Conteúdo

Capítulo 1

Introdução

A Inteligência Artificial é um campo de estudo que alia a Ciência da Computação, a Engenharia e a Psicologia Cognitiva, de forma a tentar criar agentes capazes de ter comportamentos semelhantes a humanos em determinadas situações.

> "A Inteligência Artificial é o estudo de ideias que, implementadas no computador, lhes permitem realizar os mesmos objectivos que fazem as pessoas parecer inteligentes" - Patrick Winston, ex-director do laboratório de IA do MIT.

A Inteligência Artificial organiza-se em 4 categorias, dependendo dos objectivos do agente:

- Pensar Humanamente;
- Agir Humanamente;
- Pensar Racionalmente;
- Agir Racionalmente;

Capítulo 2

Métodos de Resolução de Problemas

Existem 3 tipos de métodos de resolução de problemas:

1. Métodos Fracos (Pesquisa em profundidade, Pesquisa em Largura, Custo Uniforme...)
2. Métodos Informados (Hill Climbing, Simmulated Anealling, Algoritmos Genéticos/Evolutivos, A*...)
3. Pesquisa Adversarial (Minimax e suas variantes)

De notar que alguns algoritmos se encaixam em mais de um destes métodos, visto que o método em si depende da forma como o algoritmo é usado.

2.1 Sistemas de Produções

Os Sistemas de Produções são a arquitectura mais básica dos Sistemas de Inteligência Artificial.

Segundo esta arquitectura, um problema é descrito como um espaço de estados e as transições entre estes e o conhecimento é representado através de lógica (predicados, lógica difusa...) e estruturas de informação.

Sendo assim, a solução do problema pode ser obtida percorrendo o espaço de estados desde o estado inicial até o estado final (armazenando o caminho percorrido, visto este representar a solução).

De notar que, dependendo do problema, pode ser importante associar a cada estado uma função de mérito/custo (por exemplo, num problema de pesquisa de caminho mais curto, cada estado tem um determinado custo, que corresponde à distância percorrida). Também pode ser importante armazenar o caminho percorrido desde o estado inicial até o estado final. Isto normalmente é feito colocando em cada estado um apontador para o seu predecessor mais promissor.

Exemplo

Consideremos o clássico problema dos missionários e canibais:
Numa margem direita de um rio encontram-se 3 missionários, 3 canibais e um barco. Tanto os missionários e os canibais desejam atravessar para a margem esquerda, no entanto, o barco apenas consegue carregar 2 pessoas de cada vez, e se numa margem estiverem mais canibais que missionários, os canibais comem os missionários dessa margem. De notar que para o barco atravessar o rio precisa de levar, pelo menos, uma pessoa

- Representação de um estado: estado(Número de missionários na margem direita, Número de canibais na margem direita, Margem do barco)
- Estado Inicial: estado(3,3,direita)
- Estado Final: estado(0,0,esquerda)
- Transições:
  - estado(M,C,direita) -> estado(M-X,C-Y,esquerda) se 0<=X<=2, 0<=Y<=2, 0<X+Y<=2, M-X>=C-Y, 3-(M-X)>=3-(C-Y).
  - estado(M,C,esquerda) -> estado(M+X,C+Y,direita) se 0<=X<=2, 0<=Y<=2, 0<X+Y<=2, M+X>=C+Y, 3-(M+X)>=3-(C+Y).

Os Sistemas de Produções são compostos por:

- Memória de Trabalho: Colecção de factos verdadeiros;
- Memória de Regras: Transições entre estados;
- Motor de Inferência;

Um Sistema de Produções é considerado puro se possuir as seguintes características:

- Acessibilidade Total;
- Independência Total;
- Sistemática ou Heurística;

A existência de uma sistemática ou de uma heurística permite que alguns problemas de complexidade exponencial possam ser simplificados em problemas de complexidade polinomial.

2.2 Árvores ou Grafos

Dependendo do problema, pode ser preferível representá-lo como um grafo ou uma árvore.

É importante notar que:

- Um grafo pode-se transformar numa árvore, aumentando o número de nós, mas levando à simplificação do algoritmo de pesquisa.
- Uma árvore pode-se transformar num grafo, eventualmente reduzindo o número de nós, mas levando a passos mais complexos (detecção de ciclos...)

2.3 Métodos Fracos

Estes métodos usam técnicas genéricas de pesquisa, sendo independentes do problema. Como tal, são também sensíveis a uma explosão combinatória do número de estados a pesquisar, sendo assim necessário ter muito cuidado com estes.

A pesquisa pode ser feita em ambas as direcções, ou seja:

- Por encadeamento directo;
- Por encadeamento inverso;

A direcção escolhida deve ter em conta vários factores, como o factor de ramificação da árvore de pesquisa e a quantidade de estados finais. Para ter uma ideia de qual o melhor processo pode ajudar pensar em semelhanças entre este e o processo de raciocínio real.

2.3.1 Pesquisa em Profundidade

Este algoritmo, também conhecido como "primeiro em profundidade", "Depth-first search" ou simplesmente "DFS", é o método de pesquisa mais simples. Começando num nó (estado inicial):

1. Se o nó for solução, das duas uma: ou termina-se o algoritmo, ou efectua-se backtrack (depende da implementação... o objectivo é encontrar a melhor solução ou apenas uma solução?);
2. Se o nó não tem sucessores, faz-se backtrack;
3. Escolhe-se um filho do nó ainda não visitado (de notar que pode ser necessário verificar se o estado a que esse nó corresponde não foi visitado anteriormente);
4. Repete-se o algoritmo a partir desse filho;
5. Se houver nós por visitar, repete-se 3, se não, faz-se backtrack.

Pode-se ver um exemplo da ordem de pesquisa deste algoritmo na figura 2.1.

