Instituto de Ciências Matemáticas de São Carlos

ISSN - 0109-2569

Descrição e Implementação de um Algoritmo Indutivo Incremental para Aprendizado de Árvores de Decisão Binárias[1]

CLAUDIA APARECIDA MARTINS

MARIA CAROLINA MONARD

Nº 30

RELATÓRIOS TÉCNICOS DO ICMSC

SÃO CARLOS
Nov. / 1994

SYSNO 872881
DATA / /
ICMC - SBAB

Descrição e Implementação de um Algoritmo Indutivo Incremental para Aprendizado de Árvores de Decisão Binárias[1]

Claudia Aparecida Martins
Universidade de São Paulo
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP
e-mail: cam@icmsc.sc.usp.br

Maria Carolina Monard
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP
e-mail: mcmonard@icmsc.sc.usp.br

Sumário

Um dos paradigmas mais amplamente pesquisados em Aprendizado de Máquina, na área de Inteligência Artificial, é o *Aprendizado Indutivo por Exemplos*. Este paradigma infere uma descrição geral do conceito, a partir de um conjunto de exemplos fornecidos por um professor ou pelo ambiente. O conjunto de exemplos, assim como o conceito induzido, necessitam de um formalismo — linguagem de descrição — para representá-los.
Árvore de Decisão é um dos formalismos frequentemente utilizados quando os exemplos são descritos em função de atributos e seus respectivos valores. A indução de uma árvore de decisão pode ser realizada no modo incremental ou não incremental. No modo não incremental o algoritmo infere o conceito usando todo conjunto de instâncias de exemplos de treinamento disponíveis para construir a árvore. No modo incremental, o algoritmo revê a definicão do conceito corrente, se necessário, cada vez que uma nova instância de treinamento for processada.

Neste trabalho é apresentada e discutida em detalhes a implementação de um sistema que constrói árvores de decisão de modo incremental, a partir de um conjunto de exemplos expressos através de atributos binários, para realizar aprendizado indutivo no modo incremental.

Novembro de 1994

[1]Trabalho realizado com o auxílio do CNPq e FAPESP de acordo com o Proc. Nº 92/1586-0 e Proc. Nº 92/2151-8.

Conteúdo

1 Introdução

A aquisição de conhecimento é uma das tarefas mais difíceis, porém uma das principais para o desenvolvimento de Sistemas Baseados em Conhecimento (SBC). Aprendizado de Máquina (AM) é uma área de pesquisa em Inteligência Artificial (IA), que auxilia na automação do processo de aquisição de conhecimento, através de estudos de métodos computacionais para aquisição de novos conhecimentos, novas habilidades e novos meios de organizar o conhecimento existente.

Um dos paradigmas mais amplamente pesquisado em Aprendizado de Máquina, na área de IA, é o *Aprendizado Indutivo por Exemplos*. Este paradigma infere uma descrição geral do conceito a partir de um conjunto de exemplos, fornecidos por um professor ou pelo ambiente. O conjunto de exemplos, assim como o conceito induzido, necessitam de um formalismo — linguagem de descrição — para representá-los.

Árvore de Decisão é um formalismo frequentemente utilizado para a representação do conceito aprendido quando, numa situação de aprendizado, os exemplos são descritos em função de atributos e seus respectivos valores. A indução de uma árvore de decisão pode ser realizada no modo não incremental ou no incremental. No modo não incremental o algoritmo infere o conceito baseado no conjunto total de instâncias de exemplos de treinamento disponíveis para construir a árvore. No modo incremental, o algoritmo revê a definicão do conceito corrente, quando necessário, cada vez que uma nova instância de treinamento for processada.

No ICMSC-USP está sendo desenvolvido um ambiente de AM que tem por objetivo gerar regras de conhecimento através de árvores de decisão utilizando a estratégia do aprendizado indutivo por exemplos dos sistemas da família TDIDT — Top Down Induction of Decision Trees. O algoritmo geral dos sistemas da família TDIDT começa com uma árvore de decisão vazia e, através de um conjunto de exemplos de aprendizado, chamados de instâncias de treinamento, de uma condição de parada e de uma função de avaliação, que calcula o atributo com maior ganho de informação, constrói uma árvore de decisão e a refina até que classifique corretamente todos (ou a maioria) dos exemplos do conjunto.

Dentro deste projeto já foram realizadas diversas implementações, na linguagem de programação Prolog, que utilizam um procedimento geral não incremental para construir a árvore de decisão. Esse procedimento geral é tal que dado um conjunto de exemplos de treinamento, com qualquer número de atributos, bem como qualquer número de valores discretos de atributos, constrói a árvore de decisão correspondente [Castineira 91]. Deve ser observado que a estrutura dinâmica de dados que representa essa árvore de decisão geral, bem como as técnicas de programação utilizadas, são bastante complexas.

Com base nesta árvore de decisão geral, diversas implementações foram desenvolvidas, tais como

- mecanismo de janela e pós-poda da árvore de decisão [Nicoletti 92]
- algoritmos de aprendizado construtivo para a construção automática de *features* [Nicoletti 92a,Nicoletti 92,Nicoletti 93a,Nicoletti 93b,Nicoletti 93c,Nicoletti 94].

Neste último caso — aprendizado construtivo para a construção automática de features — a informação que se tem com relação aos atributos é que eles podem assumir apenas dois valores possíveis, isto é, os atributos estão presentes ou ausentes no conjunto de treinamento. Com isso, a árvore de decisão obtida é binária, resultando em uma estrutura de dados mais simples. Devido à esta situação, as implementações correspondentes à construção e poda da árvore, bem como a construção automática de *features* utilizando diversas estratégias, são menos elaboradas. Assim, foi decidido implementar algoritmos semelhantes mas utilizando a estrutura de árvore binária. Utilizando essa estrutura diversas implementações foram realizadas, tais como

- construção no modo não incremental de árvores de decisão binárias [Martins 94]
- gerador de exemplos de funções booleanas [Monard 94a]
- implementação de algoritmos para construção automática de features [Monard 94b].

O objetivo deste trabalho é descrever uma implementação que constrói uma árvore de decisão binária, no modo incremental, utilizando a metodologia TDIDT. Como o algoritmo implementado utiliza a abordagem incremental. a árvore de decisão deve ser construída exemplo a exemplo — a cada novo exemplo a árvore resultante é considerada a árvore final. À medida que novos exemplos vão se tornando disponíveis, a árvore vai sendo revista e reestruturada de maneira a classificá-los. Assim, a árvore vai se alterando e crescendo à medida que os exemplos se tornam disponíveis.

Este trabalho está organizado da seguinte forma: a seção 2 apresenta uma introdução geral sobre indução de árvores de decisão onde são apresentados os algoritmos ID5 e ID5R, da família TDIDT. A seção 3 descreve a implementação da construção da árvore de decisão através de exemplos para o algoritmo específico que utiliza o modo incremental. A seção 4 exemplifica o uso da implementação construída. seção 5 mostra a listagem completa do algoritmo implementado e, finalmente, na seção 6 são apresentadas as conclusões.

2 Indução de Árvores de Decisão

Aprendizado indutivo é o processo de inferência indutiva através de fatos fornecidos por um professor ou ambiente. Um tipo especial de aprendizado indutivo é o *aprendizado de conceitos através de exemplos*, cuja tarefa é induzir descrições gerais de conceitos através de instâncias específicas destes conceitos [Michalski 83]. Em todo processo o de aprendizado é necessário um formalismo que descreva objetos e conceitos. Um dos formalismos existentes para descrever conceitos, quando os objetos são descritos pelos seus atributos, é o da Árvore de Decisão. Uma árvore de decisão é uma representação de um procedimento de decisão para determinar a classe de uma dada instância. Cada nó da árvore especifica ou um nome de classe ou um teste específico que particiona o espaço de instâncias no nó, de acordo com os valores possíveis do teste. Cada subconjunto da partição corresponde a um subproblema de classificação para aquele subespaço de instâncias, que é resolvido por uma subárvore [Utgoff 89].

Formalmente, pode-se definir uma árvore de decisão como:

1. um nó folha (ou nó resposta) que contém um nome de classe, ou
2. um nó não folha (ou nó de decisão) que contém um atributo teste com uma ramificação para outra árvore de decisão para cada valor possível do atributo.

2.1 Modo Incremental e Não Incremental

A indução de árvores de decisão através de exemplos pode ser feita em duas abordagens:

1. modo não incremental
2. modo incremental.

No modo não incremental, o algoritmo infere o conceito de uma vez, baseado no conjunto total de instâncias de treinamento disponíveis. No modo incremental, o algoritmo revê a definição do conceito corrente, se necessário, em resposta a cada nova instância de treinamento observada. Uma das vantagens de usar um algoritmo incremental é que o conhecimento pode ser rapidamente atualizado a cada nova observação e é mais eficiente revisar uma hipótese existente do que gerar uma hipótese cada vez que uma nova instância é observada [Utgoff 89].

Como já mencionado, a família TDIDT — Top Down Induction of Decision Tree — é uma família de sistemas de aprendizado, que é caracterizada por representar o conhecimento adquirido através de árvores de decisão. São sistemas de propósito geral cujo objetivo é classificar objetos. Estes sistemas produzem regras ou descrições de um determinado número de classes de objetos. Quando novos objetos são observados, estas regras devem predizer a que classe os objetos pertencem.

Todos os sistemas existentes pertencentes à família TDIDT são sucessores do sistema CLS [Hunt 66]. O CLS foi o primeiro software construído com o objetivo de generalizar automaticamente regras a partir de exemplos. ID3 [Quinlan 79] é um dos mais conhecidos algoritmos da família TDIDT. O ID3 constrói a árvore de decisão no modo não incremental onde utiliza um conjunto de exemplos para determinar o conceito. Outros algoritmos, como ID4 [Schlimmer 86], ID5 e ID5R [Utgoff 88] são versões do ID3, para torná-lo um algoritmo incremental. No algoritmo ID4 a árvore de decisão vai se adequando às instâncias de treinamento, à medida que estas vão se tornando disponíveis. A principal desvantagem deste algoritmo está na eliminação de toda ou parte da árvore de decisão sempre que, durante sua construção, um atributo de teste for substituído por um atributo considerado mais informativo. ID5 e ID5R serão comentados na próxima seção porque são a base deste trabalho. Existem vários outros sistemas pertencentes à esta família que não serão descritos aqui.

2.2 O algoritmo ID5 e ID5R

Os algoritmos ID5 e ID5R [Utgoff 88], assim como ID4 [Schlimmer 86] são algoritmos incrementais sucessores do algoritmo ID3.

