



**TÉCNICO**  
LISBOA

**Relatório de Projecto**

# **Greedy Hill Climber para Aprendizagem de BNC's**

2012/2013

Algoritmos e Modelação Computacional

Prof: Paulo Mateus

Trabalho realizado por:

Manuel Figueiral nº 69264

Francisco Guerra nº 69874

Cheila Madaleno nº 70089

## Índice

1. Objectivos.....	3
2. Manual do Utilizador .....	4
3. Opções de Implementação e Tipos de Dados .....	6
3.1. 1ª Parte do Projecto .....	8
3.1.1. MDL .....	8
3.1.2. MDLdelta .....	12
3.1.3. Rede de Bayes .....	13
3.2. 2ª Parte do Projecto.....	18
3.2.1. grafMDLmin .....	18
3.2.2. CicleQ.....	18
3.2.3. randomGraf.....	19
3.2.4. greedy.....	20
3.2.4. Reader/Save/Load .....	21
4. Alterações efectuadas .....	22
5. Interfaces.....	23
6. Experiências .....	25
6.1 Tiróide .....	25
6.2 Hepatite.....	26
6.3 Cancro da mama.....	27
6.4 Diabetes.....	28
7. Conclusões e críticas.....	29

## 1. Objectivos

O objectivo deste projecto é desenvolver em *Java* um programa que funcione como classificador de doença baseado numa Rede de Bayes.

A principal finalidade do projecto é o diagnóstico de doenças como a diabetes, tiróide, e cancro da mama, no entanto tem mais aplicações dentro da área biomédica, bem como em muitas outras áreas, por exemplo, previsão da bolsa de valores e de resultados de eventos desportivos.

## 2. Manual do Utilizador

Para interagir com o programa desenvolvido e conseguir obter o diagnóstico de um paciente, o utilizador terá de seguir os passos que se referem de seguida.

Antes de devolver resultados a aplicação deve aprender a rede de Bayes da amostra a analisar. Para isso:

**1º** Depois de iniciar a 1ª aplicação (1\_Aprendizagem.jar), clicar no botão “Procurar”. Surge uma janela como a da figura 1. Escolher o ficheiro .csv que vai gerar a amostra.

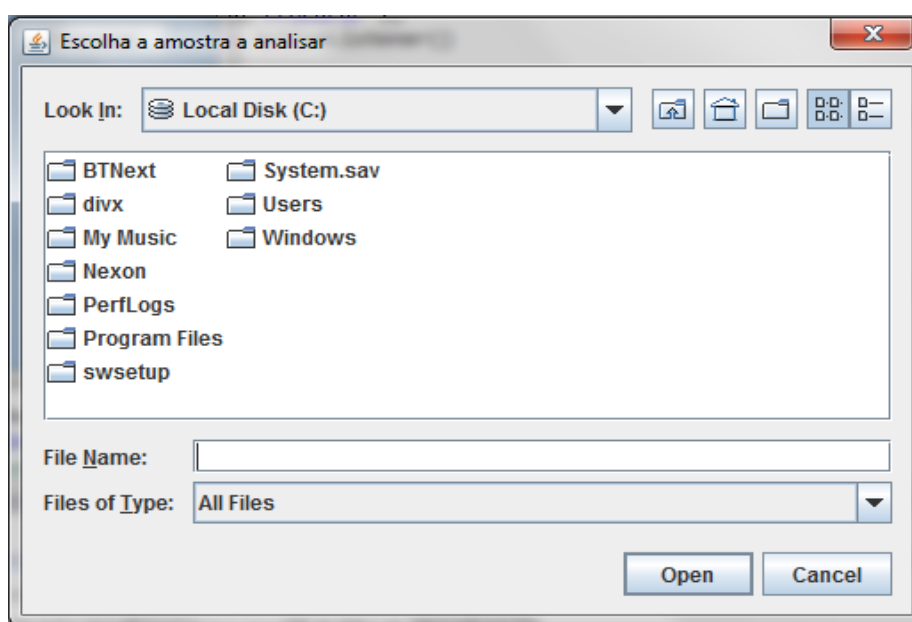


Figura 1: Interface gráfica “Aprendizagem”

**2º** Introduzir um número máximo de pais (menor ou igual a 4) e um número de grafos (da ordem dos parâmetros da amostra) e clicar “Aprender”.

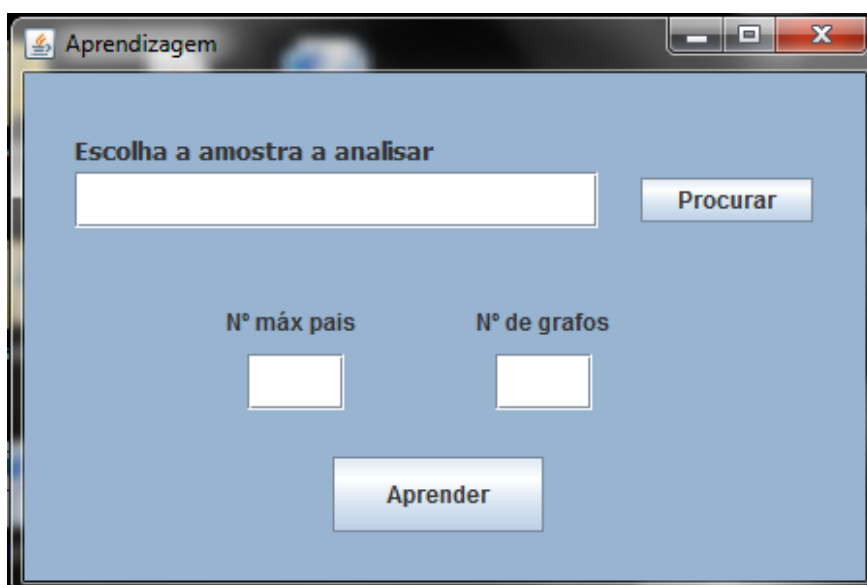


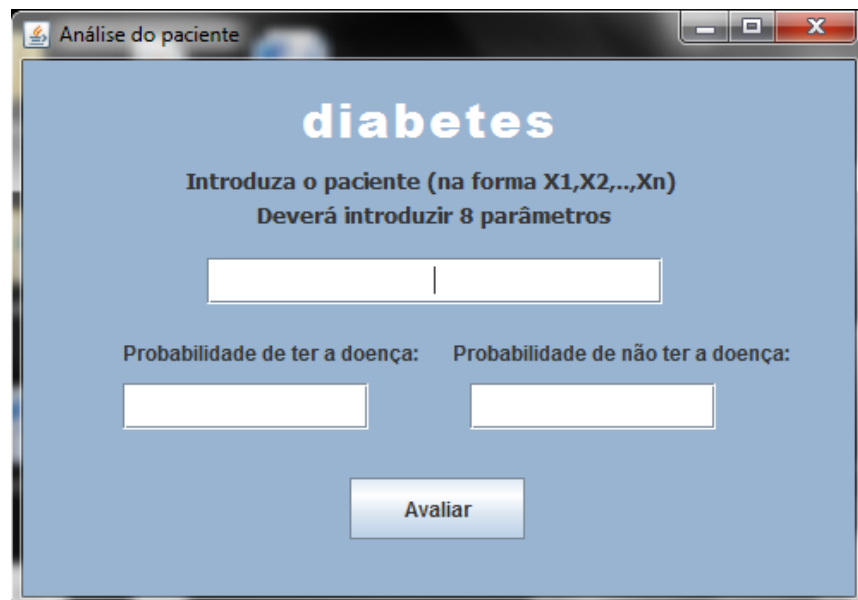
Figura 2: Interface gráfica “Aprendizagem”

Para que a aplicação classifique um paciente relativamente à doença que se quer analisar:

**3º** Iniciar a 2ª aplicação (2\_Análise.jar) (Figura 2).

**4º** Introduzir os parâmetros correspondentes ao paciente. Clicar “Avaliar”.

**5º** A aplicação devolve a probabilidade do paciente analisado ter ou não a doença.



**Figura 3:** Interface gráfica “Análise do paciente”

### 3. Opções de Implementação e Tipos de Dados

No início do projecto, escolhemos a classe do *Java*, *ArrayList*, que é uma classe que permite construir uma lista vazia necessária para criar listas de vectores e listas de valores inteiros ou decimais.