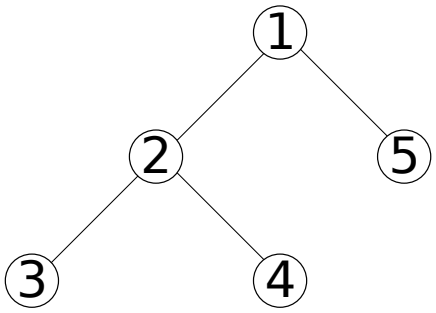


Figura 2.1: Exemplo da Pesquisa em Profundidade

Análise do Algoritmo

Sendo:

- r - factor de ramificação médio;
- p - profundidade máxima;

O algoritmo, para cada ramo, terá de o percorrer até à sua profundidade máxima, logo, terá complexidade temporal $O(r^p)$.

Por outro lado, o algoritmo apenas precisa de armazenar o estado dos filhos (visitado ou não visitado, de forma a poder evitar a visita a estados repetidos) dos nós que fazem parte da solução actual (caminho da raiz até ao nó actual), ou seja, a complexidade espacial será apenas $O(r \times p)$ (Caso o nosso algoritmo não tivesse de se preocupar com estados repetidos, apenas seria necessário armazenar o caminho, ou seja $O(p)$, embora este normalmente não seja o caso).

Este algoritmo não é considerado completo, pois caso um caminho seja infinitamente grande, este nunca encontra a solução.

Também não é óptimo, pois a primeira solução que encontra pode não ser a que necessite de menos passos.

2.3.2 Pesquisa em Largura

Neste algoritmo, ao contrário da pesquisa em profundidade, é dada prioridade aos nós mais próximos da raiz. O algoritmo funciona da seguinte forma (começando na raiz e começando com uma fila vazia):

1. Se o nó for solução, termina o algoritmo (solução garantidamente óptima).
2. Se o nó tiver sucessores, estes (apenas os não visitados, caso seja um grafo) são adicionados ao fim da fila;
3. Retira-se o nó que estiver no topo da fila e repete-se o algoritmo para esse nó.

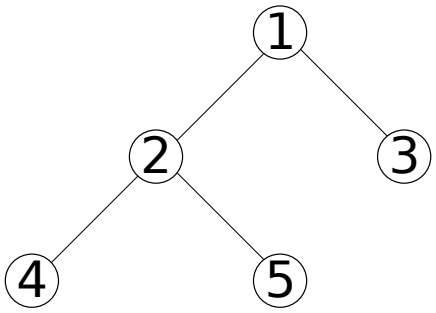


Figura 2.2: Exemplo da Pesquisa em Largura

Este algoritmo também é conhecido como "primeiro em largura", "Breadth-first search" ou simplesmente "BFS",

Pode-se ver um exemplo da ordem de pesquisa deste algoritmo na figura 2.2.

Análise do Algoritmo

Sendo:

- r - factor de ramificação média;
- p - profundidade da solução;

Como este algoritmo apenas visita a árvore até à profundidade da solução, a sua complexidade temporal é $O(r^p)$ ou $O(r^{p+1})$, dependendo do momento em que se faz a verificação da solução (antes ou depois de visitar o nó).

De notar que, mesmo o número de nós visitados ser aproximadamente $\sum_{i=0}^{p} r^i$, apenas se considera o termo mais significativo nas análises de complexidade.

Tal como a complexidade temporal, a complexidade espacial deste algoritmo também é $O(r^p)$ ou $O(r^{p+1})$, visto que a nossa fila vai ter de armazenar todos os nós do nível da solução.

Este algoritmo é óptimo (caso todas as arestas possuam o mesmo custo/valor), no sentido que a primeira solução encontrada é necessariamente a melhor e, se r for finito, é também um algoritmo completo.

2.3.3 Custo Uniforme

Nalguns problemas, nem todas as acções possuem o mesmo custo. Às vezes pode ser preferível avançar 2 estados em vez de apenas 1, pois cada transição pode possuir um peso ou um valor. Um exemplo comum é o cálculo do caminho mais curto entre duas cidades (sendo cada cidade um estado e as estradas as transições).

Para resolver estes problemas podemos usar o algoritmo de custo uniforme. Este algoritmo é, de certa forma, análogo a uma pesquisa em largura, no entanto, a fila passa a ser uma fila de prioridades em que a prioridade corresponde ao

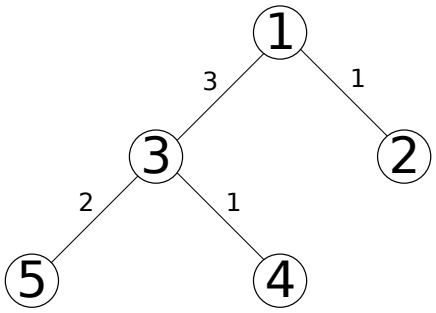


Figura 2.3: Exemplo do Custo Uniforme

custo de efectuar essa acção. Caso todos os custos sejam iguais, este método é igual a uma pesquisa em largura.

O algoritmo funciona da seguinte forma (considerando que começamos na raiz, com uma fila de prioridades vazia e assumindo um problema em que o objectivo é ter o menor custo possível):

1. Se o nó for solução, termina o algoritmo (solução garantidamente óptima).
2. Se o nó tiver sucessores, estes (apenas os não visitados, caso seja um grafo) são adicionados à fila de prioridades com custo=custo actual+custo da transição;
   (a) Caso o nó já se encontre na fila, se o novo custo for menor, actualiza-se o seu valor, se não, não se faz nada.
3. Retira-se o nó que estiver no topo da fila e repete-se o algoritmo para esse nó.

Neste caso, a fila de prioridade terá no topo o nó com menor custo. Caso a intenção seja obter a solução com o maior valor (e não com o menor custo), o algoritmo é semelhante, mas a fila de prioridade é ordenada ao contrário.

Pode-se ver um exemplo da ordem de pesquisa deste algoritmo na figura 2.3.

Análise do Algoritmo

Sendo

- r - factor de ramificação médio;
- c - custo da solução óptima;
- l - limite inferior do custo;

De uma forma muito simples, este algoritmo funciona como uma pesquisa em largura, só que em vez de a pesquisa ser efectuada até à profundidade desejada, é

efectuada até ao custo desejado, ou seja, com esta analogia, poderíamos considerar a complexidade temporal deste algoritmo como $O(r^{c+1})$ e a complexidade espacial como $O(r^{c+1})$ (Em ambos os casos, é importante ter em conta que o 1 adicionado ao expoente depende da implementação).