O algoritmo ID5R é um algoritmo que constrói uma árvore de decisão similar àquela construída pelo ID3, para um dado conjunto de instâncias de treinamento. Assim como o ID4 e ID5, o algoritmo ID5R mantém informações suficientes para calcular o *E-score*[2] de um dado atributo em um nó, possibilitando a troca do atributo teste do nó, caso necessário. Quando ID5R muda o atributo teste, ele não descarta toda subárvore abaixo do nó, como o faz o algoritmo ID4; o ID5R reestrutura a árvore fazendo com que o atributo teste desejado seja a raiz da árvore. O processo de reestruturação, chamado *pull-up*, é um processo de rearranjo da árvore que preserva consistência com as instâncias de treinamento observadas, colocando o atributo desejado como raiz da árvore ou subárvore. A vantagem do processo de reestruturação consiste em permitir a recontagem das classes positivas e negativas para cada valor do atributo, durante o rearranjo da árvore, sem o reprocessamento das instâncias de treinamento.

Formalmente, uma árvore de decisão ID5R é definida como sendo:

1. um nó folha (ou nó resposta) que contém
   (a) o valor da classe, e
   (b) o conjunto de descrições de instâncias naquele nó, que pertencem à classe
2. um nó não folha (ou nó de decisão) que contém
   (a) um atributo de teste que, para cada valor possível, tem ramificações para outra árvore de decisão, bem como contagem dos exemplos nas classes positiva e negativa, para cada um dos possíveis valores do atributo de teste
   (b) o conjunto de atributos não-teste no nó, cada um com contagens positiva e negativa para cada possível valor do atributo.

Esta forma de árvore de decisão difere das adotadas pelo ID3 e ID4, porque retém nela instâncias de treinamento, bem como as contagens de classes. Consequentemente, a árvore deverá ser interpretada diferentemente. Quando utilizada para classificar uma nova instância, a árvore é percorrida até que seja encontrado um nó onde todas as instâncias de treinamento pertencem à mesma classe. Este nó pode ser um nó folha ou um nó não folha. Quando for um nó não folha, deverá apenas existir uma classe com um contador de instância não-zero.

O algoritmo básico ID5R, consiste dos seguintes passos:

If a árvore está vazia

[2] *E-score* é o valor da função de avaliação utilizada na busca do atributo que tem o maior ganho de informação.

Then define-a como uma forma não expandida[3], setando o nome da classe para a classe da instância, e o conjunto de instâncias para um simples conjunto contendo a instância.

Else

If a árvore está na forma não expandida e a instância é da mesma classe

Then adicione a instância ao conjunto de instâncias mantidas no nó.

Else

If a árvore está na forma não expandida

Then expanda-a em um nível, escolhendo, arbitrariamente, um atributo teste para ser a raiz.

Para o atributo teste e todos os atributos não teste no nó corrente, atualize a contagem de instâncias positivas e negativas para o valor daquele atributo na instância de treinamento.

If o nó corrente contém um atributo teste que não tem a menor entropia

Then

(a) Reestruture a árvore (*pull-up*) para que o atributo com menor entropia fique na raiz.

(b) Recursivamente reestabeleça o melhor atributo teste em cada subárvore exceto naquela que será atualizada no próximo passo.

Endif

Endif

Recursivamente atualize a árvore de decisão abaixo do nó de decisão corrente pertencente ao ramo para o valor do atributo teste que ocorre na descrição da instância. Aumente o ramo se necessário.

Endif

O algoritmo *pull-up* de ID5R consiste dos seguintes passos:

If o atributo a_{new} a ser "empurrado" já é a raiz da árvore

Then pare

Else

1. Recursivamente empurre o atributo a_{new} para a raiz de cada subárvore imediata. Converta qualquer árvore não expandida para a forma expandida caso necessário, escolhendo o atributo a_{new} como o atributo teste.

[3]Chama-se um nó folha de uma árvore não expandida e um nó não folha de uma árvore expandida.

2. Transforme a árvore, resultando em uma nova árvore com a_{new} na raiz, e o antigo atributo raiz a_{old} na raiz de cada subárvore imediata.

Endif

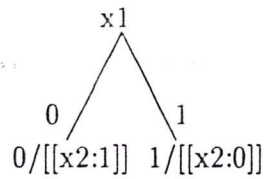
Para efeito de ilustração do algoritmo considere o seguinte conjunto de exemplos:

Atributos		**Classe**
x1	**x2**	
0	1	0
1	0	1
0	0	1
1	1	0

O algoritmo de construção da árvore irá processar cada instância individualmente. Para a primeira instância (x1:0,x2:1.classe:0), a árvore resultante consiste de um simples nó folha

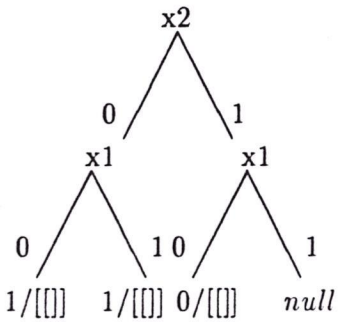
0/[[x1:0,x2:1]]

Na segunda instância (x1:1,x2:0,classe:1), a classe é positiva, então a árvore é expandida escolhendo *x1*, arbitrariamente, para ser a raiz da árvore.

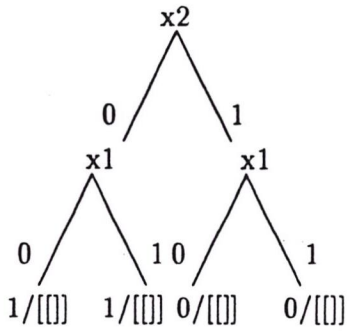


A representação gráfica da árvore de decisão apresentada não mostra a lista de contadores associada a cada nó de decisão. Estes são criados no momento da expansão da árvore e atualizados a cada nova instância classificada pelo nó.

Quando se realiza o processamento da terceira instância (x1:0,x2:0,classe:1), verifica-se que o atributo x2 contém o menor valor de entropia, assim a árvore deve ser reestruturada. O objetivo, então, é fazer com que o atributo x2, seja a raiz da árvore. Após este processo, a árvore resultante será:



Por fim, o último exemplo (x1:1,x2:1,classe:0), é classificado pela árvore sem causar uma reestruturação na mesma. A árvore final é representada por:



O algoritmo ID5R é sucessor do algoritmo ID5. A diferença entre os dois algoritmos reside no passo (b) do algoritmo ID5R, pg. 4, que é omitido no algoritmo ID5. Isto significa que, após reestruturar a árvore para trazer o atributo desejado para a raiz, a subárvore que não classifica o exemplo não será reestruturada recursivamente, ou seja, a raiz desta subárvore poderá não ter o melhor atributo. Eliminando-se este passo, o custo para manipulação da árvore é reduzido. Entretanto, a consequência dessa omissão é que nada garante que a árvore resultante será equivalente àquela que ID3 construirá para o mesmo conjunto de treinamento. Outra imposição para que as árvores, construídas pelo ID3 e pelo ID5R, sejam iguais, é que ambos algoritmos devem dar o mesmo tratamento na escolha entre dois atributos com o mesmo valor da entropia.

3 Descrição da Implementação

A implementação realizada neste trabalho tem como objetivo construir uma árvore de decisão binária no modo incremental baseada no algoritmo ID5. Como visto anteriormente, este algoritmo considera que os exemplos do conjunto de treinamento são fornecidos um a um. Na classificação de um novo exemplo na árvore de decisão, para cada nó da árvore, busca-se o atributo com o maior ganho de informação e reestrutura-se a árvore caso o atributo seja diferente do atributo do nó corrente. É utilizada uma função de avaliação para calcular o atributo que tem o maior ganho de informação dos exemplos fornecidos até o momento. A função de avaliação utilizada nesta implementação é a entropia.

> *Entropia de classificação é usada em teoria da informação e interpretada como a quantidade de informação contida em uma mensagem. Quanto maior for o valor da entropia de classificação, maior a incerteza com relação ao conteúdo da mensagem. Neste contexto, uma árvore de decisão para a classificação de exemplos pode ser vista como uma fonte de informação que, dado um exemplo, gera uma mensagem indicando a classificação do exemplo. Quando o nó da árvore contém apenas exemplos da mesma classe, a entropia é igual a 0 — significando que a decisão de classificação é definitiva para exemplos pertencentes àquele nó [Shaw 90].*

3.1 Estrutura de Dados

A implementação foi realizada na linguagem de programação Prolog [Arity 92] utilizando listas e árvores binárias como estrutura de dados. Durante o processo de construção e atualização da árvore de decisão incremental, a seguinte estrutura é utilizada

arv(SubArvEsq,Raiz/ListaContadores,SubArvDir)

onde a árvore de decisão é representada pelo

- nó raiz e uma lista de contadores associada — *Raiz/ListaContadores*
- a subárvore esquerda — *SubArvEsq* — para a ramificação cujo valor da raiz é 0, e
- a subárvore direita — *SubArvDir* — para a ramificação cujo valor da raiz é 1.

Cada subárvore pode ser outra árvore de decisão ou um nó folha rotulado com o valor da classe e o conjunto de exemplos associado. É válido lembrar que a árvore poderá ser representada com o átomo *null*, quando estiver vazia, ou representada através de um nó folha rotulado com o valor da classe, no caso de todos os exemplos pertencerem à mesma classe.

A árvore de decisão retém informações suficientes para calcular a entropia de cada atributo em um determinado nó. Estas informações são armazenadas em uma lista — *ListaContadores* — do tipo

$$[[At_1, A_1 : B_1, C_1 : D_1], [At_2, A_2 : B_2, C_2 : D_2], ..., [At_n, A_n : B_n, C_n : D_n]]$$

onde A_i, B_i, C_i, D_i para $i = 1, 2, ...n$ são contadores de exemplos positivos e negativos para cada valor possível do atributo At_i que pertence àquele nó. Isto é, $A_i : B_i$ representam contadores de exemplos cuja classe é 0 e valor do atributo é 0 (A_1) e 1 (B_1), e $C_1 : D_1$ contadores cuja classe é 1 e o valor do atributo é 0 (C_1) e 1 (D_1), respectivamente. A árvore retém, também, informação referente a instâncias de treinamento nos nós folhas. Essas instâncias de treinamento são utilizadas para expandir a árvore ou para auxiliar na contagem dos atributos na lista de contadores.

O procedimento responsável pela construção incremental da árvore de decisão, ilustrado na Figura 1, tem como entrada um exemplo a ser inserido na árvore e a árvore anteriormente construída pelo algoritmo na forma acima descrita.