Então, recorremos ao *ArrayList* para poder considerar uma amostra como sendo uma lista de vectores, vectores esses que correspondem aos vários pacientes com as respectivas variáveis.

A vantagem da classe *ArrayList* é a possibilidade de usar as funções que a ela estão associadas como *add*, que adiciona um elemento ao array, *remove* que retira um elemento, *get* que retorna um determinado elemento do array e *size* que devolve o seu comprimento.

No entanto, apresenta como desvantagem o facto de ser muito lenta quando se pretende retirar elementos do array que não estejam na última posição.

Um tipo de dados utilizados foram os grafos, que considerámos como sendo uma matriz de adjacência. Um “1” na posição *i,j* da matriz, representa uma aresta. Ou seja, o parâmetro *i* é pai do parâmetro *j*.

É importante referir que a diagonal só pode ter zeros porque um parâmetro não pode ser pai de si próprio, e que a sua dimensão deve ser inferior em uma unidade, em relação à dimensão dos pacientes, uma vez que o grafo não considera o classificador.

Ao longo do projecto foi necessário importar as seguintes classes:

Para os *Array Lists*, utilizados na Amostra, na Rede de Bayes, na RW, importou-se “*java.util.ArrayList*”.

A classe *java.util.Arrays* foi importada para poder converter vectores para Strings, para facilitar a comparação e impressão de vectores.

Na função *randomGraf* é necessário gerar posições aleatórias no grafo. Deste modo, importou-se “*java.util.Random*” que permite gerar números dentro de um determinado intervalo.

Para a importação e exportação de dados para ficheiros são precisas várias funções, fornecidas por “**import** java.io.\*;”

Finalmente, devido à quantidade de recursos necessários para construir e manipular a interface importaram-se as seguintes classes:

**import** java.awt.BorderLayout;

**import** java.awt.EventQueue;

**import** javax.swing.JFileChooser;

**import** javax.swing.JFrame;

**import** javax.swing.JOptionPane;

**import** javax.swing.JPanel;

**import** javax.swing.border.EmptyBorder;

**import** javax.swing.JTextField;

**import** javax.swing.JButton;

**import** java.awt.event.ActionListener;

**import** java.awt.event.ActionEvent;

**import** java.io.File;

**import** java.io.IOException;

**import** java.util.Arrays;

**import** javax.swing.JLabel;

**import** java.awt.Font;

**import** java.awt.SystemColor;

**import** javax.swing.SwingConstants;

De seguida apresentamos alguns dos métodos que, devido à sua importância em todo projecto ou complexidade, achámos conveniente comentar, tanto da 1ª parte como da 2ª parte do projecto.

### 3.1. 1ª Parte do Projecto

Considerámos que seria mais rápido e eficiente incluir os métodos *domainArray*, que devolve o domínio de uma variável, e *length* dentro do *add*, para que cada vez que a amostra fosse alterada, os domínios fossem imediatamente actualizados e guardados, bem como o comprimento da amostra.

#### 3.1.1. MDL

A fórmula para o Minimum Description Length (*MDL*) é:

$$MDL(G|T) = \frac{\log_2 N}{2} |\Theta| - N \sum_{i=1}^n I_T(X_i; \Pi_i | C)$$

Devido à sua complexidade, tornou-se necessário dividir o *MDL* em funções mais pequenas e básicas, construídas na classe amostra, e que explicaremos de seguida.

O  $|\theta|$  é dado pela seguinte fórmula:

$$|\theta| = |D_C - 1| + \sum_{i=1}^n (k_i - 1) q_i$$

onde:

$$|q_i| = |D_{\Pi_i}| = |D_C| \times \prod_{X_j \in \Pi_i} |D_j|$$

e

$$k_i = |D_i|$$

Para construir o método *Qi*, que depois será utilizada no cálculo de  $|\theta|$ , foi necessário:

- criar o método *domainP*, que recebe um grafo e um parâmetro *i* e retorna o domínio dos pais desse parâmetro;
- recorrer ao *domainArray* para guardar o domínio de *C* (último valor de um vector da amostra).



```

protected int Qi (grafoOr g, int i)

    int c=1;
    ArrayList<Integer> x = this.domainP(g, i);
    if (x.size() == 0)

    {
        return c=domainsArray.get(this.tLength-1);
    }
    else
    {
        for (int j=0; j<x.size(); j++)
        {
            c*=x.get(j);
        }
        c*=domainsArray.get(this.tLength-1);
        return c;
    }

```

Assim,  $Q_i$  irá retornar a multiplicação do domínio do classificador  $C$  pela multiplicação dos domínios dos pais de  $i$ , caso  $i$  tenha pais.

Uma vez criado  $Q_i$ , construiu-se  $theta(\theta)$ , cujo código é apresentado em baixo:

```

protected int theta (grafoOr g)

    int c=0;
    for (int i=0; i<this.tLength-1; i++)
    {
        c+=(domainsArray.get(i)-1)*this.Qi(g, i);
    }
    c+=domainsArray.get(this.tLength-1)-1;
    return c;
}

```

Utilizou-se o *for* para percorrer todas as variáveis de um paciente e guardar numa variável o somatório do domínio - 1 de cada uma multiplicado por  $Q_i$ . No fim retorna o valor desse somatório somado ao domínio de  $C$ .

Para o cálculo de  $I_T$  recorreu-se à seguinte fórmula:

$$I_T(X;Y|C) = \sum_{x,y,c} Pr_T(x,y,c) \log \left( \frac{Pr_T(x,y,c)Pr_T(c)}{Pr_T(y,c)Pr_T(x,c)} \right)$$

onde  $Pr_T(x,y,c) = \frac{N_{x,y,c}}{N}$ ,  $Pr_T(x,c) = \frac{N_{x,c}}{N}$ ,  $Pr_T(y,c) = \frac{N_{y,c}}{N}$  e  $Pr_T(c) = \frac{N_c}{N}$ ;

De forma a simplificar  $I_T$  decidiu-se criar o método  $P_rTotal$  que retorna um elemento do somatório de  $I_T$  :

```
protected double PrTotal (int[] comb, int[] pos, int l) {
    double a = this.PrXYC(comb,pos,l);
    double b = this.PrYC(comb,pos,l);
    double c = this.PrXC(comb,pos,l);
    double d = this.PrC(comb,pos,l);

    if (a*d==0 || b*c==0 ){
        return 0;
    }
    else{
        return a*Math.log10((a*d)/(b*c));
    }
}
```

Inicializaram-se as variáveis **a**, **b**, **c** e **d** que guardam respectivamente, o valor de  $P_rXYC$ ,  $P_r(YC)$ ,  $P_r(XC)$  e  $P_r(C)$ , que são métodos criados separadamente. Se se garantir que a divisão de  $\frac{P_{rT}(x,y,c)P_{rT}(c)}{P_{rT}(y,c)P_{rT}(x,c)}$  não é uma indeterminação,  $P_rTotal$  retorna a multiplicação de  $P_{rT}(x,y,c)$  pelo  $\log_{10} \frac{P_{rT}(x,y,c)P_{rT}(c)}{P_{rT}(y,c)P_{rT}(x,c)}$  que corresponde ao termo do somatório de  $I_T$ .