No entanto, há também que ter em conta que pode ser necessário normalizar o custo (uma árvore em que todas as arestas tenham peso 2 deve ter a mesma complexidade que uma árvore em que todas as arestas tenham peso 1, o que não se verifica com esta formula), pelo que, na realidade, a complexidade espacial é $O(r^{\frac{c}{l}+1})$ e a complexidade temporal também será $O(r^{\frac{c}{l}+1})$.

Regra geral, este algoritmo é mais complexo que a pesquisa em largura.

Tal como na pesquisa em largura, este é um algoritmo óptimo e completo (caso r seja finito), no entanto, este é óptimo mesmo considerando os custos de cada aresta.

2.3.4 Aprofundamento Progressivo

Este algoritmo é uma variante da pesquisa em profundidade, sendo uma combinação de pesquisas em profundidade com uma profundidade limite.

Ou seja, considerando p=1:

1. Efectua-se uma pesquisa em profundidade com profundidade máxima p;
2. Se a pesquisa encontrar uma solução, o algoritmo termina (solução óptima);
3. Incrementa-se p e repete-se o algoritmo;

Este algoritmo permite fazer algo muito interessante: caso cada solução possua um valor/custo associado, o passo 2 pode não retornar imediatamente (visto que deixa de haver garantias que a solução seja óptima), no entanto, passa a anotar a melhor solução encontrada até ao momento. Passando um limite de tempo t, retorna-se a melhor solução encontrada até ao momento (esta técnica também se pode implementar numa pesquisa em largura).

Análise do Algoritmo

Sendo:

- r - factor de ramificação médio;
- p - profundidade da solução;

Este algoritmo tem um comportamento semelhante a várias pesquisas em profundidade com profundidade máxima limitada (até essa profundidade ser igual à profundidade da solução), logo as iterações necessárias serão $\sum_{i=1}^{p} r^i = \frac{(r^{p+1}-1)}{r-1} - 1$, havendo um gasto de memória de $\sum_{i=1}^{p} r \times i = r \times \sum_{i=1}^{p} i = r\frac{i^2+i}{2}$.

No entanto, tendo em conta que os valores menos significativos desaparecem da análise de complexidade, a complexidade temporal pode ser considerada como $O(r^p)$ e a complexidade espacial $O(r \times p)$.

Ao contrário da pesquisa em profundidade, este é um algoritmo óptimo (mais uma vez, se o custo de todos os nós for igual) e completo (para r finito).

2.4 Métodos Informados

Ao contrário dos métodos fracos, os métodos informados possuem alguma informação do problema, nomeadamente algum tipo de função heurística que permita dar um valor a um determinado estado. Com este valor, é possível escolher aquela que se acredita ser a melhor opção, ignorando as outras.

É de notar, no entanto, que a função heurística não deve ser uma função qualquer. Uma boa heurística tem as seguintes propriedades:

1. A heurística deve tentar ser mais simples que o cálculo do valor real do estado (que, por vezes, pode corresponder à solução do problema) sendo, no entanto, o mais próxma do valor real possível (quanto mais próxima do valor real for a heurística, mais depressa o método converge para a solução).

2. A heurística deve ser admissivel (optimista), ou seja:

   (a) Caso se pretenda minimizar o custo, a função heurística é sempre menor ou igual ao custo real;

   (b) Caso se pretenda maximizar o valor, a função heurística é sempre maior ou igual ao valor real;

3. No caso de pesquisa em grafos, a heurística deve ser consistente (monótona), ou seja:

   (a) Para todos os sucessores N de um estado E, o custo estimado de chegar de E ao objectivo nunca é maior que a soma do custo de ir de E até N e o custo estimado de ir de N até ao destino.

   (b) Para todos os sucessores N de um estado E, o valor estimado de chegar de E ao objectivo nunca é menor que a soma do valor de ir de E até N e o valor estimado de ir de N até ao destino.

Se uma heurística for consistente, é imediatamente admissivel.

Uma forma melhor de entender a consistência é pensar em distâncias físicas entre dois pontos. Consideremos a nossa função heurística a distância em linha recta entre os 2 pontos (claramente, esta heurística é optimisita, pois a distância real não pode ser menor que a nossa função).

Tendo em conta o exemplo da figura 2.4, a nossa heurística é completa se $h(1,F) \leq 1 + h(2,F) \wedge h(1,F) \leq 2 + h(3,F)$. Mais uma vez, considerando a heurística (linhas a tracejado) a distância em linha recta e o custo real (linhas preenchidas) a distância real (que pode incluir curvas), é fácil entender o que representa uma heurística consistente.

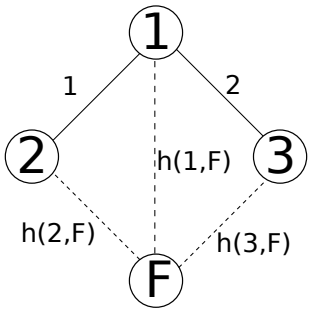


Figura 2.4: Exemplo de uma heurística consistente

Uma forma simples de pensar numa heurística para um problema é retirar regras ao problema. Por exemplo, no problema dos missionários e canibais a nossa função heirística poderia ser "quantas viagens são necessárias ignorando o facto dos canibais comerem os missionários". Esta, no entanto, é uma má heurística, visto simplificar demasiado o problema. Em problemas mais complexos em que há bastantes regras este método torna-se mais simples de aplicar.

Alguns métodos que iremos ver a seguir ("Hill Climbing", "Simulated Annealing" e "Algoritmos Evolutivos") são chamados de "meta-heurísticos". Isto quer dizer que são algoritmos genéricos que, tendo em conta uma heuristica, vão tentando melhorando iterativamente uma solução candidata.