No caso de não existir a árvore de entrada, ou seja, a primeira vez que o procedimento vai ser ativado, a árvore anterior pode ser representada por uma variável livre ou o átomo *null*.

Um exemplo é representado por uma lista cujos elementos são estruturas da forma

$< atributo >:< valor >$

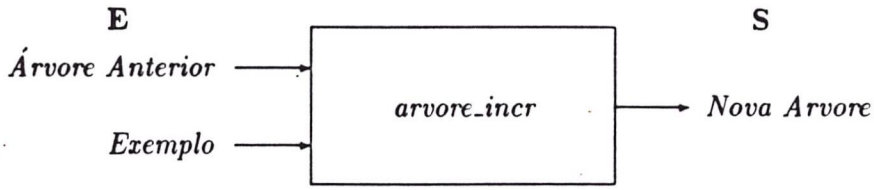


Figura 1: Procedimento Responsável pela Construção da Árvore de Decisão Incremental

ou

classe :< *valor* >

Assim, os exemplos da tabela

Atributos			Classe
x1	x2	x3	
sim	sim	sim	sim
não	não	não	sim
sim	não	não	não
sim	não	sim	sim

a lista utilizada na implementação, correspondente ao exemplo são representados por

```
[x1:1,x2:1,x3:1,classe:1]
[x1:0,x2:0,x3:0,classe:1]
[x1:1,x2:0,x3:0,classe:0]
[x1:1,x2:0,x3:1,classe:1]
```

A ordem dos elementos na lista é irrelevante. A implementação realizada tenta classificar o exemplo independente dele ter o mesmo número de atributos que aqueles que compareceram na árvore anteriormente construída. Se o número de atributos do exemplo for maior, existe a possibilidade de não poder ser inserido nessa árvore e, nesse caso, o exemplo é ignorado. Caso o exemplo possa ser inserido na árvore, os novos atributos são adicionados nos nós folhas. Portanto, é recomendável que todos os atributos compareçam em todos os exemplos.

3.2 Procedimentos: Definição e Discussão

Em Prolog, um *procedimento* consiste do conjunto de cláusulas referente à uma mesma relação ou predicado. Predicados são notados pelo seu nome e pela sua aridade. Por exemplo, a notação `abc/4` refere-se ao predicado de nome `abc` que tem 4 argumentos,

ou seja, aridade 4. Esta convenção é utilizada neste texto. Além disso, quando for conveniente, os argumentos dos procedimentos serão descritos da seguinte forma[4]:

[+] $< arg_n >$: o n-ésimo argumento é de entrada

[−] $< arg_n >$: o n-ésimo argumento é de saída

[?] $< arg_n >$: o n-ésimo argumento é de entrada/saída

O procedimento **arvore_incr/3**, descrito a seguir, é o procedimento principal para a construção e atualização da árvore de decisão binária.

Procedimento 3.2.1 `arvore_incr/3`

Os argumentos de `arvore_incr/3` são:

[+] $< arg_1 >$: exemplo a ser classificado

[+] $< arg_2 >$: árvore de decisão a ser atualizada

[−] $< arg_3 >$: árvore de decisão atualizada com o exemplo

[4]Observar que a instanciação dos argumentos, descrita neste trabalho, refere-se ao caso da árvore estar sendo construída.

```
arvore_incr([],Arv,Arv).

arvore_incr(Exemplo,null,Classe/[E1]) :-
    verifica_at(Exemplo,classe,Classe),
    retira_at(Exemplo,classe,E1),
    !.

arvore_incr(Exemplo,Classe/E,Classe/E1) :-
    atomic(Classe),
    verifica_at(Exemplo,classe,Classe),
    retira_at(Exemplo,classe,Ex),
    insere_ex(Ex,E,E1),
    !.

arvore_incr(Exemplo,Classe/E,Arv) :-
    atomic(Classe),
    verifica_at(Exemplo,classe,Classe1),
    pega_at(E,At),
    criar_subarv(At,Classe/E,ArvE),
    arvore_incr(Exemplo,ArvE,Arv,F),
    !.

arvore_incr(Exemplo,arv(Se,Raiz/L,Sd),Arv) :-
    incrementa_lista(Exemplo,L,LI),
    calcule_entropia(LI,[Raiz1,_]),
    pull_up1(Raiz1,arv(Se,Raiz/L,Sd),arv(Sep,Raiz1/L,Sdp)),
    verifica_at(Exemplo,Raiz1,Val),
    retira_at(Exemplo,Raiz1,E1),
    ifthenelse(Val=0,
        (arvore_incr(E1,Sep,Se1,F),Arv=arv(Se1,Raiz1/LI,Sdp)),
        (arvore_incr(E1,Sdp,Sd1,F),Arv=arv(Sep,Raiz1/LI,Sd1))),
    !.
```

A primeira cláusula trata a situação onde não existe nenhum exemplo a ser classificado. Neste caso, a árvore de decisão não é modificada. Nas próximas cláusulas o procedimento irá tratar casos particulares da estrutura que representa a árvore de decisão no argumento de entrada. Estas estruturas são:

- a árvore de decisão pode ser uma árvore vazia, neste caso ela pode estar rotulada com o átomo `null` ou ser uma variável livre que se instância com `null` — cláusula 2.
- a árvore de decisão consiste de um simples nó rotulado com o valor do atributo `classe` — cláusula 3 e 4.
- a árvore de decisão tem a estrutura `arv(Se,Raiz/L,Sd)` — cláusula 5.

Quando o procedimento encontra o átomo `null` — cláusula 2 — representando a árvore de decisão, seu objetivo é retornar um nó rotulado com o valor da classe juntamente com o exemplo sem o atributo `classe`. A árvore de decisão é representada, simplesmente, por esse nó folha.

Por exemplo, a interrogação

```
?- arvore_incr([x1:0,x2:1,x3:0,classe:1],null,Arvore).
```

tem como resultado

```
Arvore = 1/[[x1:0,x2:1,x3:0]]
```

Quando a árvore é representada por um simples nó rotulado com o valor do atributo `classe` — cláusula 3 e 4 — e os vários exemplos pertencentes àquele nó, duas situações podem ocorrer. Na primeira situação — cláusula 3 —, o valor do atributo `classe` do exemplo a ser classificado é igual ao valor da classe rotulado no nó que representa a árvore de decisão. Neste caso o exemplo é, simplesmente, adicionado à lista de exemplos, sem o atributo `classe` que o nó possui. Caso os valores das classes sejam diferentes — a classe do exemplo diferente do valor contido no nó —, um atributo é escolhido, aleatoriamente, no conjunto de exemplos armazenados naquele nó e são criadas subárvores para este atributo através do predicado `criar_subarv/3` — cláusula 4. A árvore que antes consistia de apenas um nó é expandida em uma árvore com a seguinte estrutura

$$arv(SubArvEsq, Raiz/Lista, SubArvDir)$$

onde

- *Raiz* é o atributo escolhido aleatoriamente para ser a raiz da árvore
- *Lista* é a lista de atributos com contadores para exemplos positivos e negativos para cada valor do atributo, positivo e negativo, e
- *SubArvEsq* e *SubArvDir* são as subárvores esquerda e direita de *Raiz* onde os exemplos são particionados, respectivamente, de acordo com os valores 0 e 1 do atributo *Raiz*.

Por exemplo, a interrogação

```
?- criar_subarv(At,Classe/E,ArvExp).
```

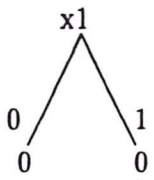
ativada com

```
At = x1

Classe/E = 0/[[x1:1,x2:0],[x1:0,x2:0]]
```

é bem sucedida com

```
ArvExp = arv(0/[[x2:0]],x1/[[x1,1:0,1:0],[x2:2:0,0:0]],0/[[x2:0]])
```

Árvore Gráfica[5]



Com a árvore expandida em um nível, o processo de classificar o exemplo é repetido, agora com a nova árvore.[6]

A última cláusula de `arvore_incr/3` trata a situação onde o exemplo será classificado em uma árvore de decisão com a seguinte estrutura `arv(Se,Raiz/L,Sd)`. O primeiro passo a ser realizado é atualizar a lista que faz a contagem dos elementos positivos e negativos, de acordo com o novo exemplo. É através desta lista que será calculada a entropia de cada atributo. O procedimento `calcule_entropia/2` retornará o atributo com o menor valor da entropia. O procedimento `pull_up1/3` é responsável por realizar o processo de reestruturação da árvore, caso seja necessário. Este processo de reestruturação consiste em fazer com que o melhor atributo — `MelhorAt` —, aquele que tiver o menor valor de entropia, se torne a raiz da árvore. Após todo este processo, é verificado o valor de `MelhorAt` em `Exemplo` para que o processo seja repetido na subárvore esquerda ou direita, respectivamente, dependendo do valor de `MelhorAt` ser 0 ou 1.

Procedimento 3.2.2 `incrementa_lista/3`

Os argumentos de `incrementa_lista/3` são:

[+] < arg_1 >: exemplo a ser considerado

[+] < arg_2 >: lista de contadores

[−] < arg_3 >: lista de contadores incrementada

```
incrementa_lista(Ex,Lista,ListaIncr):-
    verifica_at(Ex,classe,V),
    retira_at(Ex,classe,Ex1),
    atualiza_lista(Ex1,V,Lista,ListaIncr).
```

[5]No decorrer deste trabalho será mostrada a árvore resultante em uma forma gráfica sem as informações adicionais (lista de contadores e exemplos armazenados nos nós folhas), para maior facilidade de visualização.

[6]Note que os dois nós folhas da árvore expandida têm o mesmo valor de classe. Isto é devido ao fato de considerar apenas o conjunto de exemplos pertencentes àquele nó. O novo exemplo não é considerado na expansão da árvore.