O código de  $I_T$  é apresentado e comentado, por partes, de seguida:

```
protected double It (int xi, grafoOr g){
    ArrayList<Integer> pais = g.parents(xi);

    if (pais.size()==0){
        return 0;
    }
}
```

Foi necessário criar uma guarda para garantir que quando  $X_i$  não tivesse pais,  $I_T$  retornasse zero.

Posteriormente, construímos o vector *doms*, constituído pelo domínio de  $X_i$ , dos pais de  $X_i$  e de  $C$  e o vector das posições respectivas, *pos*. O vector *doms* é fundamental para serem criadas mais à frente combinações de vectores a partir deste, que juntamente com *pos*, irão ser as variáveis de entrada do método  $P_rTotal$ .

```

else
{
    int l = pais.size()+2;

    int[] doms = new int[l];
    int[] pos = new int[l];

    doms[0] = domainsArray.get(xi);
    pos[0]=xi;

    for (int j=1; j<l-1; j++)
    {
        doms[j]=domainsArray.get(pais.get(j-1));
        pos[j]=pais.get(j-1);
    }

    doms[l-1] = domainsArray.get(this.tLength-1);
    pos[l-1]=this.tLength-1;
}

```

Uma vez feito o vector *doms*, foi calculado o produtório dos seus elementos, cujo valor irá corresponder ao número de combinações possíveis de vectores.

```

int prod = 1;
for (int k = 0; k<l; k++)
{
    prod*=doms[k];
}

double sum = 0;

```

Inicializou-se a variável *comb*, que é um vector gerado pelo método *fComb*. Na variável *sum* foi-se guardando a soma dos cálculos das probabilidades de cada combinação possível de vector, *comb*, obtidos pelo método *PrTotal*

```

int[] comb;
for (int i=0; i<prod; i++)
{
    comb = this.fComb(i,doms);
    sum += this.PrTotal(comb,pos,l);
}

return sum;
}
}

```

Por fim, a partir de todos os métodos apresentados anteriormente foi possível construir o MDL.

Inicializou-se a variável  $x$  que guarda o valor do primeiro termo do  $MDL$ , que corresponde a  $\frac{\log_{10} N}{2} |\theta|$ . Através de um ciclo *for*, guardou-se o valor do somatório do  $I_T$  de todas as variáveis de um paciente na variável  $y$ . Findo esse ciclo, é retornada a diferença entre  $x$  e a multiplicação do tamanho da amostra (que corresponde ao número de pacientes) pelo somatório de  $I_T$ .

```
public double MDL (amostra a)
    //Retorna o MDL score da amostra
{
    double x = (Math.log10(a.amostraLength)/2)*a.theta(this);
    double y = 0;

    for (int i=0; i<a.tLength-1; i++)
    {
        y += a.It(i, this);
    }

    return x-y*a.amostraLength;}

```

### 3.1.2. MDLdelta

#### *MDLdelta*

Para a implementação do cálculo da variação de  $MDL$  de um grafo (devida à adição ou remoção de uma aresta), concluiu-se que fazer o  $MDL$  score do grafo novo menos o  $MDL$  score do grafo inicial não seria nada eficiente. No caso de se fazer esta diferença, várias parcelas iriam desaparecer, visto que apenas os pais de um parâmetro são alterados. Desta forma, foi implementada a seguinte expressão matemática:

$$\Delta MDL = \frac{\log N}{2} \times \theta_{Delta} - N \times \left[ \sum_{XY'C} P_{(XY'C)} \times \log \left( \frac{P_{(XY'C)}}{P_{(Y'C)}} \right) - \sum_{XYC} P_{(XYC)} \times \log \left( \frac{P_{(XYC)}}{P_{(YC)}} \right) \right]$$

Foi criado um método chamado `sumItDelta`, que vai permitir calcular cada um dos somatórios: `sum1` (quando recebe o vector dos domínios de  $XYC$ ) e `sum2` (quando recebe o vector dos domínios de  $XY'C$ ). `sum1` e `sum2` são as designações que estes somatórios tomam no código do método `MDLdelta`.

O parâmetro  $\theta_{\text{Delta}}$  é calculado dentro do método MDLdelta. Esta fórmula será sempre aplicada ao grafo sem a aresta. Isto é, para o caso em que se adiciona uma aresta ao grafo, o  $q_i$  vai receber o grafo inicial. Caso contrário, se for removida uma aresta, o  $q_i$  vai receber o grafo alterado, sem o parâmetro  $\theta_{\text{Delta}}$  retornado com sinal negativo. É ainda inicializado um novo grafo “ng”, inicialmente igual ao grafo “this”, ao qual será adicionada ou retirada uma aresta, Permite-nos desta forma guardar o grafo alterado, sem alterar a matriz `ma[][]` (variável global da classe `grafoOr`).

$$\theta_{\text{Delta}} = q_i \times (D_{\text{filho}} - 1) \times (D_{\text{pai}} - 1)$$

### 3.1.3. Rede de Bayes

*BN*: Método construtor que recebe um grafo, um conjunto de dados e um double *S* e retorna a rede de Bayes com as distribuições DFO amortizadas com pseudo-contagens *S*.

$$\Theta_{i|w_i}(d_i) = \frac{|T_{d_i,w_i}| + S}{|T_{w_i}| + S \times |D_i|}.$$

O nosso método vai, a partir de uma lista com  $(Xi, \text{Pais de } Xi, C)$ , retornar uma lista de vectores com probabilidades. Vão existir tantas sublistas quanto o número de parâmetros *Xi* e tantas probabilidades desse vector quanto o número de subvectores criados a partir do vector  $(\text{Dom } Xi, \text{DomP } Xi, C)$ .

Para além disto o método *BN* vai ainda criar paralelamente outra lista de sublistas com os subvectores. Vão existir tantas sublistas quanto o número de parâmetros *Xi* e tantos subvectores quanto o produtório dos valores do vector em causa. Esta lista vai ter particular interesse no seguinte método *prob*.

```
public class BN {
    amostra am;
    grafoOr graf;
```

Inicialização da *ArrayList rede* e *listaComb* que são respectivamente a Rede de Bayes e as combinações de vectores que entram para o cálculo das probabilidades.

```
ArrayList<double[]> rede = new ArrayList<double[]>();  
ArrayList<ArrayList<int[]>> listaComb = new ArrayList<ArrayList<int[]>>();
```

Método que vai criar uma Rede de Bayes vazia

```
public BN() {  
  
}
```

Método que cria uma nova Rede de Bayes.

```
public BN(amostra a, grafoOr g, double s) {  
  
    am = a;  
    graf = g;
```