2.4.1 Hill Climbing

Este algoritmo necessita já de algum conhecimento do problema, nomeadamente, de ter alguma forma de receber algum tipo de feedback para saber se um movimento é favorável ou não (por exemplo, através de uma função que diga o valor previsto para cada nó).

O algoritmo usa este conhecimento local para fazer a melhor escolha a determinado momento, ou seja, começando no estado inicial (dependendo do problema, o estado inicial pode ser um estado aleatório):

1. Se o nó for solução (ou se não tiver sucessores, sendo um máximo local), termina o algoritmo (solução localmente óptima, pode não ser a solução ideal).

2. Escolhe o estado sucessor mais próximo da solução (variante "Steepest Ascent") ou simplesmente um estado sucessor melhor (Hill Climbing básico) e repete o algoritmo para esse estado.

Este algoritmo apenas retorna a solução óptima em funções bem comportadas, ou seja, em que o único máximo local é o máximo global, caso contrário, pode ficar "preso" num máximo local, nunca convergindo para a solução (uma forma de tentar resolver isto é executar o algoritmo a partir de vários estados iniciais escolhidos aleatóriamente).

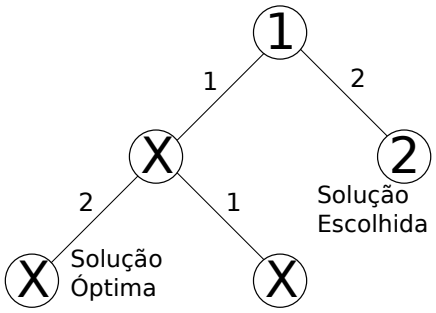


Figura 2.5: Exemplo de Hill Climbing

Pode-se ver um exemplo da ordem de pesquisa deste algoritmo na figura 2.5.

Análise do Algoritmo

Sendo:

- r - factor de ramificação médio;
- p - profundidade do máximo encontrado;

Tendo em conta que este algoritmo apenas tem de seguir o melhor caminho (ou seja, fazer uma comparação com todos os sucessores dos nós de cada estado do percurso), a sua complexidade temporal é $O(r \times p)$. Já espacialmente, apenas necessita de manter armazenado o estado dos sucessores actuais (para os comparar), logo a sua complexidade espacial é $O(r)$.

Isto é muito bom, pois todos os métodos de pesquisa apresentados anteriormente possuiam uma complexidade temporal de crescimento exponêncial, sendo que agora possuímos um algoritmo com complexidade polinomial.

No entanto, como já foi dito antes, este algoritmo não é óptimo, podendo mesmo não chegar a encontrar a solução ideal, visto tratar-se de um algoritmo ganancioso. Pode, no entanto, também não ser completo, se considerarmos um caso em que haja um caminho infinito sempre com filhos de tamanho positivo. Na figura 2.6 é possível ver um exemplo em que este algoritmo não é nem óptimo nem completo.

2.4.2 Simulated Annealing

Visto que o Hill Climbing pode ficar preso em máximos e mínimos locais com bastante facilidade, é necessário arranjar uma maneira de desviarmos o nosso algoritmo destes máximos.

O algoritmo chamado de "Simulated Annealing" faz exactamente isso.

Fazendo a analogia com a metalurgia, os átomos do metal inicialmente encontram-se em todos em mínimos locais. Ao aquecer o metal, o ferreiro faz com que a temperatura aumente, fazendo com que os átomos se movam

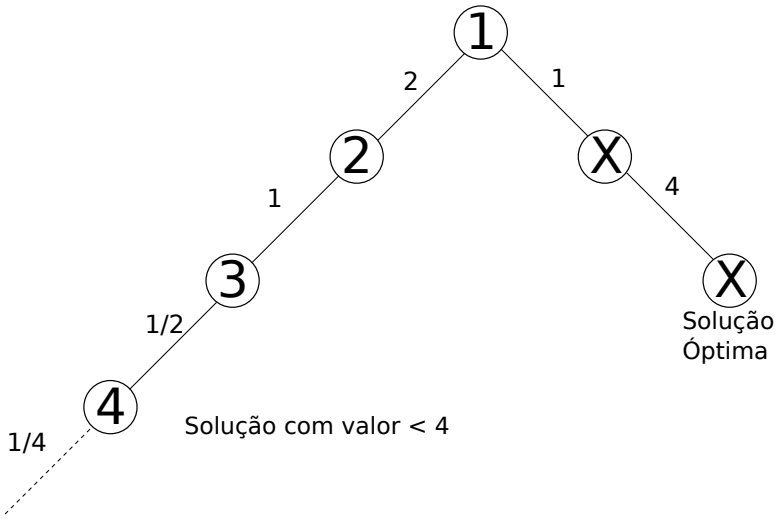


Figura 2.6: Exemplo de Hill Climbing não óptimo e não completo

aleatoriamente e de forma intensa, saindo dos seus mínimos locais. Ao arrefecer lentamente o metal, os átomos vão se movimentando de forma cada vez menos aleatória, tendo a possibilidade de ir parar a novos mínimos locais que sejam melhores que os mínimos iniciais.

Para continuar com esta analogia, vamos considerar um problema em que o objectivo é encontrar o mínimo global.

O algoritmo tem um parametro global T (Temperatura) que vai reduzindo com o tempo. Quanto maior for a temperatura, maior é a possíbilidade de se darem mudanças de estado aleatórias. Mantendo esta analogia, vamos considerar também que o valor de cada estado é a sua energia (E), sendo que queremos obter a menor energia possível.

Sabendo isto, o algoritmo funciona da seguinte maneira, começando no estado inicial (dependendo do problema, o estado inicial pode ser um estado aleatório):

1. Se se atingir a condição de paragem (encontrada uma solução, encontrada uma solução suficientemente boa, timeout...), termina o algoritmo.
2. Escolhe-se aleatoriamente um sucessor do estado actual:
   (a) Se o novo estado for melhor que o actual, esse será o próximo estado.
   (b) Caso contrário, com probablilidade $e^{-\left|\frac{\Delta E}{T}\right|}$, esse estado é o próximo estado.
3. Actualiza-se o valor da temperatura (diminui-se a temperatura por um determinado factor dependendo da implementação).
4. Repete-se o algoritmo para o próximo estado.