O predicado **incrementa_lista/3** tem como objetivo criar uma lista onde são lembradas informações, através dos vários exemplos classificados pela árvore, tais como a quantidade de vezes que o atributo A_n cujo valor é i ($i = 0$ ou 1) e o valor da classe é j ($j = 0$ ou 1) comparece no processo de atualização da árvore de decisão. Isto será útil para realizar o cálculo da entropia (ou eventualmente, para implementar outra função de avaliação)[7], utilizado nesta implementação. A lista criada pelo procedimento **incrementa_lista/3** tem a seguinte forma:

$$[[At_1, A_1 : B_1, C_1 : D_1], [At_2, A_2 : B_2, C_2 : D_2], \ldots,]$$

onde

- At_n é o nome do atributo n
- A_n é o número de vezes que o atributo n comparece no nó de decisão com valor de atributo 0 e o valor da classe 0
- B_n é o número de vezes que o atributo n comparece no nó de decisão com valor de atributo 0 e o valor da classe 1
- C_n é o número de vezes que o atributo n comparece no nó de decisão com valor de atributo 1 e o valor da classe 0
- D_n é o número de vezes que o atributo n comparece no nó de decisão com valor de atributo 1 e o valor da classe 1

Por exemplo, a interrogação

```
?- incrementa_lista(Exemplo,Lista,ListaIncr).
```

ativada com

```
Exemplo = [[x1:1,x2:0,classe:1,x3:1]]

Lista = [[x1,0:1,1:1],[x2,1:1,0:1],[x3,1:1,0:1]]
```

é bem sucedida com

```
ListaIncr = [[x1,0:1,1:2],[x2,1:2,0:1],[x3,1:1,0:2]]
```

Após o procedimento **incrementa_lista/3** ser bem sucedido, a lista criada com essas informações é utilizada pelo procedimento **calcule_entropia/2** para realizar o cálculo da entropia e retornar o atributo mais representativo — aquele que tiver a menor entropia — para ser a raiz da árvore.

[7] Devido à modularidade da implementação, se outro critério for escolhido, somente o procedimento **calcule_entropia/2** deve ser redefinido.

Procedimento 3.2.3 `calcule_entropia/2`

Para calcular a entropia de um determinado atributo binário, é necessário calcular a entropia de suas duas ramificações. Se um ramo contém observações de uma única classe, pode-se dizer que ele está ordenado e possui entropia zero; sua desorganização é nula. Caso contrário, uma observação pode ser classificada em 2 classes diferentes: c_0 (classe = 0) e c_1 (classe = 1). Considerando que a probabilidade de um objeto pertencer à classe c_i é $p(i)$, então a entropia de classificação do ramo é dada por

$$En(A = v_j) = -\sum_{i=0}^{1} p(i)\log_2 p(i) \tag{1}$$

onde $A = v_j$ significa que o atributo A tem o valor v_j com $v_j \in \{0,1\}$. Isto é, $En(A = v_j)$ é a entropia do ramo correspondente ao valor v_j do atributo A. Para calcular a entropia total de um atributo deve-se considerar a entropia de cada um dos ramos correspondentes a este atributo. Seja:

M o número total de observações,

En_0 **e** En_1 as entropias dos dois ramos 0 e 1 respectivamente, associados ao atributo escolhido, e

N_0 **e** N_1 o número de observações de cada ramo.

Neste caso, a entropia de toda a ramificação, ou seja, a entropia do atributo A é dada por:

$$En(A) = \sum_{i=0}^{1} E_i \frac{N_i}{M} \tag{2}$$

Assim, para gerar árvores de decisão mais simples, em cada nó da árvore deve-se escolher o atributo de menor entropia para continuar ramificando a árvore. Deve ser observado que o algoritmo deve encontrar o atributo com menor valor de entropia, mas não o verdadeiro valor da entropia. Assim, a divisão pelo número total de exemplos é desnecessária. Os argumentos de `calcule_entropia/2` são:

[+] $< arg_1 >$: lista criada pelo procedimento `incrementa_lista/3`,

[−] $< arg_2 >$: lista contendo o menor valor da entropia e seu respectivo atributo

```
calcule_entropia(Lista,W) :-
    calcule_entropia(Lista,L,W).

calcule_entropia([],L,L]).

calcule_entropia([At,A:B,C:D]|R],L,W) :-
    entropia([A,B],E),
    entropia([C,D],I),
    Ent is ((A+C)*E) + ((B+D)*I),
    guarda_entropia(At,Ent,L,L1),
    calcule_entropia(R,L1,W).
```

O procedimento `calcule_entropia/2` ativa `calcule_entropia/3`. Este procedimento é composto de duas cláusulas. A primeira cláusula é o critério de parada. A segunda realiza, recursivamente, o cálculo da entropia para todos os atributos da lista. Os três primeiros predicados da segunda cláusula são responsáveis pelo cálculo da entropia do primeiro atributo, na lista corrente. Logo após, o valor da entropia calculado é comparado com o maior valor da entropia até agora encontrado, realizando a substituição se for o caso — procedimento `guarda_entropia/4`. Finalmente, `calcule_entropia/3` é chamado recursivamente para repetir este processo sobre os atributos restantes. Por exemplo, a interrogação

```
?- calcule_entropia(Lista,W).
```

ativada com

```
Lista = [[x1,0:1,1:2],[x2,1:2,0:1],[x3,1:1,0:2]]
```

é bem sucedida com

```
W = [x1,1.80617997]
```

Procedimento 3.2.4 `pull_up1/3`

Os argumentos de **`pull_up1/3`** são:

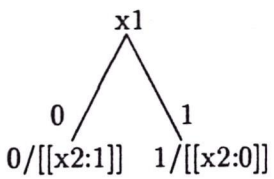
[+] < arg_1 >: atributo com o menor valor de entropia

[+] < arg_2 >: árvore de decisão a ser reestruturada

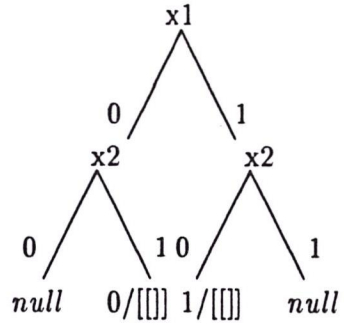
[−] < arg_3 >: árvore de decisão reestruturada

```
pull_up1(At,null,arv(null,At/[[At,0:0,0:0]],null)) :- !.

pull_up1(At,Classe/E,NovaArvore) :-
    criar_subarv(At,Classe/E,NovaArvore), !.

pull_up1(At,arv(Se,At/L,Sd),arv(Se,At/L,Sd)) :- !.

pull_up1(At,arv(null,R/L,null),arv(null,At/L,null)) :- !.

pull_up1(At,arv(Se,Raiz/L,Sd),NovaArvore) :-
    pull_up1(At,Se,Se1),
    pull_up1(At,Sd,Sd1),
    trocar_esq(Raiz,Se1,Se2),
    trocar_dir(Raiz,Sd1,Sd2),
    juntar_arv(L,Se2,Sd2,NovaArvore).
```

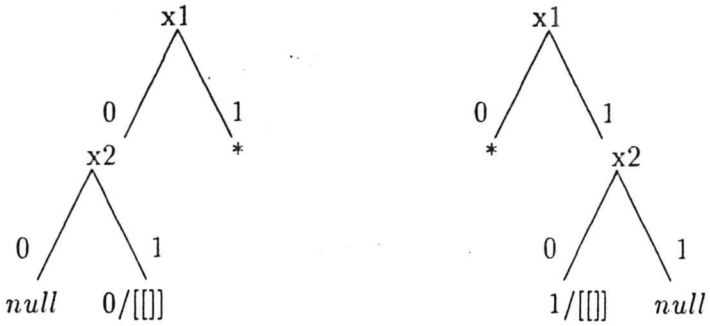
O processo `pull_up1/3` é responsável pela reestruturação da árvore. Este processo consiste em "empurrar" o atributo com maior ganho de informação, encontrado por `calcule_entropia/2`, para ser a raiz da árvore, sem que haja perda de informação de treinamento. Para reestruturar uma árvore de decisão no nó raiz de nível 0, a ordem dos atributos teste do nível 0 e do nível 1 é trocada, resultando em um reagrupamento de subárvores de 2 níveis. Este processo de reestruturação é realizado em cada uma das subárvores, até que o melhor atributo seja a raiz das subárvores imediatamente abaixo do nó que está sendo reestruturado. Durante o processo, a árvore é dividida em duas árvores para que o processo de "troca" dos atributos ocorra. O procedimento `juntar_arv/4` é responsável por agrupar as árvores depois da troca. Os contadores dos valores dos atributos no nível 0 já são conhecidos. Assim, apenas os contadores dos atributos teste e não teste do nível 1 necessitam ser calculados. Estes valores são obtidos diretamente adicionando os contadores respectivos das subárvores do nível 2. Finalmente, o procedimento de atualização da árvore continua recursivamente no próximo nível. A fim de ilustrar o processo realizado por `pull_up1/3`, será considerada a seguinte árvore de decisão:



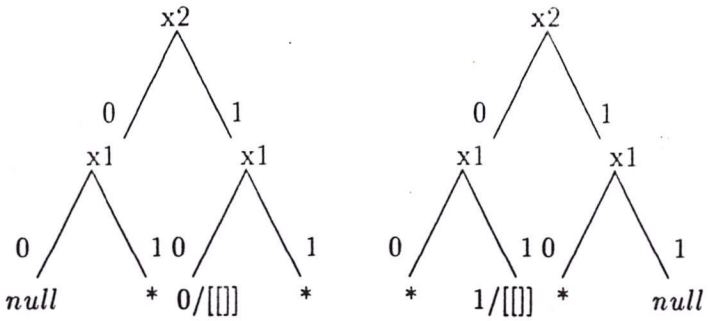
Através do cálculo da entropia, verifica-se que o melhor atributo é x2. O primeiro passo é fazer com que x2 seja a raiz de cada subárvore de x1. Assim a árvore é expandida nos nós folhas, como ilustrado a seguir:



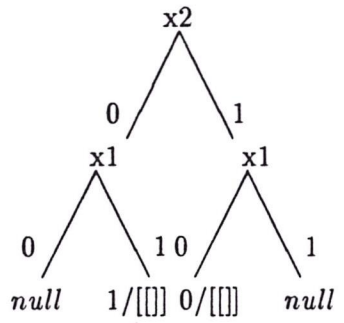
logo após, a árvore é dividida em duas:



e os atributos do nível 0 e 1 são trocados:



finalmente, as duas árvores são agrupadas:



O processo `pull_up1/3` é composto por cinco cláusulas que tratam as diferentes situações apresentadas na árvore de decisão, acarretando a cada uma um tratamento diferente. Estas situações são descritas a seguir.

A primeira cláusula trata a situação onde o nó a ser reestruturado é um nó folha rotulado com o átomo **null**. Quando isso ocorre, o nó passa a ser representado na forma de uma árvore de decisão onde a raiz da árvore contém o melhor atributo e suas subárvores são rotuladas com **null**. São criados os contadores com valor 0 para o atributo no nó, isto porque apesar da árvore ser expandida em um nível, nenhum exemplo é classificado naquele nó.

