de amostras o sub-passo para fazer o teste estatístico pode utilizar diferentes métodos estatísticos. Este sub-passo é deixado como um método abstrato e é implementado diferentemente nas suas classes filhas SampleSimplifierTTest e SampleSimplifierWilcoxon.

5.3 Factory

Neste padrão de projeto a responsabilidade pela instanciação de uma classe A é passada para um método de uma classe B, a "fabrica". Dependendo dos argumentos passados para o método de B, podem ser produzidas instancias de filhos da classe A.

Neste projeto utilizasse uma fábrica para criar instancias de SampleSimplifierTTest e SampleSimplifierWilcoxon, sem que a classe que irá receber estas instancias precise conhecer alguma classe além da classe pai SampleSimplifier e a fábrica, TestFactory. O método estático TestFactory.create() recebe como parâmetro um tipo de teste estatístico e retorna uma instancia de uma classe diferente dependendo dele.

5.4 Singleton

O padrão de projeto *Singleton* tem um proposito bastante simples: assegurar que apenas uma instancia de uma classe exista e prover um ponto de acesso global para ela.

Neste projeto isso é utilizado na classe TestFactory. Isso foi feito tornando-a abstrata, dessa forma sempre há apenas uma instancia dela e (como ela é pública e seu único método é público) qualquer classe poderá ter acesso global a ela.

6. Algoritmos utilizados e sua Analise

Aqui será descrito o pseudocódigo dos algoritmos descritos na sessão "4." deste relatório técnico e suas analises assintóticas em O(n) (big O) em pior caso.

6.1. Assinatura Genética

O método SampleSimplifier.getSignature() é responsável por gerar uma assinatura genética baseado nos arquivos e parâmetros passados pelo usuário. A variável **n** é o número de amostras no arquivo de amostras e as constantes **K** e **Q** são, respectivamente, o número máximo de genes a serem inseridos e a quantidade original de genes.

Linhas de Código	***Número de passos***
getSignature(arquivoDeAmostras, arquivoDeGrupos, pValue, maxGenes){	
dadosDeGrupos = novo SampleGroupsData(arquivoDeGrupos)	2
assinatura = novo AVLTreeDouble[groupsData.getNumberOfSamples()]	3
numeroDeGenes = arquivoDeAmostras.numeroDeLinhas	2

heap = novo HeapMinDouble<Double[]>(numeroDeGenes)	2
enquanto(arquivoDeAmostras.temProximaLinha()){	Q
numeros = arquivoDeAmostras.proximaLinha().dividirPalavrasEmVetor()	3 x Q
vetorDeCasos = novo Vetor<Double>	2 x Q
vetorDeControle = novo Vetor<Double>	2 x Q
para (i de 1 até 10){	n x Q
grupoDaAmostra = groupsData.grupoDe(i)	n x Q
se (grupoDaAmostra = caso){	n x Q
vetorDeCasos.adicionar(num)	n x Q
}else {	
vetorDeControle.adicionar(num)	n x Q
}	
}	
pValueResultante = test(vetorDeCasos, vetorDeControle)	2 x Q
se (pValueResultante <= pValue){	Q
heap.insert(pValueResultante, numeros)	Log Q x Q
}	
}	
i = 0	2
enquanto(heap.getSize() > 0 e i < maxGenes){	K
gene = heap.getMin();	2 x K
para (x de 1 até 10){	n x K
assinatura[x].insert(gene[x]);	n x K x Log K
}	
}	
retornar assinatura	1
}	

Baseado nisso, o número de passos é dado pela função:

$$f(n) = n.(5.Q + K + K.\log K) + Q(11 + \log Q) + 3.K + 12$$

Dado que Q e K são constantes, podemos assumir que f(n) = O(n).

6.2. Clustering

O método DistanceMatrix.clustering() é responsável por agrupar os clusters de acordo com suas diferenças a até que haja apenas um cluster (antes dela ser chamada as amostras já foram distribuídas em seus respectivos clusters iniciais). O método DistanceMatrix.generateDifferenceMatrix() é utilizado pelo método citado anteriormente para refazer a matriz de distancias, de acordo com os novos agrupamentos. A variável **n** aqui se refere a quantidade de amostras genéticas e a constante **k** se refere ao número de genes na assinatura genética.

Linhas de código	**Número de passos**
clustering(){	
enquanto(space > 1){	**n**
para(y de 1 até space){	**n x n**
se(clusters[y] == clusters[y].getRoot()){	**n x n**
menorDiferença = 1	**2 x n x n**
para(x de 1 até space){	**n x n x n**
se(x != y){	**n x n x n**
se (matriz[x][y] < matriz[menorDiferença][y]){	**n x n x n**
menorDiferença = x	**n x n x n**
}	
}	
}	
se(clusters[menorDiferença] !=	**n x n**
clusters[menorDiferença].getRoot()){	
united =clusters[y].union(	

clusters[menorDiferença])	**n x n x k**
}	
}	
}	
para cada cluster em clusters{	**n x n**
cluster = cluster.getRoot();	**n x n**
}	
para(x de 1 até space){	**n x n**
para (y de x+1 até space){	**n x n x n**
se (clusters[x] == clusters[y]){	**n x n x n**
clusters.remover(y)	**n x n x n**
}	
}	
}	
space = clusters.numeroDeElementos()	**n**
generateDifferenceMatrix()	**4 x n x n + n**
}	
}	
generateDifferenceMatrix(){	
para(i de 1 até space){	**n**
para(j de 1 até space){	**n x n**
diferença = clusters[i].difference(clusters[j])	**2 x n x n**
matriz[i][j] = diferença	**n x n**
}	
}	
}	

Baseado nisso, o número de passos é dado pela função:

$$g(n) = n(4 + n(16+7n + k))$$

Retirando as constantes, obtemos que $g(n) = n^3$. Então podemos assumir que $g(n) = O(n^3)$.