Análise de complexidade

Sendo este um algoritmo probabilistico, não será aqui apresentada a sua complexidade.

É de notar no entanto que, mesmo sendo este algoritmo melhor que o Hill Climbing, a sua natureza probabilística faz com que também só seja óptimo e completo em funções bem comportadas.

2.4.3 Algoritmos Evolutivos

Algoritmos evolutivos são baseados na ideia de evolução e selecção natural. Antes de mais, é de notar que, ao contrário do que o senso comum pode levar a pensar, agoritmos genéticos são apenas um subconjunto de algoritmos evolutivos, sendo que a diferença entre estes vai ser explicitada mais à frente.

Como poderá ser de imaginar, tal como na selecção natural, estes algoritmos são lentos e podem levar a soluções dificeis de explicar (mesmo que funcionem). Sendo assim, pode surgir a dúvida: "Quando devem ser estes algoritmos utilizados?".

Uma resposta comum a esta pergunta é: "Os algoritmos evolutivos nunca são a melhor solução, no entanto são sempre a segunda melhor solução", ou seja, estes algoritmos são úteis quando:

- O espaço de pesquisa é demasiado grande e complexo para se utilizar os outros métodos de pesquisa;
- Não é necessário uma solução óptima, apenas uma boa solução;
- Temos capacidade computacional para implementar o algoritmo paralelamente (algoritmos genéricos são fácilmente implementados de forma concorrente, é só dividir os nossos "indivíduos" por várias máquinas/processadores).

Os algoritmos evolutivos podem ser, muito por alto, descritos da seguinte forma:

1. População Inicial;
2. Selecção;
3. Emparelhamento;
4. Crossing-Over;
5. Mutação;
6. A solução é suficientemente boa?
   (a) Se sim, termina.
   (b) Se não, volta a 2.

Em seguida, estas fases serão explicadas em detalhe, sendo que a figura 2.7pode ajudar a uma melhor compreensão destas.

2.4.3.1 Representação dos indivíduos

Antes de mais, para conseguir resolver problemas através de algoritmos evolutivos, é necessário generalizar os possíveis estados e soluções sob a forma de indivíduos. Estes podem ser definidos de várias formas: Como uma string, um número, um objecto...

Cada elemento da representação de um individuo passa a representar um gene.

No caso especifico de algoritmos genéticos, a representação de cada uma destas caracteristicas é feita em binário, passando cada 1 e 0 a ser chamado de alelo. Nestes algoritmos, as variáveis reais normalmente são convertidas de virgula flutuante para virgula fixa. Para algumas caraceristicas que necessitem de mais de 1 bit, pode também ser necessário definir genes dominantes (00 - Verde, 01 - Azul, 10 - Vermelho, 11 - ?) ou fazer algum cuidado especial nas fases seguintes, para evitar estados inválidos.

2.4.3.2 Função de adaptação

Para que estes algoritmos avancem, é necessário uma função de adaptação que receba um individuo e retorne um valor real que corresponda à qualidade desse individuo. Por exemplo, quanto maior este valor, mais apto é o individuo (dependendo da implementação, pode ser mais apto o individuo com o valor mais próximo de 0).

Quanto mais precisa for esta função, mais rapidamente o algoritmo converge para uma solução.

2.4.3.3 Selecção

Esta é a primeira fase do algoritmo, onde se escolhem quais os indivíduos que são mais aptos e vão fazer parte da próxima geração (ou "reproduzir-se", como se verá no passo seguinte).

Existem várias formas de selecção:

- Selecção elitista: Escolhe apenas os melhores indivíduos e repete-os as vezes necessárias;
- Selecção probabilística: sendo f a função de adaptação, um individuo X é escolhido com probabilidade $\frac{f(I_X)}{\sum_{i=0}^{N} f(I_i)}$ (N - número de indivíduos)

É importante que o número de indivíduos se mantenha constante, logo se a população inicial possuir N indivíduos, é necessário fazer N selecções (mesmo que algumas sejam repetidas). Caso contrário, haveria o risco de o algoritmo parar por "extinção".

2.4.3.4 Emparelhamento

A seguir à selecção é necessário definir que indivíduos se emparelham e com quem se emparelham.

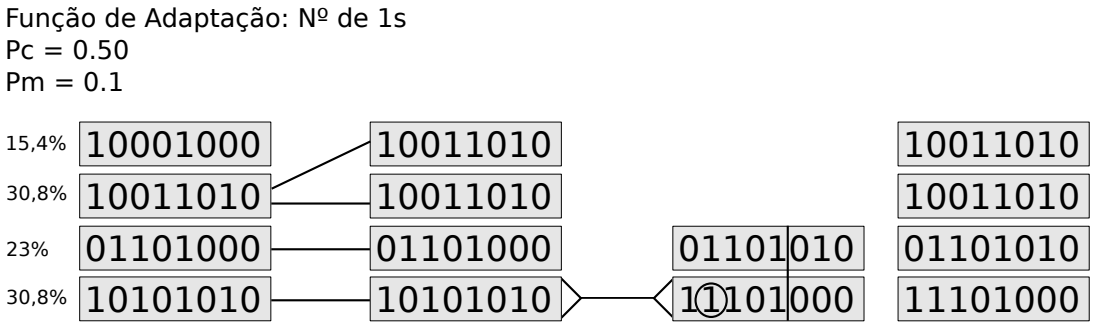


Figura 2.7: Exemplo da aplicação de Algoritmos Evolutivos

Para isso, precisamos de uma probabilidade de cruzamento Pc (geralmente alta). Com esta probabilidade, dos indivíduos selecionados, apenas alguns se poderão reproduzir.

Desses indivíduos, é necessário escolher quem se emparelha com quem. Esta escolha pode ser tanto aleatória como ordenada.

De notar que, por vezes, o número de indivíduos pode ser impar, ficando um indivíduo por emparelhar.