Para cada parâmetro  $X_i$ , representado por  $i$  no nosso código, cria-se uma lista pais com os pais de  $i$ . A partir desses dados e do classificador ( $C$ ) criaram-se os vectores  $doms = (DomX_i, DomPais\ X_i, DomC)$ ,  $pos = (X_i, Pais, C)$ ,  $pos2 = (Pais, C)$ .

```
ArrayList<Integer> pais;  
  
for(int i=0; i<a.tLength-1; i++)  
{  
    pais = g.parents(i);  
    int l = pais.size()+2;  
  
    int[] doms = new int[l];  
    int[] pos = new int[l];  
    int[] pos2 = new int[l-1];  
  
    doms[0] = a.domainsArray.get(i);  
    pos[0]=i;  
  
    for (int j=0; j<l-2; j++)  
    {  
        doms[j+1]=a.domainsArray.get(pais.get(j));  
        pos[j+1]=pais.get(j);  
    }  
  
    for (int k=0; k<l-1; k++)  
    {  
        pos2[k]=pais.get(k+1);  
    }  
  
    doms[l-1] = a.domainsArray.get(a.tLength-1);  
    pos[l-1]=a.tLength-1;  
    pos2[l-2]=a.tLength-1;
```

O produtório do vector *doms* é a referência para o número de subvectores possíveis das diferentes combinações originadas a partir de *doms*.

```
int prod = 1;
for (int f = 0; f<l; f++)
{
    prod*=doms[f];
}
```

Criou-se uma *ArrayList combsXi* que vai incluir todos os subvectores *comb* criados a partir do vector *doms* em questão. Os subvectores *comb* vão surgir da função *fComb* para cada valor de *d* = 0 até ao produtório do vector *doms*. Cada um dos vectores *comb* vai ter um correspondente vector *comb2* que tem os mesmos valores de *comb* excepto a primeira posição que vai deixar de existir. A partir destes dois últimos vectores de valores, *comb* e *comb2*, e dos vectores de posições *pos* e *pos2* é então possível calcular as probabilidades *Pdfo* (\*) que vão ser guardadas num vector *dfo*.

```
double[] dfo = new double[prod];
ArrayList<int[]> combsXi = new ArrayList<int[]>();
int[] comb = new int[l];
int[] comb2 = new int[l-1];
double Pdfo = 0;

for (int d=0; d<prod; d++)
{
    comb = a.fComb(d,doms);
    combsXi.add(comb);

    for (int q=1; q<l; q++)
    {
        comb2[q-1]=comb[q];
    }

    Pdfo = ((double)a.count(pos,
comb)+s)/(a.count(pos2, comb2)+s*a.domainsArray.get(i));
    dfo[d] = Pdfo;
}
```

Finalmente, após a análise anterior em relação ao parâmetro  $i$  e antes de passar para o seguinte parâmetro  $i + 1$ , é acrescentada à rede o vector das probabilidades  $dfo$  e, paralelamente, acrescentada à *listaComb* a lista de subvectores *comb*.

```

        rede.add(dfo);
        listaComb.add(combsXi);
    }}

```

*prob*: Recebe uma rede de Bayes e um vector e retorna a probabilidade desse vector.

A nossa função, de um modo geral, vai receber um vector com diversos parâmetros e com um classificador ( exemplo:  $(X_0, X_1, X_2, \dots, C)$ ). Para cada um dos parâmetros vai ser criado um vector  $(X, \text{Pais de } X, C)$  que vai ser comparado com os vectores da *ListaComb*, resultando a posição desse vector. Seguidamente vamos retirar essa mesma posição da lista *dfo* que tem a probabilidade  $Pdfo$ . Finalmente cria-se um vector com essas probabilidades e com a probabilidade de  $C$ . Fazendo o produtório desse vector obtemos o valor final do método *prob*.

```

public Double prob (int[] t){
    for (int x=0; x<am.tLength; x++)

```

Fizemos uma guarda que retorna null se alguma das variáveis do vector de entrada  $t$  exceder o domínio da amostra.

```

        {
            if (t[x]+1>am.domainsArray.get(x))
            {
                return null;
            }
        }

```

Criámos uma *ArrayList* *pais* que vai incluir os pais de cada variável do vector de entrada, excepto o classificador, e também um vector *probs* que vai ter a probabilidade correspondente a cada vector  $(X, \text{Pais de } X, C)$ .

```

ArrayList<Integer> pais;
double[] probs = new double[am.tLength];

for (int i=0; i<am.tLength-1; i++)
{
    pais = graf.parents(i);
    int l = pais.size()+2;

```



Criamos o *vector comb* =  $(X, \text{País de } X, C)$  para cada um dos parâmetros  $X$ .

```
int[] comb = new int[1];
comb[0] = t[i];
comb[1-1] = t[am.tLength-1];

for (int j=1; j<1-1; j++)
{
    comb[j] = t[pais.get(j-1)];
}
```

Cada um dos vectores *comb* deste método foi comparado aos vectores *comb* (combinações de vectores) da *ListaComb* do método anterior. A comparação é feita por conversão dos vectores para strings usando a função *Array.toString*. Após determinar a posição do vector pertencente à *ListaComb*, obtém se a probabilidade *pdfo* respectiva na Rede de Bayes *rede* sendo depois inserida no vector *probs*.

```
int position = -1;
for (int k=0; k<this.listaComb.get(i).size(); k++){
if(Arrays.toString(comb).equals(Arrays.toString(this.listaComb.get(i).get(k)))
))
    {
        position = k;
        break;
    }
}
probs[i] = this.rede.get(i)[position];
}
```

Calculámos ainda a probabilidade do classificador  $C$  do vector introduzido, sendo a última posição do vector *probs*.

```
int[] v = {t[am.tLength-1]};
int[] p = {am.tLength-1};
probs[am.tLength-1]=
((double)am.count(p,v))/am.amostraLength;
```

Finalmente fazemos o produtório do vector *probs* que dá a probabilidade a retornar pelo método depois de terem sido analisados todos os parâmetros do vector de entrada.

```
double prod = 1;
for (int f = 0; f<am.tLength; f++)
{
    prod*=probs[f];
}

return prod;}
```

## 3.2. 2ª Parte do Projecto

### 3.2.1. grafMDLmin

Este método foi criado para que, ao receber um grafo, retornasse um grafo resultante da adição e remoção de arestas, de modo a minimizar o seu MDL score.

Para todas as entradas da matriz (grafo), é utilizado o método *MDLdelta* para verificar se a adição/remoção da aresta contribuiria para minimizar o MDL score do grafo. Dado que se pretende minimizar o MDL score do grafo, apenas nos interessa memorizar os valores de *MDLdelta* negativos, bem como as respectivas posições em que isso acontece.

De seguida, procura-se o valor mais negativo, e tenta-se alterar o grafo nessa posição. Se não for possível fazer essa alteração, devido ao número máximo de pais desse parâmetro já ter sido atingido, elimina-se esse valor do ArrayList e procura-se novamente o valor de *MDLdelta* mais negativo.

Se eventualmente se conseguir alterar o grafo, é criado um novo ArrayList de *MDLdelta's* negativos e das suas respectivas posições e tenta-se alterar o grafo novamente.

Se depois de se percorrer todo o ArrayList não tiver sido possível alterar o grafo, o "aux" passa a "false" e o método retorna o grafo (grafo de MDL score mínimo).

### 3.2.2. CicleQ

Este método recursivo foi criado para que não fossem adicionadas arestas ao grafo que o tornassem cíclico. Isto é, se o parâmetro 1 for pai do parâmetro 2, e este for pai do parâmetro 3, não poderá ser adicionada uma aresta que faça com que o parâmetro 3 seja pai do parâmetro 1.

Admitindo que se pretende adicionar uma aresta que aponta do pai para o filho, o procedimento do algoritmo é o seguinte: vão ver verificados os pais do pai, e assim sucessivamente. Se nesta pesquisa, se encontrar o futuro parâmetro filho, o método retorna "true", isto é, informa que a adição da aresta tornaria o grafo cíclico.