Na segunda cláusula, o nó é representado como um nó folha contendo o valor da classe. Neste caso, o nó é expandido, também, em um nível através do procedimento **criar_subarv/3**. Na nova árvore, a raiz contém o melhor atributo e os exemplos armazenados no nó são divididos de acordo com os valores que a raiz pode assumir. É criado o contador de exemplos positivos e negativos para cada atributo possível para aquele nó de acordo com os exemplos armazenados, e as subárvores são rotuladas com o valor da classe. Por exemplo, a interrogação

```
?- pull_up1(At,Classe/E,NovaArvore).
```

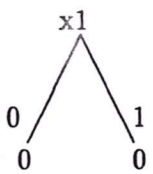
ativada com

```
At = x3

Classe/E = 0/[[x1:0,x2:1,x3:1],[x1:1,x2:1,x3:0],[x1:0,x2:0,x3:0]]
```

é bem sucedida com

```
NovaArvore = arv(0/[[x1:1,x2:1],[x1:0,x2:0]],x3/[[x1,2:0,1:0],
                    [x2,1:0,2:0],[x3,2:0,1:0]],0/[[x1:0,x2:1]]
```

Árvore Gráfica:



Ou seja, as duas primeiras cláusulas fazem a expansão da árvore de decisão. Isto é, antes a árvore consistia de, no máximo, apenas um nó, após ativar uma das cláusulas, a árvore de decisão é retornada com a estrutura completa

arv(SubArvEsq,Raiz/L,SubArvDir)

A terceira cláusula é ativada quando a raiz da subárvore já é o atributo mais informativo. Neste caso, não haverá nenhuma transformação e a árvore original não é alterada.

A quarta cláusula é ativada quando a raiz da árvore é um atributo diferente do atributo mais informativo e suas subárvores estão rotuladas com `null`. A tarefa desta cláusula é trocar o atributo da raiz conservando a lista de contadores e suas subárvores. Não é necessário expandir a árvore porque não existe nenhum exemplo classificado pelo nó, dessa forma o atributo que representa a raiz é irrelevante. Com isto economiza-se tempo e espaço, pois em cada nó de decisão, todos os contadores para os atributos devem ser mantidos.

Por fim, a última cláusula trata o caso genérico, onde efetivamente é realizado o processo de reestruturação da árvore. Como o procedimento é recursivo, é realizado para cada subárvore, esquerda e direita, o processo de `pull_up1/3` para "empurrar" o atributo mais relevante para a raiz de cada subárvore imediatamente inferior à raiz do nó da subárvore que será reestruturada. Com o atributo mais relevante sendo a raiz de cada subárvore no nível 1, a árvore é, teoricamente, dividida para realizar a troca entre os atributos das raízes do nível 0 e 1. Depois de realizada a troca, unem-se as duas árvores e com isso todas as informações necessárias são mantidas e a consistência da árvore é preservada.

Quando o procedimento `pull_up1/3` é chamado por `arvore_incr/3`, é ativada a última cláusula do procedimento. Isto porque sempre que for necessário reestruturar uma árvore de decisão, esta terá a estrutura completa da árvore — `arv(Se,Raiz/L,Sd)`.

A seguir é comentado, em maiores detalhes, os procedimentos que realizam a troca entre a raiz da árvore e a raiz da subárvore, bem como processo que une as duas árvores.

Procedimento 3.2.5 `trocar_esq/3` e `trocar_dir/3`

Os argumentos de `trocar_esq/3` e `trocar_dir/3` são:

[+] $< arg_1 >$: raiz da árvore original que está sendo reestruturada

[+] $< arg_2 >$: subárvore que tem como raiz o melhor atributo

[−] $< arg_3 >$: subárvore com o atributo `R` adicionado como um nó filho da subárvore

```
trocar_esq(R,arv(Se,MAt,Sd),arv(arv(Se,R,_),MAt,arv(Sd,R,_))).

trocar_dir(R,arv(Se,MAt,Sd),arv(arv(_,R,Se),MAt,arv(_,R,Sd))).
```

Os procedimentos `trocar_esq/3` e `trocar_dir/3` são responsáveis por realizar a troca do atributo que está na raiz da árvore no nível 0, pelo atributo que está na raiz das subárvores imediatamente abaixo. É importante lembrar que a raiz das subárvores esquerda e direita contém o melhor atributo obtido através do procedimento recursivo `pull_up1/3`, realizado nas duas subárvores. Com isso as subárvores são agora tratadas como uma árvore isolada, tendo como raiz o melhor atributo. O atributo que representa a raiz – `R` – da árvore original, passa a ser o nó filho imediatamente abaixo do melhor atributo de cada árvore que foi isolada. As subárvores, esquerda e direita, da árvore

esquerda que foi isolada, são agora subárvores esquerda da raiz - R. As subárvores direitas de R são variáveis livres. Isto porque a árvore isolada era uma subárvore esquerda de R, assim os exemplos, neste ramo da árvore, são classificados para o atributo R. O mesmo se aplica para a subárvore direita.

Procedimento 3.2.6 juntar_arv/4

Os argumentos de **juntar_arv/4** são:

[+] $< arg_1 >$: lista de contadores do nó que está sendo reestruturado

[+] $< arg_2 >$: subárvore esquerda modificada

[+] $< arg_3 >$: subárvore direita modificada

[−] $< arg_4 >$: árvore de decisão representando as duas subárvores

```
juntar_arv(L,arv(arv(Se1,R1,_),R/L1,arv(Se2,R1,_)),
           arv(arv(_,R1,Sd1),R/L2,arv(_,R1,Sd2)),
           arv(arv(Se1,R1/L3,Sd1),R/L,arv(Se2,R1/L4,Sd2))) :-
    organiza_lista(R,R1,Se1,Sd1,Se2,Sd2,L1,L2,L3,L4).
```

Este procedimento é responsável por agrupar as duas subárvores que foram isoladas. O processo é relativamente simples. Como as duas subárvores têm o mesmo atributo representando a raiz da árvore no nível 0 e os mesmos atributos representando os filhos de raiz no nível 1, o processo consiste em manter na nova árvore todos estes atributos, da raiz e dos filhos da raiz, e adicionar as subárvores pertencentes aos filhos da raiz, nível 2 da árvore. Deve ser lembrado que a árvore que era considerada subárvore esquerda, depois da troca, contém apenas subárvore esquerda no nível 2; o mesmo acontece com a subárvore direita. Assim, obedecendo as ramificações das raízes no nível 0 e no nível 1, basta acrescentar as subárvores esquerda e direita do nível 2. Por exemplo, considerando as duas árvores de decisões como árvores distintas tem-se:

```
ArvoreEsq = arv(arv(Se1,R1,_),R/L1,arv(Se2,R1,_))

ArvoreDir = arv(arv(_,R1,Sd1),R/L2,arv(_,R1,Sd2))
```

Observe que a raiz - R - de cada árvore, assim como a raiz de suas subárvores - R1 - são iguais. O procedimento mantém estes nós de decisão na árvore e adiciona as subárvores restantes **Se1**, **Sd1** como subárvores esquerda e direita, respectivamente, no ramo esquerdo e **Se2**, **Sd2** no ramo direito dos nós de decisão de **R1**. A árvore de decisão final será:

```
ArvoreFinal = arv(arv(Se1,R1/L3,Sd1),R/L,arv(Se2,R1/L4,Sd2))
```

O procedimento `juntar_arv/4` ativa apenas um predicado — `organiza_lista/10` — que é o responsável por atualizar e organizar a lista de contadores para cada nó de decisão da árvore. O procedimento necessita de várias informações armazenadas na árvore e, além disso, trata os atributos de forma diferenciada, isto é, o melhor atributo, que será o primeiro elemento da lista, necessita de informações diferentes dos outros atributos. Por exemplo, as informações que o melhor atributo necessita estão presentes nas listas de contadores das raízes de cada árvore isolada, enquanto que o restante dos atributos necessitam de informações contidas nas instâncias de treinamento pertencentes aos nós folhas ou nas listas de contadores armazenadas nas raízes inferiores do nó. Este cálculo é realizado através do procedimento `organiza_lista/10`.

Procedimento 3.2.7 `organiza_lista/10`

Os argumentos de `organiza_lista/10` são:

[+] $< arg_1 >$: atributo com menor valor de entropia

[+] $< arg_2 >$: atributo que era a raiz da árvore antes do processo de reestruturação

[+] $< arg_3 >$: subárvore esquerda do nível 1 da subárvore esquerda do nível 0 da primeira árvore isolada

[+] $< arg_4 >$: subárvore direita do nível 1 da subárvore esquerda do nível 0 da segunda árvore isolada

[+] $< arg_5 >$: subárvore esquerda do nível 1 da subárvore direita do nível 0 da primeira árvore isolada

[+] $< arg_6 >$: subárvore direita do nível 1 da subárvore direita do nível 0 da segunda árvore isolada

[+] $< arg_7 >$: lista de contadores do nó que contém o melhor atributo da primeira árvore

[+] $< arg_8 >$: lista de contadores do nó que contém o melhor atributo da segunda árvore

[−] $< arg_9 >$: lista de contadores do nó raiz da subárvore esquerda do nível 1

[−] $< arg_{10} >$: lista de contadores do nó raiz da subárvore direita do nível 1

```
organiza_lista(MAt,At,Se1,Sd1,Se2,Sd2,L1,L2,[El1|Resto1],
                                              [El2|Resto2]) :-
    organiza_cabeca(MAt,At,L1,L2,El1,El2),
    organiza_resto(MAt,L1,Se1,Sd1,Resto1),
    organiza_resto(MAt,L1,Se2,Sd2,Resto2).
```

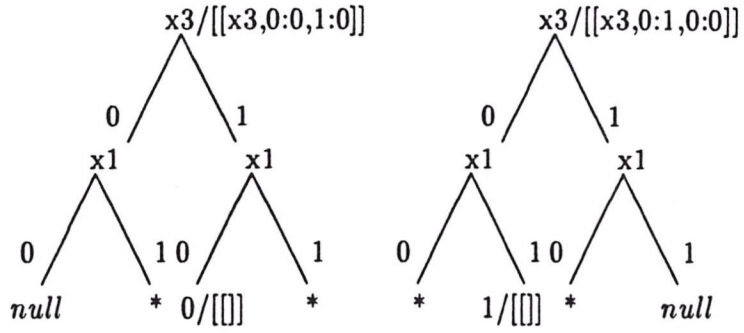
A lista de contadores é importante para verificar, através do cálculo da entropia, os atributos com maior ganho de informação. Sempre que um novo exemplo for classificado pela árvore de decisão, a cada nó de decisão da árvore que classifica o exemplo, a lista de contadores é incrementada — **incrementa_lista/3**, pg. 13 — e o cálculo da entropia é realizado para todos os atributos possíveis naquele nó.