2.4.3.5 Crossing-Over

Estando o emparelhamento feito, para cada par, temos de efectuar o crossing-over (cruzamento de genes ou alelos) de forma a fazer um novo individuo.

Para isso, precisamos de uma estratégia para definir o ponto de crossing-over e, dependendo desta, de uma probabilidade de crossing-over Px.

- Estratégia UX (uniform crossover): havendo N genes num individuo, gera-se um número aleatório X (de distribuição uniforme) entre 0 e N-1. O ponto de crossover será entre o gene 0 e o gene X.
- Estratégia HUX (half uniform crossover): com probabilidade Px, o gene de um individuo é trocado pelo gene do outro e vice-versa.

2.4.3.6 Mutação

Para além do crossing-over, alguns indivíduos poderão sofrer mutações. Como tal, para cada gene de um individuo, este pode ser aleatoriamente alterado com probabilidade Pm (normalmente reduzida).

2.4.3.7 Recursividade

Por fim, na população seleccionada, os filhos substituem os pais, obtendo-se uma nova população.

O algoritmo é depois repetido até ser obtida uma solução desejável.

2.4.4 A*

O algoritmo A* é uma mistura do Custo Uniforme com métodos gananciosos, no sentido em que:

- Custo uniforme: Apenas tem em conta os custos até ao sucessor $C(S, S') = c(S) + c(S')$.
- Métodos gananciosos: Apenas tem em conta a heurística do sucessor $H(S, S') = h(S')$.
- A*: O valor de cada sucessor é dado por $f(S, S') = C(S, S') + H(S, S')$.

Ou seja, se todos os nós tiverem custo 0, o algoritmo passa a ser uma pesquisa gananciosa e se a heurística for constante, o algoritmo é igual ao Custo Uniforme. Se todos os nós tiverem custo constante e a heurística for constante, este algoritmo passa a ser uma Pesquisa em Largura.

Esta algoritmo utiliza duas listas:

- Lista aberta: Nós a expandir (semelhante ás filas da Pesquisa em Largura e do Custo Uniforme). Deve ser uma fila de prioridades ordenada por f;
- Lista fechada: Nós já expandidos (pode ser representada implicitamente como um estado do nó)

O algoritmo funciona da seguinte forma (tendo uma lista aberta apenas com o estado inicial e uma lista fechada vazia):

1. Se a lista aberta estiver vazia, o algoritmo termina;
2. Retira-se o primeiro estado da lista aberta e adiciona-se-o à lista fechada (ou marca-se como expandido);
3. Se o estado for o estado final, o algoritmo retorna;
4. Caso contrário, para cada sucessor que não esteja na lista fechada:
   (a) Calcula-se o seu possível novo valor estimando, $f'(S') = f(S, S')$
   (b) Se o sucessor não estiver na lista aberta ou $f'(S')$ (novo valor estimado para S') for melhor que $f(S')$ (valor estimado para S' anterior), o sucessor é adicionado (ou actualizado) na lista aberta.
5. Repete-se o algoritmo.

Análise do Algoritmo

A complexidade do algoritmo A* depende muito da heurística, no entanto, para todos os casos, este algoritmo é bastante dispendioso na memória, visto que tem de armazenar informação de todos os estados até chegar ao estado final. Como tal, em problemas de larga escala são usadas outras variantes deste algoritmo como o IDA* (Baseado no Aprofundamento Progressivo, mas com custo máximo em vez de profundidade limite).

2.5 Pesquisa Adversarial

Nos casos apresentados anteriormente, o nosso objectivo era apenas ter o melhor valor possível, sem qualquer entrave, no entanto, isso nem sempre se aplica à realidade.

Em casos de competição poderemos ter um adversário que nos quer dificultar a vida (ou, pelo menos, conseguir o melhor valor possível para ele). Nestes casos os algoritmos apresentados anteriormente deixam de funcionar, pois a cada passo o conjunto de estados possíveis é alterado de acordo com os interesses do adversário.

Vamos apenas estudar um tipo muito restrito de jogos, nomeadamente jogos de soma-nula (se o jogador 1 vai para um estado mais favorável, o jogador 2 vai para um estado mais desfavorável), com informação perfeita (sabemos tudo o que se passa no jogo) e determinísticos (uma acção não tem resultados aleatórios).

2.5.1 Minimax

O algoritmo utilizado para resolver este problema é o minimax (ou uma das variantes deste).

Em teoria, este algoritmo funciona da seguinte forma: A cada um dos estados finais associa-se um valor (por exemplo, 1=Vitória, 0=Empate, -1=Derrota), que depois é propagado para os estados predecessores tendo em conta as possíveis jogadas do adversário de forma a saber a que valor um estado nos pode levar. Isto funciona muito bem para jogos curtos, como o jogo do galo.

No entanto, para jogos com um espaço de estados muito maior (que é o caso mais comum), isto torna-se impossível, sendo que é necessária uma função heurística para associar valores a estados intermédios, havendo assim uma profundidade limite.

Para simular o nosso comportamento e o do adversário, este algoritmo associa um tipo cada nível da árvore de pesquisa (que representa o turno de um jogador): minimizador ou maximizador, sendo que o valor de cada estado é escolhido tendo em conta esta função, como pode ser visto na figura 2.8.

O funcionamento do algoritmo é o seguinte (iniciando na raiz do nó):

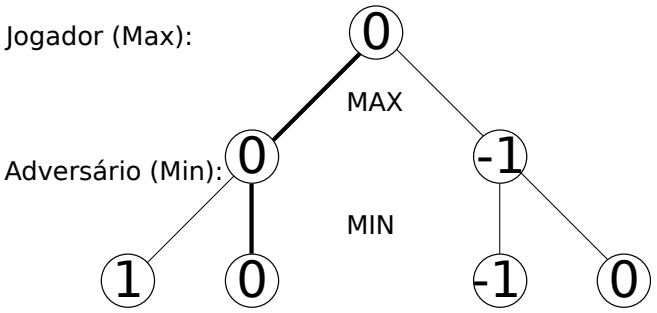
1. Se a profundidade limite tiver sido atingida (ou for um estado final), o valor do nó é calculado através da heurística;

2. Se o nível for maximizador, aplica-se o minimax a cada um dos seus sucessores e o valor do nó passa a ser o maior valor dos sucessores;

3. Se o nível for minimizador, aplica-se o minimax a cada um dos seus sucessores e o valor do nó passa a ser o menor valor dos sucessores;

Análise do Algoritmo

Sendo:



Neste exemplo, o jogador apenas pode empatar, sendo que o jogo decorrerá da seguinte maneira:

- O Jogador joga para a esquerda (Se jogar para a direita, arrisca-se a perder);
- Se o adversário jogar para a esquerda perde (1), logo joga para a direita, para garantir o empate.