Se nesta pesquisa recursiva, se chegar a um ponto em que já não existem mais pais a analisar, o método retorna “false”, o que se significa que se poderá adicionar a aresta sem que o grafo se torne cíclico.

### 3.2.3. randomGraf

Este método vai receber um número máximo de pais *Npais* por parâmetro (sem contar com o classificador uma vez que não existem entradas no grafo para este). Retorna um grafo aleatório *g*.

```
protected grafoOr randomGraf (int Npais) {
```

Criou-se o grafo *g* vazio. O número de iterações da função *while* vai ser, na prática, o número tentativas de colocar 1's na matriz de zeros *g*. Tendo por base que o número máximo de iterações teria de ser obtido a partir de uma relação de *Npais* com o número de parâmetros, decidimos que esse valor seria o *Npais \* N°* de parâmetros (*this.tLength - 1*). Geraram-se, então, posições aleatórias para a colocação de arestas (estas posições podem repetir-se). Para tal utilizámos a função *r.nextInt* (*this.tLength - 1*). Obtemos assim a entrada (ir,jr) do grafo.

```
    grafoOr g = new grafoOr (this.tLength-1);  
  
    int i = 0;  
    while (i<Npais*(this.tLength-1))  
    {  
        int ir = r.nextInt(this.tLength-1);  
        int jr = r.nextInt(this.tLength-1);
```

Para a posição aleatória criada, a função vai agora analisar se é possível adicionar aresta nessa posição. Se não exceder o n° de pais vai adicionar um 1, caso contrário vai incrementar o *i* criando uma nova posição aleatória. Após todas as iterações possíveis o método vai finalmente retornar o grafo *g* que vai estar preenchido aleatoriamente com 1's.

```
        if (g.parents(jr).size()<Npais)  
        {  
            g.add_edge(ir, jr);  
            i++;  
        }  
    }  
    return g;}
```

### 3.2.4. greedy

O método *greedy* recebe *Npais* (número máximo de pais por parâmetro) e o número de grafos aleatórios *Ngrafos* com que se vai realizar a aprendizagem. A aprendizagem consiste, na prática, em escolher de entre os grafos que resultaram do método *MDLmin*, que transforma os *Ngrafos* de forma a minimizar o seu *MDL*, aquele que apresenta o valor mais baixo de *MDL*. Esse grafo vai ser *Gmin*.

```
public grafoOr greedy (int Npais, int Ngrafos) {
```

Criámos um grafo totalmente desconexo *G0*, ou seja, em que a matriz é preenchida totalmente por zeros. Este grafo vai ser um ponto de partida, sendo um dos *Ngrafos* que vai ser submetido à aprendizagem, para além daqueles que foram criados aleatoriamente pelo método *randomGraf*.

Inicializámos também *Gmin* que vai guardar o grafo de *MDL* com o score mínimo até ao momento, começando portanto por guardar o grafo resultante método de *MDLmin* aplicado ao grafo desconexo *G0*.

Foi ainda criada a variável *MDLmin* que vai guardar o *MDL* score do grafo *Gmin*.

```
    int gDim = this.tLength-1;

    grafoOr G0 = new grafoOr(gDim);

    grafoOr Gmin = this.grafMDLmin(G0, Npais);

    double MDLmin = Gmin.MDL(this);

    double MDLaux = 0;
```

Este ciclo vai ser iniciado com  $i = 1$  uma vez que o primeiro grafo dos *Ngrafos* é o *G0*. Em cada incrementação *Gr* vai criar o grafo aleatório a partir de *Npais*, *Gaux* vai guardar o grafo que resulta de aplicar o método *MDLmin* a *Gr* e *MDLaux* vai guardar *MDL* score do grafo *Gaux*.

Este valor de *MDL* vai ser então comparado com o da entrada *i* anterior. Se o valor *MDLaux* for inferior ao de *MDLmin*, então *MDLmin* e *Gmin* passam a tomar os valores de *MDLaux* e *Gaux* respectivamente, caso contrário o *i* é incrementado passando a analisar um novo grafo dos *Ngrafos*.

No final do ciclo, após ter percorrido todos os grafos de entrada *Ngrafos*, vai retornar *Gmin* que vai ser o grafo final resultante da aprendizagem.

```
for (int i=1; i<Ngrafos; i++)
{
    grafoOr Gr = this.randomGraf(Npais);
    grafoOr Gaux = this.grafMDLmin(Gr, Npais);
    MDLaux = Gaux.MDL(this);

    if (MDLaux<MDLmin)
    {
        MDLmin=MDLaux;
        Gmin=Gaux;
    }
}
return Gmin;
}
```

### 3.2.4. Reader/Save/Load

Estes métodos não diferiram muito das do professor em termos de opções de implementação. São essenciais pelo facto de evitarem que seja necessário criar uma nova Rede de Bayes de cada vez que se corre o programa para diversos doentes de uma mesma amostra.

O método *Reader* vai construir uma amostra a partir de um ficheiro .csv onde vai estar guardada a amostra a testar.

O método *SaveBN* vai guardar a Rede de Bayes no disco.

*LoadBN* vai ler a Rede de Bayes guardada no disco permitindo a sua posterior utilização.

## 4. Alterações efectuadas

Na execução da 2ª parte do projecto foi necessário alterar algumas classes criadas na 1ª parte do mesmo.

- Métodos adicionados a classes já existentes:

Amostra - *randomGraf*; *grafMDLmin*; *greedy*

Grafos – *cycleQ*

BN – Método construtor que inicializa uma Rede de Bayes vazia.

(foi também detectado um erro na criação da rede de bayes. A má formação de um vector, devida a um erro num ciclo “for”, por vezes gerava valores de probabilidades errados)

- Alterações feitas a funções da 1ª parte do projecto:

Grafos :

add\_edge - adição de uma guarda, porque o grafo não pode ser cíclico. Utiliza o novo método *cycleQ*.

MDLdelta - adição de uma guarda. Se se fosse estudar a variação de MDL para a adição de uma aresta que iria tornar o grafo cíclico, o método retorna zero. Utiliza o novo método *cycleQ*.

- Classes novas:

inter1 – 1ª interface;

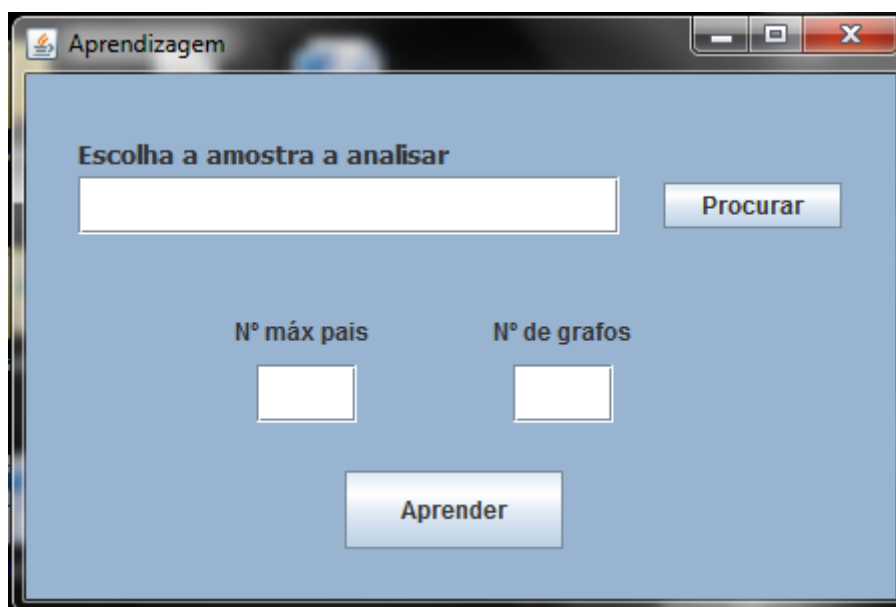
inter2 – 2ª aplicação;

RW – Métodos “Reader”, “SaveBN” e “LoadBN”.