O procedimento **organiza_lista/10** atualiza a lista de contadores não incrementando-a, mas sim, reorganizando-a com os exemplos já classificados por ela. A lista de contadores que pertence ao nó que está sendo reestruturado permanece inalterada, pois independente de qual atributo compõe o nó raiz da árvore, todos os exemplos são classificados igualmente naquele nó. A partir do momento em que o atributo da raiz é mudado, os exemplos devem ser, novamente, classificados em suas subárvores, pois os exemplos, agora, são divididos de acordo com o novo atributo. Para reorganizar a lista de contadores são necessárias informações diferentes para atributos diferentes. O procedimento organiza_lista/10 ativa dois outros procedimentos: organiza_cabeca/6 e organiza_resto/5. O procedimento organiza_cabeca/6 é o responsável por reorganizar a lista que contém o melhor atributo. Esta lista será o primeiro elemento da nova lista de contadores. O procedimento organiza_resto/5 é responsável pela reorganização dos atributos restantes.

Procedimento 3.2.8 organiza_cabeca/6

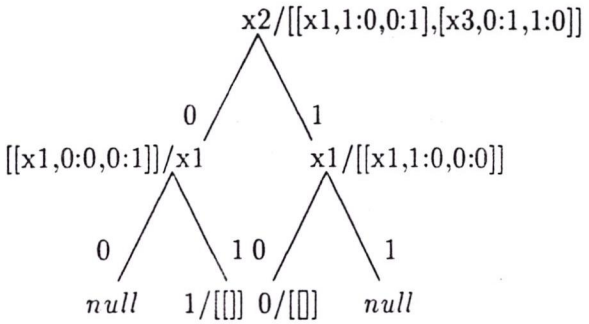
```
organiza_cabeca(MAt,At,L1,L2,[At,A,A1],[At,B,B1]) :-
    busca_at(MAt,L1,A,B),
    busca_at(MAt,L2,A1,B1).
```

Este procedimento busca uma relação entre o melhor atributo e o atributo que era a raiz da árvore original. Se forem consideradas as duas árvores separadamente, nota-se que as raízes possuem listas de contadores, onde o atributo que pertence à raiz de suas subárvores não faz parte da lista. Deve ser lembrado que o melhor atributo – **MAt** – é agora a raiz principal da árvore de decisão, e que o atributo – **At** – que representava a raiz principal anterior ao processo de reestruturação, está em um nível abaixo do melhor atributo, isto é, o atributo **At** é o nó filho do atributo **MAt**. A lista de contadores ao qual pertence o nó de **MAt** não terá modificações. No nível abaixo, no nó de **At** não existe, ainda, nenhuma lista de contadores. Esta lista será criada de acordo com as informações contidas na árvore de decisão. A construção da lista pertencente à **At** a partir de **MAt** é quase imediata. Para melhor entendimento dos procedimentos de organização das listas de contadores, considere as seguintes árvores de decisão:



A primeira árvore é considerada, teoricamente, como subárvore esquerda do atributo `At`; neste caso `At` é representado pelo atributo `x1` e `MAt` por `x3`. A segunda árvore é, consequentemente, considerada como subárvore direita. Daí, tira-se a informação que os contadores de `x3` na primeira árvore são valores que indicam a quantidade de exemplos classificados por `x3` e para os quais `x1` tem valor 0 e na segunda árvore, `x1` tem valor 1.

Com o procedimento `juntar_arv/4` as duas subárvores são agrupadas, tornando-se apenas uma.



As duas listas pertencentes a `x3` — denominadas `L1` e `L2` no procedimento — são substituídas por `L` — a lista que pertence à `x1` na árvore original. As listas de contadores `L1` (`L1 = [[x3,0:0,1:0]]`) e `L2` (`L2 = [[x3,0:1,0:0]]`) são necessárias apenas para o cálculo do atributo `x1` nas subárvores. Como visto anteriormente, cada lista da lista de contadores é representada pela estrutura

$$[At_n, A_n : B_n, C_n : D_n]$$

O atributo `x1` irá assumir os valores de `x3` das listas `L1` — ramo esquerdo — e `L2` — ramo direito —, onde os contadores A_3 e B_3 de `x3` da lista `L1`, serão A_1 e B_1 de `x1` e A_3 e B_3 da lista `L2` serão C_1 e D_1 de `x1` pertencente ao ramo esquerdo da nova árvore.

Para `x1` do ramo direito da nova árvore, seus contadores A_1 e B_1 são os mesmos contadores C_3 e D_3 da lista `L1` e os contadores C_1 e D_1 de `x1` são C_3 e D_3 da lista `L2`. Isto é feito porque as listas de contadores, do atributo `x3`, foram criadas de acordo com o valor de `x1`.

Procedimento 3.2.9 `organiza_resto/5`

```
organiza_resto(MAt,L1,null,null,L) :-
    retira_atributo(MAt,L1,L2),
    zera_lista(L2,L), !.

organiza_resto(_,_,Se,Sd,L) :-
    resto(Se,L1),
    resto(Sd,L2),
    conta(L1,L2,L), !.

resto(null,[]).

resto(Classe/Exs,L) :-
    cria_lista(Exs,Classe,L), !.

resto(arv(Se,R/L,Sd),L).
```

Após criar a lista de contadores para o melhor atributo, este procedimento irá organizar a lista de contadores para o restante dos atributos. As informações necessárias para criar o restante da lista são buscadas nos nós imediatamente abaixo do nó em questão — nó filho. O procedimento considera as diversas situações possíveis do nó filho. Quando em um nó são encontradas duas subárvores rotuladas com `null`, significa que nenhum exemplo é classificado pelo nó. Neste caso, o melhor atributo é retirado da lista de contadores — o processamento está sendo realizado no nível 1 da árvore e o melhor atributo pertence à raiz do nível 0 — e a lista resultante é zerada através de `zera_lista/2`.

Caso não ocorra esta situação, isto é, pelo menos uma das ramificações não está rotulada com `null`, `organiza_resto/5` vai processar a lista de cada subárvore separada e somar as duas listas encontradas. Cada subárvore pode ser representada pelas seguintes estruturas:

1. o rótulo `null`, neste caso a lista retornada é a lista vazia
2. um nó folha contendo o valor da classe. Neste caso é criada uma lista de acordo com os exemplos pertencentes ao nó — `cria_lista/3`
3. uma subárvore com a estrutura da árvore geral — `arv(Se,R/L,Sd)` — cuja raiz contém uma lista de contadores `L` associada a ela. Esta lista `L` é retornada como a lista de contadores da subárvore.

Com a lista de cada subárvore, o procedimento `conta/3` soma os contadores dos atributos respectivos das listas.

4 Exemplos de Execução

Nesta seção são apresentados vários exemplos de execução do procedimento implementado **arvore_incr/3**, para diversos conjuntos de exemplos. Para fim de testes, foi implementado um procedimento que, começando com uma árvore vazia, ativa **arvore_incr/3** para inserir um a um os exemplos fornecidos. Este procedimento possui opções para trabalhar na memória principal, bem como em memória secundária. Em ambos os casos, a árvore construída após a inserção de um novo exemplo, constitui-se na árvore de entrada onde é inserido o próximo exemplo.

A opção de utilizar memória secundária é a que corresponde à realidade pois, devido ao fato do algoritmo ser incremental, é necessário lembrar a árvore incremental construída para nela inserir novos exemplos quando disponíveis. Sem entrar nos detalhes da implementação dos procedimentos referentes ao uso de memória — principal ou secundária — o algoritmo utilizado para fins de testes é o seguinte:

Entrada: `Arvore, Exemplos` = {conjunto de exemplos}

```
repeat
   Ex ∈ Exemplos
   arvore_incr(Ex,Arvore,ArvoreIncr)
   Exemplos = Exemplos - Ex
   Arvore = ArvoreIncr
until Exemplos = { }
enxugue_arv(ArvoreIncr,ArvorePadrao)
```

Saída: `ArvoreIncr,ArvorePadrao`

O procedimento **enxugue_arv/2** transforma a árvore de decisão incremental, que carrega todas as informações necessárias para que novos exemplos possam ser inseridos nessa árvore, na árvore de decisão binária na forma padrao

arv(SubArvEsq,Raiz,SubArvDir)

Esta é a estrutura geral da árvore de decisão binária construída pelo algoritmo não incremental ID3 implementado, a qual é possível aplicar o Método de Poda de Redução do Erro, também implementado, a fim de diminuir o erro de classificação de novos exemplos, — veja [Martins 94].

Nos exemplos a seguir é mostrado a instanciação das variáveis de E/S correspondentes a aplicação de teste.

Exemplo 4.1

Entrada:

```
Arvore = null
```

```
Exemplos = [[x1:1, x2:1, classe:0, x3:1, x4:0],
            [x1:0, x2:0, classe:1, x3:1, x4:0],
            [x1:0, x2:0, classe:0, x3:0, x4:1],
            [x1:1, x2:1, classe:1, x3:1, x4:1],
            [x1:1, x2:1, classe:1, x3:1, x4:1],
            [x1:0, x2:0, x3:1, classe:0, x4:1]].
```

Saída:

```
ArvoreIncr =
          arv(arv(arv(0/[[x4:1]],x2/[[x2,1:0,0:0],[x4,0:1,1:0]],null),
          x1/[[x1,1:0,0:0],[x2,1:0,0:0],[x4,0:0,1:0]],null),x3/[[x1,2:
          1,1:2],[x2,2:1,1:2],[x3,1:0,2:3],[x4,1:1,2:2]],arv(arv(arv(
          null,x4/[[x4,0:1,1:0]],0/[[]]),x2/[[x2,1:1,0:0],[x4,0:1,1:0]
          ],null),x1/[[x1,1:1,1:2],[x2,1:1,1:2],[x4,1:1,1:2]],arv(null,
          x2/[[x2,0:0,1:2],[x4,1:0,0:2]],arv(null,x4/[[x4,1:0,0:2]],1/
          [[]])))))

ArvorePadrao =
     arv(arv(arv(0,x2,null),x1,null),x3,arv(arv(arv(null,x4,0),
     x2,null),x1,arv(null,x2,arv(null,x4,1))))
```

Neste exemplo, a árvore de decisão inicial está vazia — `null`. Após a construção e classificação de todos exemplos, individualmente, a árvore de decisão gerada — `ArvoreIncr` — está em sua forma geral, isto é, contém tanto as informações necessárias ao cálculo da entropia armazenadas nos nós de decisão, quanto as instâncias de treinamento armazenados nos nós folhas. Esta mesma árvore de decisão em sua forma padrão é dada por `ArvorePadrao`.