Figura 2.8: Exemplo do Algoritmo MiniMax

- r - factor de ramificação médio;
- l - profundidade limite

Sendo esta implementação do algoritmo, de certa maneira, análoga à pesquisa em profundidade, é fácil de concluir que este possui complexidade temporal $O(r^l)$ e complexidade espacial $O(r \times l)$.

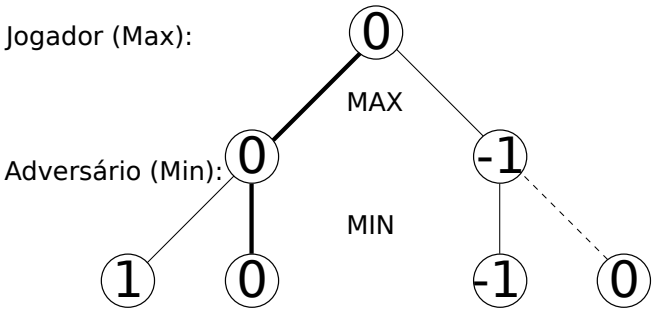
Isto não é nada agradável, especialmente se tivermos em conta que alguns jogos como o xadrez têm um factor de ramificação média de cerca de 35. Com uma profundidade limite de 6 jogadas (jogador humano relativamente aceitável), teríamos $35^6 = 1.838.265.625$ iterações. Como tal, este algoritmo tem de ser optimizado.

2.5.2 Minimax com Alpha-Beta Pruning

Analisando de novo o exemplo da figura 2.8, é possível ver que um dos testes poderia ter sido cortado, evitando cálculos desnecessários, como se pode ver na figura 2.9:

O Alpha-Beta Pruning baseia-se nesta possibilidade de cortes, de forma a reduzir o espaço de pesquisa. Para isso, passa-se a ter uma variável alpha, que representa a melhor jogada que o jogador maximizante já assegurou, e uma variável beta, que representa a melhor jogada que o jogador minimizante já assegurou.

Sendo assim, o nosso algoritmo passa a funcionar da seguinte forma (começando com alpha e beta da raiz a $-\infty$ e ∞ respectivamente):



Neste caso, o jogador vê o que acontece se jogar para a esquerda, e conclui que dessa jogada consegue tirar um empate.
Depois o jogador vê o que acontece se jogar para a direita e, logo no primeiro teste, conclui que esta jogada só pode levar a uma derrota (visto que, se o adversário consegue vencer, vai ignorar outras jogadas que sejam piores para ele). Sendo assim como o jogador já sabe que consegue um empate, não precisa de testar mais jogadas.

Figura 2.9: Exemplo do Algoritmo Minimax com cortes Alpha-Beta

1. Se a profundidade limite tiver sido atingida (ou for um estado final), o valor do nó é calculado através da heurística;
2. Se o nível for maximizador, enquanto alpha<beta e para cada sucessor:
   (a) Aplica-se o minimax ao sucessor (passando-lhe o valor actual do alpha e do beta);
   (b) Actualiza-se o alpha deste nó com o máximo entre o alpha anterior e valor do sucessor;
   (c) Se alpha<beta, o seu valor é o melhor valor dos sucessores. Se não, o valor é alpha (corte beta).
3. Se o nível for minimizador, enquanto alpha<beta e para cada sucessor:
   (a) Aplica-se o minimax ao sucessor (passando-lhe o valor actual do alpha e do beta);
   (b) Actualiza-se o beta deste nó com o mínimo entre o beta anterior e valor do sucessor;
   (c) Se alpha<beta, o seu valor é o melhor valor dos sucessores. Se não, o valor é beta (corte alpha).

Análise do Algoritmo

Sendo:

- r - factor de ramificação médio;

- l - profundidade limite

Esta optimização faz com que, em teoria, a complexidade temporal passe para $O(r^{\frac{l}{2}})$ e complexidade espacial passe a ser $O(\frac{r \times l}{2})$.

No entanto, há que ter em conta que isto depende muito da ordem com que os estados forem pesquisados, sendo que, na realidade, a complexidade pode ser muito maior ou muito menor que a prevista teoricamente.

Uma das optimização que pode ser feita a este método é a chamada "killer heuristic", que consiste em ordenar (ou, pelo menos, tentar ordenar) os nós da árvore de pesquisa por ordem crescente nos níveis maximizantes e ordem decrescente nos níveis maximizantes.

2.5.3 Negamax

O algoritmo negamax é muito semelhante ao minimax, tendo apenas a diferença que, em vez de separar cada nível entre maximizante e minimizante, trata todos os níveis como maximizantes. No entanto, entre cada nível, inverte o sinal dos valores.

Isto é só uma questão de implementação, para permitir que o algoritmo funcione sem ter de saber em que nível está, não tendo praticamente nenhuma influência no desempenho.

2.6 Método da Decomposição

Por vezes, os problemas que queremos estudar são demasiado complexos para serem resolvidos por uma máquina só, por isso o ideal é separá-los em problemas mais simples, que possam ser resolvidos em paralelo por várias máquinas.

Isto pode acontecer, por exemplo, devido a haver vários estados iniciais possíveis, ou por determinado nó ter demasiados filhos e necessitarmos de dividir o seu trabalho entre diferentes processadores.