## 5. Interfaces

Para que um utilizador possa interagir com a aplicação que se desenvolveu, criou-se duas interfaces gráficas em *Java*.

A 1ª aplicação (inter1) faz a aprendizagem. O utilizador pesquisa a localização do ficheiro .csv que vai gerar a amostra, introduz o número máximo de pais por parâmetro e o número de grafos aleatórios com que pretende realizar a aprendizagem. Obtido o melhor grafo, cria-se a Rede de Bayes, que vai ser guardada no disco para ser utilizada posteriormente na 2ª aplicação. O grafo utilizado, o caminho para o ficheiro .csv, e o nome da amostra também são guardados no disco.



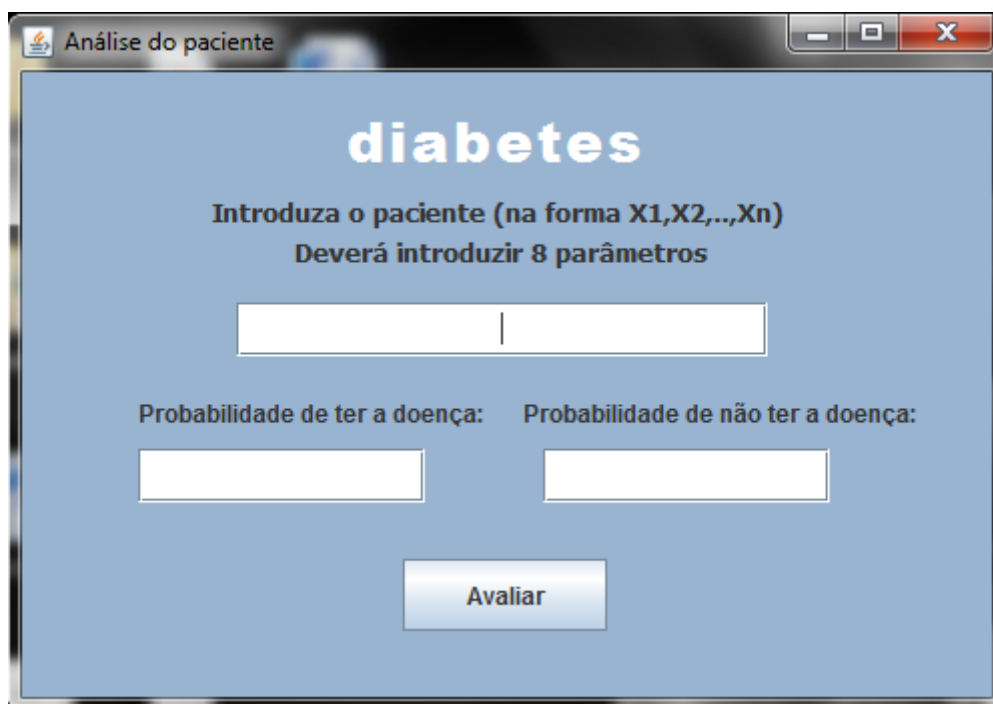
**Figura 4:** Janela de aprendizagem

Finalmente, estima-se a eficácia da Rede de Bayes criada, recorrendo aos pacientes da amostra, dos quais já é conhecido o valor do classificador (0 - não tem a doença; 1 - tem a doença). Analisando os parâmetros de um paciente da amostra, do qual se sabe que o seu classificador apresenta, por exemplo, o valor 1, já se sabe à priori que o a probabilidade de ter a doença deveria ser superior a 50%.

Analisando todos os elementos da amostra, contamos o número de vezes que os pacientes foram analisados correctamente, estimando desta forma a eficácia do diagnóstico de futuros pacientes cujos classificadores são desconhecidos. O valor da eficácia é apenas apresentado na consola, quando o projecto está a ser corrido no ambiente Eclipse.

Também é apresentado na consola o grafo que o "greedy" devolve e que foi utilizado para gerar a Rede de Bayes.)

A 2ª aplicação permite ao utilizador introduzir os parâmetros de um paciente e devolve a probabilidade de este ter e não ter a doença. (cuja soma é 100%, como era de esperar). A aplicação recupera as informações guardadas em disco pela 1ª aplicação, e calcula e apresenta essas probabilidades.



The screenshot shows a software window titled "Análise do paciente" with a blue background. At the top, the word "diabetes" is displayed in large white letters. Below it, the text "Introduza o paciente (na forma X1,X2,...,Xn)" and "Deverá introduzir 8 parâmetros" is shown. There is a single-line text input field. Below this, two labels are present: "Probabilidade de ter a doença:" and "Probabilidade de não ter a doença:", each followed by a text input field. At the bottom center, there is a button labeled "Avaliar".

**Figura 5:** Janela de introdução do doente e de resultados de diagnóstico



## 6. Experiências

Nesta parte apresentamos os testes realizados à aplicação para as doenças tiróide, diabetes e cancro.

### 6.1 Tiróide

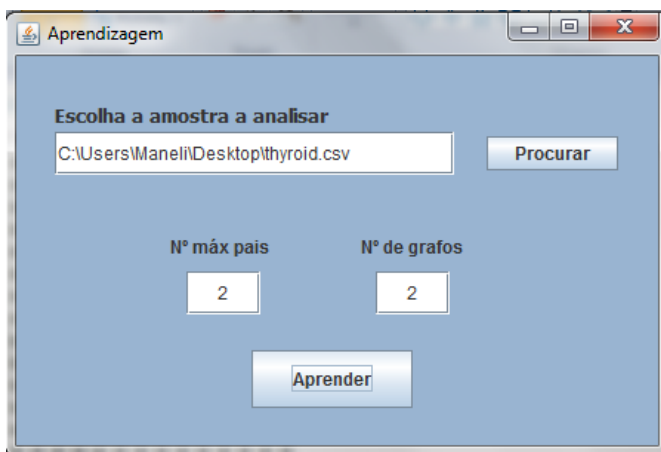


Figura 6: Janela de aprendizagem

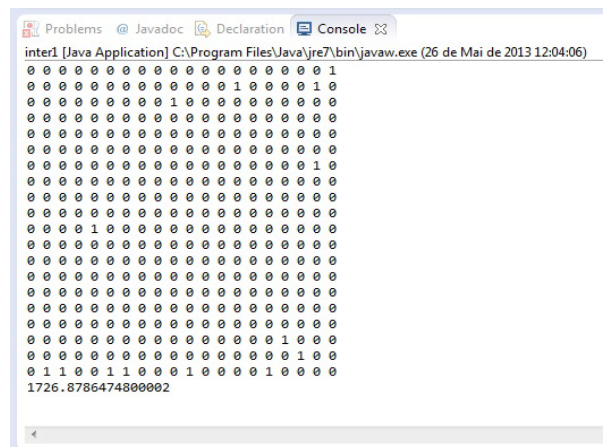


Figura 7: Grafo de MDL mínimo que gera a rede de Bayes

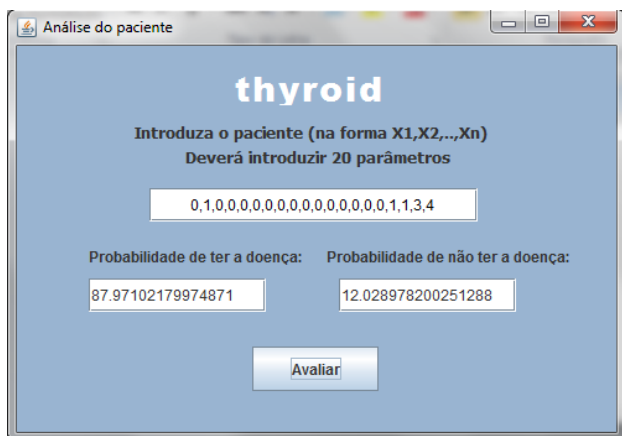


Figura 8: Resultados obtidos para o paciente de parâmetros 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,3,4,(1)