Para melhor visualização será considerada, nos exemplos a seguir, apenas a árvore de decisão no formato padrão.

Exemplo 4.2

Entrada:

```
Arvore = arv(arv(arv(0/[[x4:1]],x2/[[x2,1:0,0:0],[x4,0:1,1:0]],null),
          x1/[[x1,1:0,0:0],[x2,1:0,0:0],[x4,0:0,1:0]],null),x3/[[x1,2:
          1,1:2],[x2,2:1,1:2],[x3,1:0,2:3],[x4,1:1,2:2]],arv(arv(arv(
          null,x4/[[x4,0:1,1:0]],0/[[]]),x2/[[x2,1:1,0:0],[x4,0:1,1:0]
          ],null),x1/[[x1,1:1,1:2],[x2,1:1,1:2],[x4,1:1,1:2]],arv(null,
          x2/[[x2,0:0,1:2],[x4,1:0,0:2]],arv(null,x4/[[x4,1:0,0:2]],1/
          [[]])))))
```

```
Exemplos = [[x1:1, x2:1, classe:0, x3:0, x4:0],
            [x1:0, x2:1, classe:1, x3:0, x4:0],
            [x1:1, x2:1, classe:0, x3:0, x4:0],
            [x1:1, x2:1, classe:1, x3:1, x4:1],
            [x1:1, x2:0, x3:1, classe:1, x4:1]].
```

Saída:

```
ArvorePadrao = arv(arv(arv(arv(null,x1,null),x4,arv(null,x1,null)),x2,
               arv(arv(null,x4,null),x1,arv(0,x4,null))),x3,arv(arv(
               arv(null,x4,0),x2,null),x1,arv(arv(null,x2,null),x4,
               arv(1,x2,1))))
```

Exemplo 4.3

Entrada:

```
Arvore = null

Exemplos = [[verde:1,amarelo:0,classe:0,azul:1,vermelho:0,preto:0],
            [verde:1,amarelo:1,classe:0,azul:1,vermelho:1,preto:1],
            [verde:0,amarelo:0,azul:1,classe:1,vermelho:1,preto:1],
            [verde:0,amarelo:1,azul:0,classe:0,vermelho:0,preto:1],
            [verde:1,amarelo:1,azul:1,vermelho:0,preto:1,classe:0],
            [verde:1,amarelo:1,azul:1,vermelho:0,preto:0,classe:1],
            [verde:0,amarelo:0,azul:0,vermelho:0,preto:0,classe:0],
            [verde:1,amarelo:1,azul:1,vermelho:1,preto:1,classe:0],
            [verde:1,amarelo:0,azul:1,vermelho:0,preto:1,classe:0],
            [verde:0,amarelo:1,azul:0,vermelho:1,preto:0,classe:1]].
```

Saída:

```
ArvorePadrao = arv(arv(arv(arv(0,verde,arv(null,azul,null)),preto,
               arv(null,verde,arv(null,azul,0))),amarelo,arv(0,verde,
               arv(null,azul,arv(1,preto,null)))),vermelho,arv(1,
               verde,0))
```

5 Listagem do Programa

Segue a listagem do procedimento implementado:

```
arvore_incr([],Arv,Arv).

arvore_incr(Exemplo,null,Classe/[E1]) :-
        verifica_at(Exemplo,classe,Classe),
        retira_at(Exemplo,classe,E1),
        !.

arvore_incr(Exemplo,Classe/E,Classe/E1) :-
        verifica_at(Exemplo,classe,Classe),
        retira_at(Exemplo,classe,Ex),
        insere_ex(Ex,E,E1),
        !.

arvore_incr(Exemplo,Classe/E,Arv) :-
        verifica_at(Exemplo,classe,Classe1),
        pega_at(E,At),
        criar_subarv(At,Classe/E,ArvE),
        arvore_incr(Exemplo,ArvE,Arv),
        !.

arvore_incr(Exemplo,arv(Se,Raiz/L,Sd),Arv) :-
        incrementa_lista(Exemplo,L,LI),
        calcule_entropia(LI,[Raiz1,_]),
        pull_up1(Raiz1,arv(Se,Raiz/L,Sd),arv(Sep,Raiz1/L,Sdp)),
        verifica_at(Exemplo,Raiz1,Val),
        retira_at(Exemplo,Raiz1,E1),
        ifthenelse(Val=0,
                (arvore_incr(E1,Sep,Se1),Arv=arv(Se1,Raiz1/LI,Sdp)),
                (arvore_incr(E1,Sdp,Sd1),Arv=arv(Sep,Raiz1/LI,Sd1))),
                !.

verifica_at([At:Val|R],At,Val) :- !.

verifica_at([X:V|R],At,Val) :-
        verifica_at(R,At,Val).

retira_at([X:V|Y],X,Y).

retira_at([X:V|Y],Z,[X:V|Y1]) :-
        retira_at(Y,Z,Y1).

insere_ex(X,[],[X]) :- !.

insere_ex(X,[Y|R],[X,Y|R]).

incrementa_lista(Ex,L,L1) :-
        verifica_at(Ex,classe,V),
```

```
        retira_at(Ex,classe,Ex1),
        atualiza_lista(Ex1,V,L,L1).

atualiza_lista([],_,L,L).

atualiza_lista([At:V|R],Cl,L,L1) :-
        procura_at(At,V,Cl,L,Laux),
        atualiza_lista(R,Cl,Laux,L1).

procura_at(At,V,Cl,[],[[At,A:B,C:D]]) :-
        soma_lista(V,Cl,[At,0:0,0:0],[At,A:B,C:D]), !.

procura_at(At,V,Cl,[[At,A:B,C:D]|R],[L1|R]) :-
        !,
        soma_lista(V,Cl,[At,A:B,C:D],L1).

procura_at(At,V,Cl,[[At1,X,Y]|R],[[At1,X,Y]|L]) :-
        procura_at(At,V,Cl,R,L).

soma_lista(0,0,[At,A:B,C:D],[At,A1:B,C:D]) :-
        A1 is A + 1, !.

soma_lista(0,1,[At,A:B,C:D],[At,A:B1,C:D]) :-
        B1 is B + 1, !.

soma_lista(1,0,[At,A:B,C:D],[At,A:B,C1:D]) :-
        C1 is C + 1, !.

soma_lista(1,1,[At,A:B,C:D],[At,A:B,C:D1]) :-
        D1 is D + 1.

lista(Ex,V_Classe,ListaFinal) :-
        lista_inicial(Ex,ListaInicial),
        atualiza_lista(Ex,V_Classe,ListaInicial,ListaFinal).

lista_inicial([],[]).

lista_inicial([At:V|R],[[At,0:0,0:0]|Resto]) :-
        lista_inicial(R,Resto).

calcule_entropia(Lista,MA) :-
        entropia_arv(Lista,MA1,MA).

entropia_arv([],L,L).

entropia_arv([[At,A:B,C:D]|R],L,W) :-
        entropia([A,B],E),
        entropia([C,D],I),
        Ent is ((A+C)*E) + ((B+D)*I),
        guarda_entropia(At,Ent,L,L1),
        entropia_arv(R,L1,W).
```

```
entropia(L,E) :-
        entropia(L,0,E).

entropia([],Ent,Ent).

entropia([A|R],Temp,E) :-
        logaritmo(A,Alog),
        NovoTemp is Alog * A + Temp,
        entropia(R,NovoTemp,E).

guarda_entropia(At,E,[],[At,E]).

guarda_entropia(At,Ent,[A,E],[A,E]) :-
        Ent =< E, !.

guarda_entropia(At,Ent,_,[At,Ent]).

logaritmo(A,0) :-
        A =:= 0, !.

logaritmo(A,Alog) :-
        Alog is log(A).

pull_up1(At,null,arv(null,At/[[At,0:0,0:0]],null)) :-
        !.

pull_up1(At,Classe/E,NovaArvore) :-
        criar_subarv(At,Classe/E,NovaArvore), !.

pull_up1(At,arv(Se,At/L,Sd),arv(Se,At/L,Sd)) :- !.

pull_up1(At,arv(null,R/L,null),arv(null,At/L,null)) :- !.

pull_up1(At,arv(Se,Raiz/L,Sd),NovaArvore) :-
        pull_up1(At,Se,Se1),
        pull_up1(At,Sd,Sd1),
        trocar_esq(Raiz,Se1,Se2),
        trocar_dir(Raiz,Sd1,Sd2),
        juntar_arv(L,Se2,Sd2,NovaArvore).

trocar_esq(R,arv(Se,MAt,Sd),arv(arv(Se,R,_),MAt,arv(Sd,R,_))).

trocar_dir(R,arv(Se,MAt,Sd),arv(arv(_,R,Se),MAt,arv(_,R,Sd))).

juntar_arv(L,arv(arv(Se1,R1,_),R/L1,arv(Se2,R1,_)),
                arv(arv(_,R1,Sd1),R/L2,arv(_,R1,Sd2)),
                arv(arv(Se1,R1/L3,Sd1),R/L,arv(Se2,R1/L4,Sd2))) :-
        organiza_lista(R,R1,Se1,Sd1,Se2,Sd2,L1,L2,L3,L4).

verificar(At,Classe/E,Se,Sd) :-
        verifica_ex(At,E,E1,E2),
        ver(Classe,E1,Se),
```

```
        ver(Classe,E2,Sd).

verifica_ex(At,E,E1,E2) :-
        verifica_ex(At,E,[],E1,[],E2).

verifica_ex(At,[],E1,E1,E2,E2).

verifica_ex(At,[X|Y],E1a,E1,E2a,E2) :-
        verifica_at(X,At,V),
        retira_at(X,At,Eaux),
        ifthenelse(V=0,
                (insere_ex(Eaux,E1a,E1aux),
                 verifica_ex(At,Y,E1aux,E1,E2a,E2),!),
                (insere_ex(Eaux,E2a,E2aux),
                 verifica_ex(At,Y,E1a,E1,E2aux,E2),!)).

ver(Classe,[],null) :- !.

ver(Classe,E,Classe/E).

cria_lista([],_,null) :- !.

cria_lista([E|R],C,L) :-
        lista(E,C,L1),
        atualiza_lista1(R,C,L1,L).

atualiza_lista1([],_,L,L) :- !.

atualiza_lista1(Exs,Classe,null,Lista) :-
        cria_lista(Exs,Classe,Lista) ,!.

atualiza_lista1([X|Y],C,L1,L) :-
        atualiza_lista(X,C,L1,L2),
        atualiza_lista1(Y,C,L2,L).

organiza_lista(MAt,At,Se1,Sd1,Se2,Sd2,L1,L2,[El1|Resto1],[El2|Resto2]) :-
        organiza_cabeca(MAt,At,L1,L2,El1,El2),
        organiza_resto(MAt,L1,Se1,Sd1,Resto1),
        organiza_resto(MAt,L1,Se2,Sd2,Resto2).

organiza_cabeca(MAt,At,L1,L2,[At,A,A1],[At,B,B1]) :-
        busca_at(MAt,L1,A,B),
        busca_at(MAt,L2,A1,B1).

organiza_resto(MAt,L1,null,null,L) :-
        retira_atributo(MAt,L1,L2),
        zera_lista(L2,L), !.

organiza_resto(_,_,Se,Sd,L) :-
        resto(Se,L1),
        resto(Sd,L2),
        conta(L1,L2,L), !.
```

```
resto(null,[]).

resto(Classe/Exs,L) :-
        cria_lista(Exs,Classe,L), !.

resto(arv(Se,R/L,Sd),L).

conta([],L,L) :- !.

conta(L,[],L) :- !.

conta([[At,A,B]|Y],L,[X1|Y1]) :-
        conta1([At,A,B],L,X1), !,
        retira_atributo(At,L,L1),
        conta(Y,L1,Y1).

conta1(L,[],L) :- !.

conta1([At,A:B,C:D],[[At,A1:B1,C1:D1]|Resto1],[At,A2:B2,C2:D2]) :-
        A2 is A+A1,
        B2 is B+B1,
        C2 is C+C1,
        D2 is D+D1.

conta1([At,A,B],[[Atrib,A1,B1]|Resto],X) :-
        At \= Atrib,
        conta1([At,A,B],Resto,X).

busca_at(MAt,[],0:0,0:0) :- !.

busca_at(MAt,[[MAt,A,B]|R],A,B) :- !.

busca_at(MAt,[[At,_,_]|R],A,B) :-
        busca_at(MAt,R,A,B).

retira_atributo(MAt,[],[]).

retira_atributo(MAt,[[MAt,A,B]|R],R).

retira_atributo(MAt,[[At,A,B]|R],[[At,A,B]|R1]) :-
        retira_atributo(MAt,R,R1).

zera_lista([],[]).

zera_lista([[At,A:B,C:D]|R],[[At,0:0,0:0]|R1]) :-
        zera_lista(R,R1).

criar_subarv(At,Classe/E,arv(Se,At/L,Sd)) :-
        verificar(At,Classe/E,Se,Sd),
        cria_lista(E,Classe,L).
```

```
pega_at([L|R],At) :-
        verifica_at(L,At,_).

pegue_exemplo([],[]).

pegue_exemplo([Ex|Exs],Ex).

pegue_exemplo([Ex|Exs],Exx) :-
        pegue_exemplo(Exs,Exx).

enxugue_arv(Classe/E,Classe).

enxugue_arv(null,null).

enxugue_arv(arv(Se,R/L,Sd),arv(Se1,R,Sd1)) :-
        enxugue_arv(Se,Se1),
        enxugue_arv(Sd,Sd1).
```