6.3. Dendograma

A função Dendogram.drawClusterOnImage() é uma função recursiva que desenha cada cluster numa determinada imagem de forma a formar um dendograma de fácil leitura pelo usuário. A recursão desta função é na verdade um percurso de pré-ordem. A variável **n** é o número de agrupamentos.

Linhas de Código	**Número de passos**
drawClusterOnImage(cluster, imagem, xInicial, xFinal, y, maxY){	
larguraPorFolha = (xFinal-xInicial) / cluster.subLeafs()	2 + n
x = (xInicial + xFinal) / 2	3
endY = y + alturaPorNivel	2
se(cluster.isLeaf()){	1
endY = maxY;	1
}	
imagem.desenhaLinha(x, y, x, endY)	1
se(cluster.isLeaf()){	1
imagem.desenhaRetangulo(x-11, endY, 25, 15)	1
imagem.desenhaString(cluster.getName(), x-10, endY+12)	1
}	
se(cluster.getRightSon() != vazio e	
cluster.getLeftSon() != vazio){	2
xInicialEsquerdo = xInicial	2
xFinalEsquerdo = xInicial +	3 + n-1
*cluster.getLeftSon().subLeafs() * larguraPorFolha*	
se (xFinalEsquerdo = xInicial){	1
xFinalEsquerdo += larguraPorFolha	2
}	

```
                xInicialDireito = xFinalEsquerdo                                    2
                XFinalDireito = xFinal                                              2
                imagem.desenhaLinha((xInicialEsquerdo + xFinalEsquerdo) / 2,
                                endY,
                                (xInicialDireito + xFinalDireito) / 2,
                                endY)                                               1
                drawClusterOnImage(cluster.getLeftSon(), imagem, xInicialEsquerdo,
                                xFinalEsquerdo, endY, maxY)
                drawClusterOnImage(cluster.getRightSon(), imagem, xInicialDireito,
                                xFinalDireito, endY, maxY)
        }senão se(cluster.getLeftSon() != vazio){                                   1
                drawClusterOnImage(cluster.getLeftSon(), imagem, xInicial,
                                xFinal, endY, maxY)
        }senão se(cluster.getRightSon() != vazio){                                  1
                drawClusterOnImage(cluster.getRightSon(), imagem, xInicial,
                                xFinal, endY, maxY)
        }
}
```

Baseado nisso, o número de passos de uma execução deste algoritmo sem chamadas recursivas seria dado pela função:

$$H1(n) = 2.n + 27$$

Como a complexidade do percurso em pré-ordem é de O(n), o número de passos correto seria dado por:

$$H1(n) * O(n) = n(2.n + 27) = O(n^2)$$

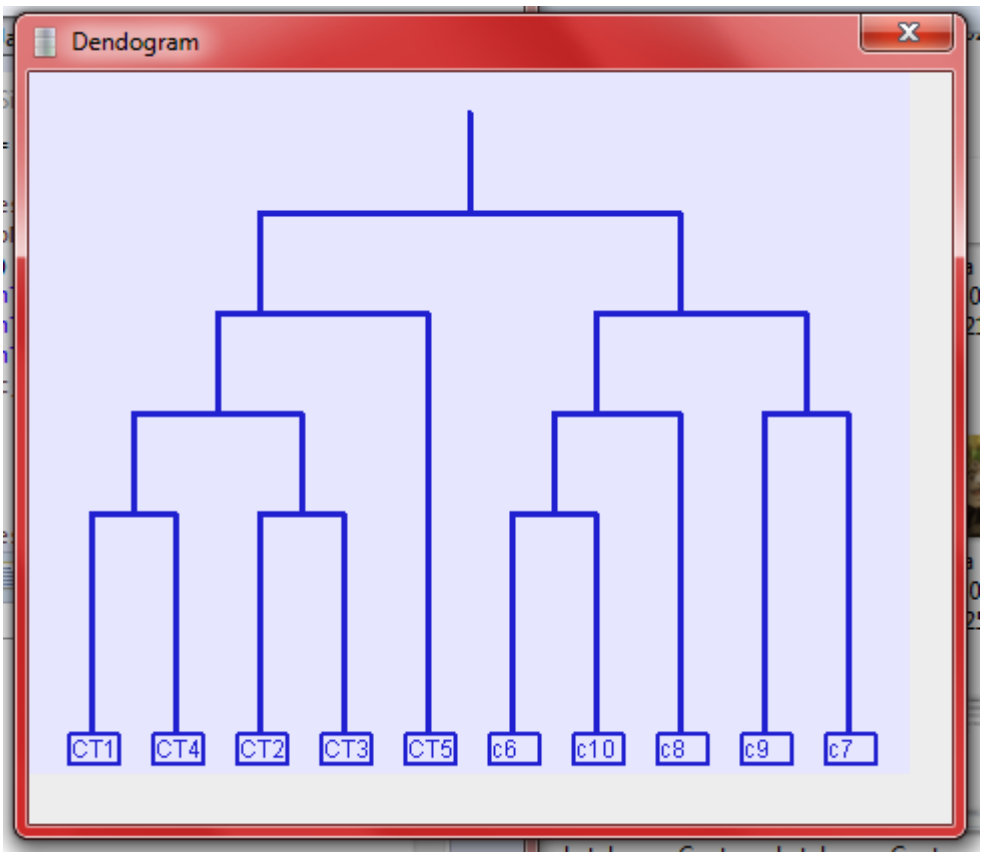

Figura 03: Exemplo de imagem gerada pelo algoritmo.

6.4. Validação Cruzada

Este método, GeneticAnalysis.crossValidation(), implementa o algoritmo de validação cruzada leave-one-out. Na análise do pseudocódigo seguinte, a variável **n** representa o número de amostras.

Linhas de código	Número de passos
crossValidation(){	
testResult= ""	**2**
rightGroup = cluster.getRightSon().leafSet()	**2 + n**
leftGroup = cluster.getLeftSon().leafSet()	**2 + n**
testResult += "Right group (" + rightGroup.length + "): "	**2**
para(i de 1 até rightGroup.length){	**n**
testResult += rightGroup[i] + ", "	**n**
}	
testResult += "\nLeft group (" + leftGroup.length + "): "	**2**
para(i de 1 até leftGroup.length){	**n**
testResult += leftGroup[i] + ", "	**n**
}	
para(i de 1 até signature.length){	**n**
testSignature = nova AVLTreeDouble[signature.length-1]	**2 x n**
x = 0	**2 x n**
para(j de 1 até i){	**n x n**
testSignature[x] = signature[j]	**n x n**
x++	**n x n**
}	
para(j de i+1 até signature.length){	**n x n**
testSignature[x] = signature[j]	**n x n**
x++	**n x n**
}	
testDistanceMatrix = nova DistanceMatrix(testSignature)	**n**
testDistanceMatrix.clustering()	**n x n x n x n**
testCluster = testDistanceMatrix.getCluster()	**2 x n**
testRightGroup = testCluster.getRightSon().leafSet()	**2 x n x n**
testLeftGroup = testCluster.getLeftSon().leafSet()	**2 x n x n**
para cada testNode em testRightGroup{	**n x n x n**
para cada leftGroupNode em leftGroup{	**n x n x n x n**
se(testNode.getName() = leftGroupNode.getName()){	**n x n x n x n**
testResult += "\nTesting without " + SampleSimplifier.groupsData.getSampleName(i) + ": "	**n x n x n x n**
testResult += "\nThe sample " + testNode.getName() + " of the left group was finded in the right group."	**n x n x n x n**
testResult += "\nTHE SAMPLES DO NOT HAVE PASSED ON THE CROSS-VALIDATION!"	**n x n x n x n**
retornar testResult	**n x n x n x n**
}	
}	
}	
para cada testNode em testLeftGroup){	**n x n x n**
para cada rightGroupNode em rightGroup){	**n x n x n x n**
se(testNode.getName() = rightGroupNode.getName()){	**n x n x n x n**
testResult += "\nTesting without " + SampleSimplifier.groupsData.getSampleName(i) + ": "	**n x n x n x n**
testResult += "\nThe sample " + testNode.getName() + " of the right group was finded in the left group."	**n x n x n x n**
testResult += "\nTHE SAMPLES DO NOT HAVE PASSED ON THE CROSS-VALIDATION!"	**n x n x n x n**
retornar testResult	**n x n x n x n**
}	
}	
}	

```
        }
        testResult += "\nCROSS-VALIDATION SUCCESSFULL!"                    1
        retornar testResult                                                 1
}
```

Baseado nisso, o número de passos é dado pela função:

$$e(n) = 12.n^4 + 2.n^3 + 14.n^2 + 14.n + 14 = O(n^4)$$

7. Conclusão

O projeto foi desenvolvido com sucesso e dentro do tempo esperado. O software desenvolvido, a ferramenta para análise de amostras genéticas GeneticZ, cumpre todos os objetivos traçados inicialmente e pode ser utilizada para quaisquer propósitos envolvendo produção de dendograma e definição de assinaturas gênicas.

8. Referências

- "p-Value", de *Wikipedia*. Acessado em 8/06/2015. Disponível em: http://en.wikipedia.org/wiki/P-value;
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7;
- Lesson 12: Cluster Analysis, STAT 505 (2015). The Pennsylvania State University, Department of Statistics. Acessado em 01/06/2015. Disponível em: https://onlinecourses.science.psu.edu/stat505/node/138;