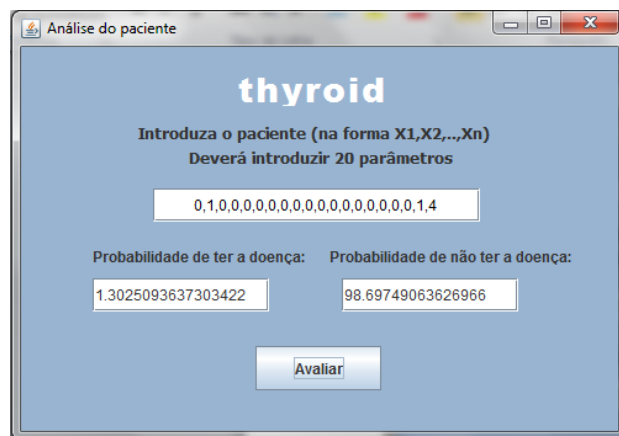


Figura 9: Resultados obtidos para o paciente de parâmetros 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,4,(0)

## 6.2 Hepatite

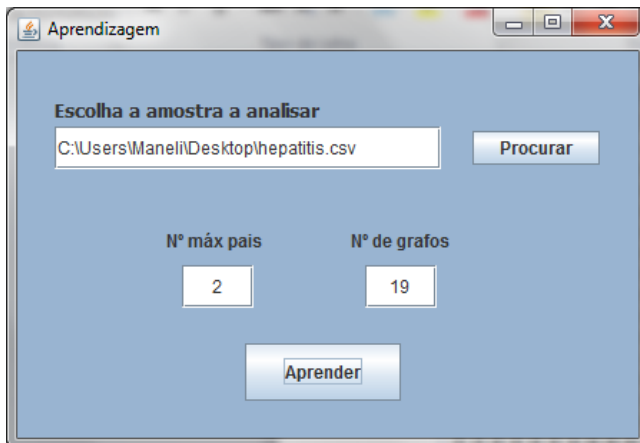


Figura 10: Janela de aprendizagem

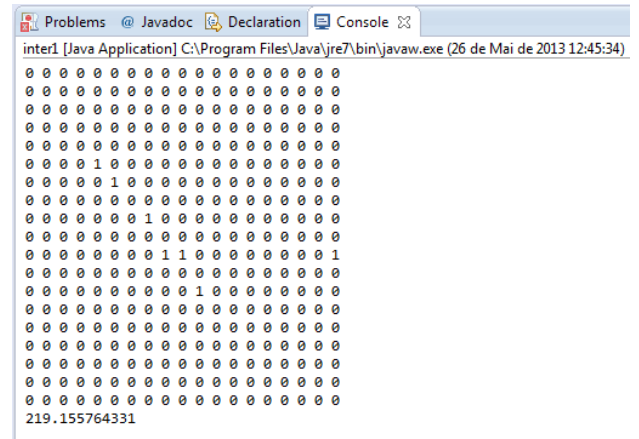


Figura 11: Grafo de MDL mínimo que gera a rede de Bayes

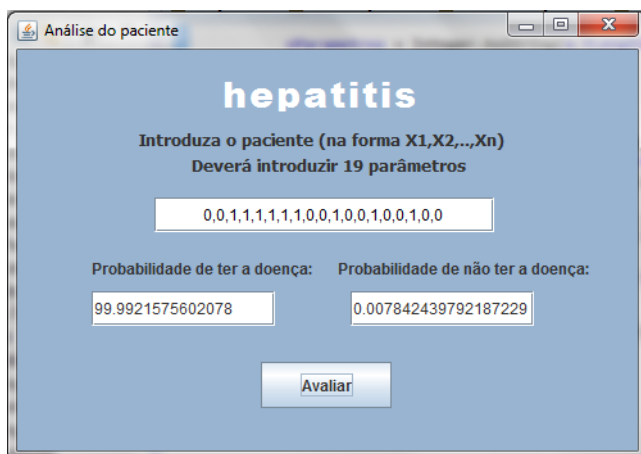


Figura 12: Resultados obtidos para o paciente de parâmetros 0,0,1,1,1,1,1,1,0,0,1,0,0,1,0,0,1,0,0,(1)



Figura 13: Resultados obtidos para o paciente de parâmetros 0,0,0,0,1,1,0,1,1,1,1,0,0,1,0,0,0,1,1,(0)

## 6.3 Cancro da mama

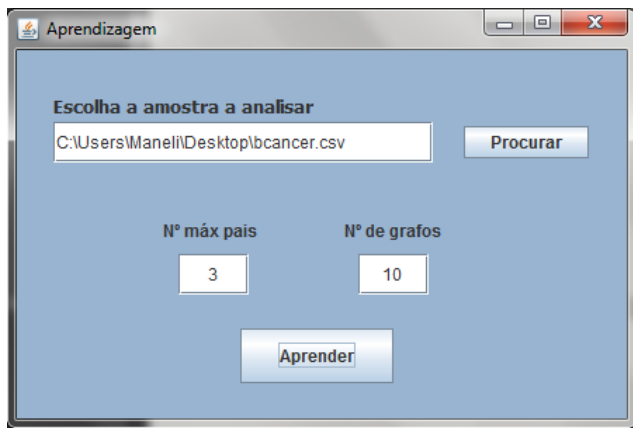


Figura 14: Janela de aprendizagem

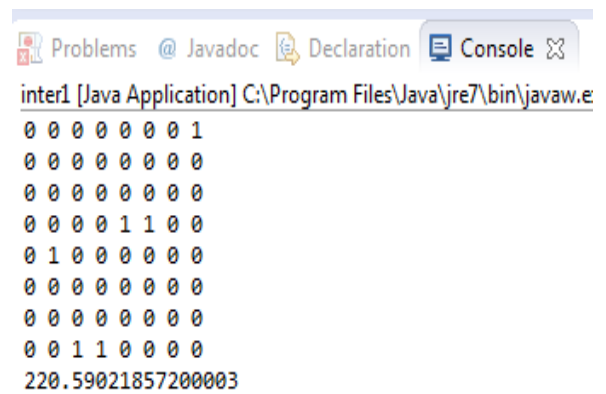


Figura 15: Grafo de MDL mínimo que gera a rede de Bayes

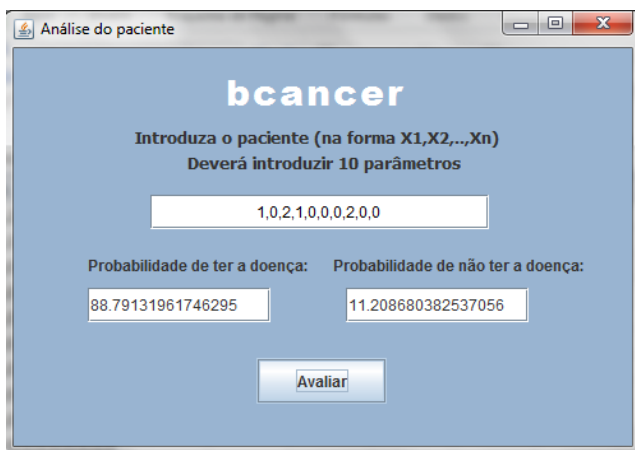


Figura 16: Resultados obtidos para o paciente de parâmetros 1,0,2,1,0,0,0,2,0,0,(1)