6 Conclusão

Neste trabalho foi apresentada e discutida em detalhes a implementação de um algoritmo indutivo incremental de AM, que constrói uma árvore de decisão para atributos com valores binários. O algoritmo utiliza a metodologia da família TDIDT e, mais especificamente, a estratégia de aprendizado do algoritmo ID5. O algoritmo ID5 tem como objetivo construir uma árvore de decisão no modo incremental; para isso ele retém informações suficientes e necessárias para calcular o *E-score* dos possíveis atributos em qualquer nó de decisão, assim como mantém instâncias de treinamento nos nós folhas da árvore. Além disso, o algoritmo utiliza um processo de reestruturação da árvore para classificar um novo exemplo e manter sempre como raiz da árvore o atributo com o menor valor de entropia.

A vantagem deste algoritmo, além de ser um algoritmo incremental — sem o custo de reconstruir toda a árvore de decisão a cada nova instância de treinamento —, é o de possibilitar o processo de reestruturação da árvore. Este processo manipula a árvore de forma a preservar a sua consistência com as instâncias de treinamento já observadas.

A árvore de decisão final — na forma padrão — construída pelo algoritmo ID5, geralmente é uma árvore de decisão com uma quantidade maior de nós do que a árvore de decisão gerada pelo algoritmo não incremental ID3 implementado [Martins 94]. Isto porque o algoritmo incremental utiliza o processo de expansão dos nós folhas da árvore de decisão, onde muitas vezes gera vários átomos `null`. Contudo, isto não é um problema já que a árvore final construída pelo ID5 — na forma padrão — pode ser submetida ao processo de poda implementado [Martins 94] que se encarregará de retirar, entre outros, esses átomos `null`.

A implementação foi realizada na linguagem de programação lógica Prolog, especificamente Arity Prolog [Arity 92], para microcomputadores PC compatíveis.

A principal desvantagem encontrada, foi a grande demanda de recursividade no processo de construção e classificação de exemplos pela árvore. Esta demanda exigiu bastante dos recursos da máquina ocasionando, algumas vezes, impossibilidade de classificar um novo exemplo na árvore de decisão, caso fosse necessário a reestruturação da mesma. Este problema foi parcialmente contornado utilizando memória secundária. Deve ser salientado que vários dos problemas referentes ao uso de memória durante a execução, estão diretamente relacionados à algumas limitações do Arity. Assim, é considerado seriamente o uso de outro software, tal como LPA Prolog, para realizar as implementações nos microcomputadores PC compatíveis.

Deve ser observado que, em geral, e em parte devido a maior disponibilidade de máquinas no Laboratório de Computação do ICMSC-USP, as implementações são primeiramente realizadas em PC's para depois serem customizadas para as estações SUN's. O software utilizado nas estações é Sicstus Prolog [Sicstus 93] que possui mais recursos.

Agradecimento:

À Profa. Maria do Carmo Nicoletti da UFSCar, São Carlos, pelas críticas e sugestões relacionadas a este trabalho.

Referências

[Arity 92] Arity Corporation. *The Arity/Prolog Programming Language.* 1992.

[Castineira 91] Castineira, M.I.; Monard, M.C.; Nicoletti, M.C. *Aprendizado de Máquina: Descrição e Implementação de um Algoritmo Geral para a Construção de Árvores de Decisão.* Notas do ICMSC-USP, n. 98, p.42, Outubro, 1991.

[Hunt 66] Hunt, E.B.; Marin, J.; Stone, P.J. *Experiments in Induction.* Academic Press, NY¡ 1966.

[Martins 94] Martins, C.A.; Monard, M.C. *Árvores de Decisão Binárias: Descrição e Implementação de Algoritmos para a Construção e Poda no Aprendizado Indutivo de Máquina.* Relatórios Técnicos do ICMSC, n. 17, 35 pg., fevereiro, 1994.

[Michalski 83] Michalski, R.S.; Stepp, R.E. *A Theory and Methodology of Inductive Learning.* Artificial Intelligence, n. 20, p. 111–6, 1983.

[Monard 94a] Monard, M.C.; Martins, C.A.; Nicoletti, M.C. *Descrição do Módulo Gerador de Exemplos de Funções Booleanas do Ambiente Experimental Construtivo de Aprendizado de Máquina.* (em preparação).

[Monard 94b] Monard, M.C.; Martins, C.A.; Nicoletti, M.C. *Projeto e Implementação de Diversas Estratégias de Construção Automática de Features para o Aprendizado de Funções Booleanas.* (em preparação).

[Nicoletti 92] Nicoletti, M.C.; Castineira, M.I.; Monard, M.C. *Descrição e Implementação dos Mecanismos de Janela e Poda em Árvores de Decisão no Aprendizado Indutivo de Máquina.* Notas do ICMSC-USP, n. 98, p. 42, Maio, 1992.

[Nicoletti 92a] Nicoletti, M.C.; Monard, M.C. *Construção Automática de Features no Aprendizado de Funções Booleanas.* Série C&T do ILTC, v. 1, n. 8, p. 1–20, Junho, 1992.

[Nicoletti 92b] Nicoletti, M.C.; Bezerra, P.C.; Monard, M.C. *Learning Boolean Functions Using Prime-implicants as Features.* Proceedings International AMSE Conference on "Signals, Data, Systems", Chicago, USA, September 2-4, AMSE Press, v. 1, p. 177–83, 1992.

[Nicoletti 93a] Nicoletti, M.C. *Compilação de Conhecimento em Aprendizado Baseado em Explicação.* Monografia do Exame de Qualificação de Doutorado, IFQSC-USP, Abril, 1993.

[Nicoletti 93b] Nicoletti, M.C.; Monard, M.C. *The Role of Description Languages in Inductive Concept Learning.* Aceito para publicação, I SBAI, Rio Claro, SP, Setembro, 1993.

[Nicoletti 93c] Nicoletti, M.C.; Monard, M.C. *Empirical Evaluation of Two Pruning Methods Applied to Constructive Learning of Boolean Functions.* Anales del Primer Congresso Internacional de Informatica, Computacion y Teleinformatica, INFORMATICA'93, Mendoza, Argentina, p. 33–42, Junho, 1993.

[Nicoletti 94] Nicoletti, M.C. *Ampliando os Limites do Aprendizado Indutivo de Máquina através das Abordagens Construtiva e Relacional.* Tese de doutorado, IFQSC-USP, Maio, 1994.

[Quinlan 79] Quinlan, J.R. *Discovering Rules by Induction from Large Collections of Examples.* Expert Systems in the Micro Electronic Age, D. Michie (Ed.), Edinburgh University Press, Edinburgh, p. 168–201, 1979.

[Schlimmer 86] Schlimmer, J.C.; Fisher, D. *A Case Study of Incremental Concept Induction.* Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA: Morgan Kaufman, p. 496–501, 1986.

[Shaw 90] Shaw, M.J.; Gentry, J.A. *Inductive Learning for Risk Classification.* University of Illinois at Urbana-Champaign, IEEE, p. 47–53, February, 1990.

[Sicstus 93] *Sicstus Prolog.* Swedish Institute of Computer Science, 1993.

[Utgoff 88] Utgoff, P.E. *ID5: An Incremental ID3.* Proceedings of the Fifth International Conference on Machine Learning, University of Michigan, Ann Arbor, MI: Morgan Kaufmann, p. 107–20, June, 1988.

[Utgoff 89] Utgoff, P.F. *Incremental Induction of Decision Trees.* Machine Learning, n. 4, p. 161–86, 1989.