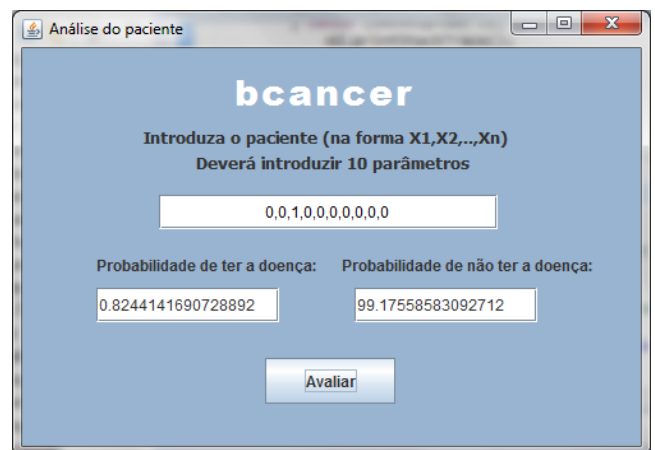


Figura 17: Resultados obtidos para o paciente de parâmetros 0,0,1,0,0,0,0,0,0,0,(0)

## 6.4 Diabetes

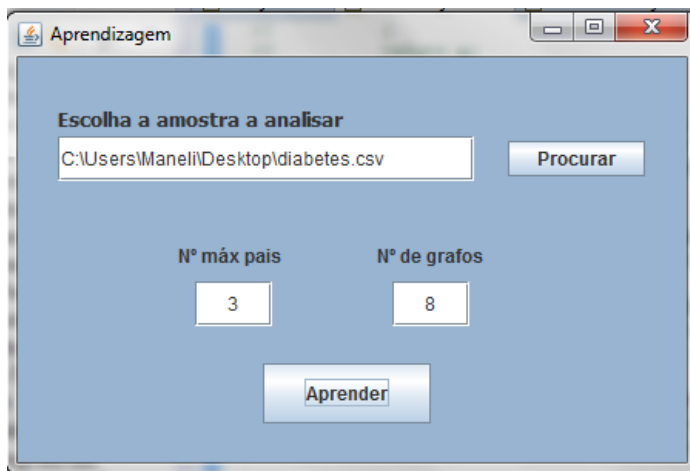


Figura 17: Janela de aprendizagem

```
Problems @ Javadoc Declaration
inter1 [Java Application] C:\Program Files\J
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0
220.59021857200003
```

Figura 18: Grafo de MDL mínimo que gera a rede de Ba

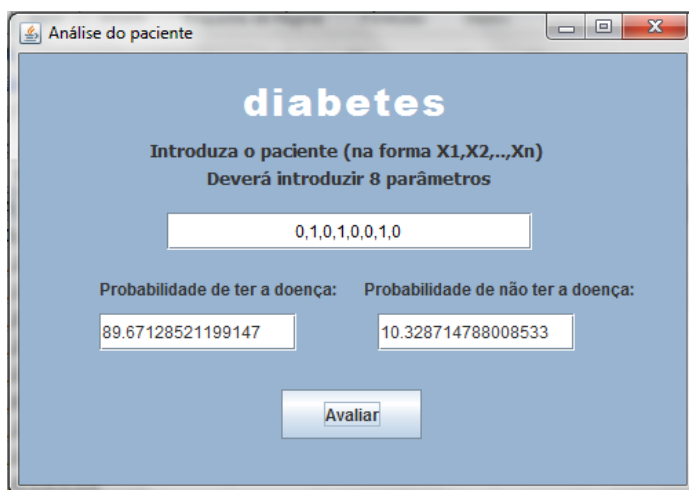


Figura 19: Resultados obtidos para o paciente de parâmetros 0,1,0,1,0,0,1,0,(1)

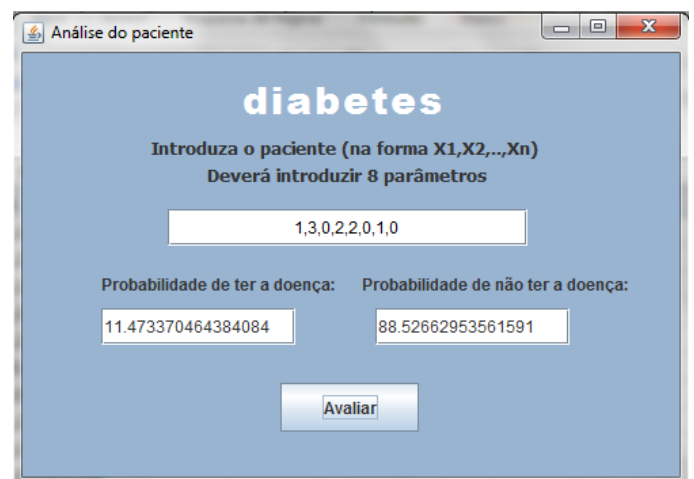


Figura 20: Resultados obtidos para o paciente de parâmetros 1,3,0,2,2,0,1,0,(0)

## 7. Conclusões e críticas

- Algumas das características mais importantes do programa são a eficácia e rapidez dos métodos. Estas foram preocupações que tivemos ao longo da realização do projecto sendo que por vezes podemos não ter conseguido criar os métodos da forma ideal.

Esta questão da duração prende-se principalmente por funções como a *Count*, *MDLdelta* e *grafMDLmin* que são funções chamadas de forma recorrente e/ou que pela sua complexidade têm naturalmente uma duração mais longa que as restantes.

- Verificámos que a criação de guardas nas funções tem grande importância para prevenir o maior número possível de erros e situações indesejadas. Por exemplo, no método *P<sub>r</sub>Total*, tivemos de garantir que a  $\frac{P_{rT}(x,y,c)P_{rT}(c)}{P_{rT}(y,c)P_{rT}(x,c)}$  não é uma indeterminação, caso contrário poderia devolver resultados incorrectos.
- Os resultados obtidos nas experiências estavam de acordo com o esperado. Foram analisados diversos pacientes, cujo diagnóstico era previamente conhecido, e verificou-se que a aplicação apresentava a correcta avaliação dos mesmos.
- Foi criado o método *eficacia* de maneira a perceber até que ponto o nosso projecto apresenta o correcto diagnóstico de doença.
- As maiores dificuldades que tivemos foram essencialmente na interpretação do guia, que impediram uma total autonomia, uma vez que levantaram uma série de questões em termos do que tínhamos de facto de implementar. Sugerimos, como possível solução, que o guia seja mais claro, incluindo mais instruções que nos direcionem de forma mais eficaz na construção do código. Podia ainda abranger mais informação em relação aos conteúdos teóricos aplicados, permitindo uma total compreensão dos conceitos utilizados (exemplo: Rede de Bayes).

- O projecto despertou interesse e motivação tanto pela íntima relação com o nosso curso, como pelo desafio criado. Foi uma forma de pôr em prática as nossas capacidades de programação em *Java*, permitindo de facto ver as suas potencialidades, e foi também uma óptima base para futuros projectos que poderemos vir a realizar no âmbito da Engenharia Biomédica.