Geralmente, isto é feito da seguinte maneira:

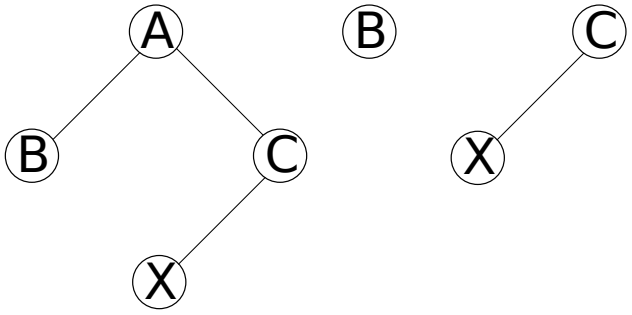
1. Divide-se a base de dados num conjunto de elementos E.
2. Enquanto houver algum elemento Ei que não satisfaça a condição final:
   (a) Seleccionar um Ei que não satisfaça a condição final.
   (b) Retira-se o elemento Ei da lista E.
   (c) Aplicam-se todas as acções possíveis a Ei e armazena-se o resultado na lista E'i
   (d) Adiciona-se E'i à Lista E.

Pode ser visto um exemplo deste método na figura 2.10.

Este método, no entanto, pode trazer alguns problemas em algoritmos como o A*, em que pode ser necessário alterar o valor dos nós (por haver um caminho mais curto para eles do que se pensava), sendo necessário utilizar uma variante

Considerando as seguinte transições possíveis:
A -> BC
A -> C
B -> D
C -> X

Considerando que a condição de paragem é "Ser igual a B"
Considerando que na base de dados inicial temos "A B C"



Estados da Base de Dados:
A B C
B C B X
B X B
B B

Figura 2.10: Exemplo do Método da Decomposição

deste que permita passar novos valores de f para outros processos e revisitar nós.

2.7 Satisfação de Restrições

Alguns problemas são baseados num conjunto de restrições, pelo que deixa de ser suficiente ver os estados como uma "caixa negra", da qual apenas era associado um valor graças a uma função heurística, passando agora a ser necessário que o nosso algoritmo de pesquisa consiga saber realmente o que representa cada estado e os seus valores internos. Os algoritmos de satisfação de restrições voltam a ser genéricos, não sendo necessário uma heurística específica a cada problema para funcionarem.

Num problema de satisfação de restrições existem três componentes:

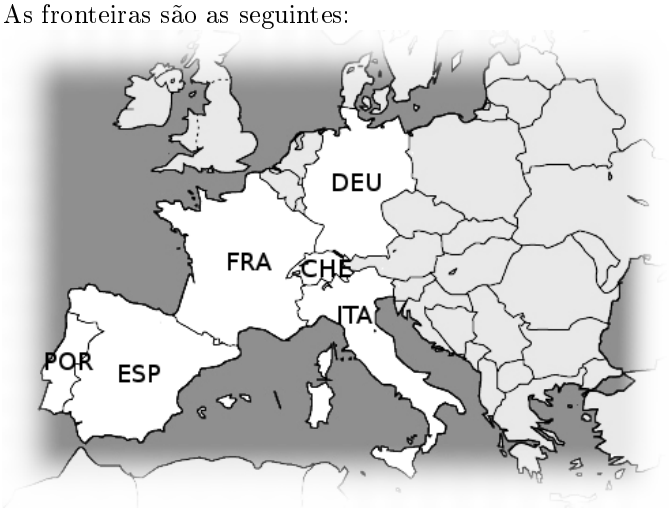
- Um conjunto de variáveis V;
- Um conjunto de domínios de variáveis D;
- Um conjunto de restrições entre variáveis R, sendo que cada restrição pode ter uma ou mais variáveis;

Estes problemas podem normalmente ser representados por um grafo, em que cada variável é um nó (com um domínio associado) e cada restrição é uma aresta entre essas variáveis.

Uma das vantagens de usar estes algoritmos ao contrário dos algoritmos normais de pesquisa é a capacidade de reduzir em muito a quantidade de estados a pesquisar, visto que algumas transições se tornam imediatamente impossíveis. Não é preciso andar um caminho todo para saber se é possível ou não chegar a uma solução.

Exemplo

Problema da Cor dos Países:
Temos 6 países: Portugal (POR), Espanha (ESP), França (FRA), Alemanha (DEU), Suiça(CHE) e Itália (ITA). Todos os países precisam de ser coloridos num mapa, sendo que cada país tem de ser colorido de uma cor diferente dos seus vizinhos. Para além disso, cada país só pode ser colorido com as cores da sua bandeira.
As fronteiras são as seguintes:



Devido à quantidade de emigrantes/imigrantes entre França e Portugal, ambos os países deverão partilhar a mesma cor.
Por outro lado, a Alemanha deve ser pintada de preto, a pedido do criador do mapa, por ser uma cor única em relação às outras bandeiras.
Assim sendo, o problema pode ser descrito da seguinte forma:

- $V = \{POR, ESP, FRA, DEU, CHE, ITA\}$
- $D = \{\{R, Y, G\}, \{R, Y\}, \{R, Y, B\}, \{R, Y, Blk\}, \{R, W\}, \{G, W, R\}\}$
- $C = \{POR = FRA, GER = Blk, POR \neq ESP, ESP \neq FRA...\}$

Neste caso, para ajudar à leitura, as restrições foram simplificadas. A representação formal adequada para $POR \neq ESP$ seria $\langle (POR, ESP), POR \neq ESP \rangle$, onde $POR \neq ESP$ pode ser enumerado como $\{(R, Y), (Y, R), (G, R), (G, Y)\}$.

Existem vários tipos de restrições, nomeadamente:

- Restrições unárias: Estas restrições apenas incluem uma variável (por exemplo, a Alemanha quer a cor preta);
- Restrições binárias: Estas restrições relacionam duas variáveis (por exemplo, Portugal quer a mesma cor que a França);
- Restrição global: Estas restrições relacionam um número arbitrário de variáveis (por exemplo, num jogo de sudoku, todos os elementos de uma

linha são diferentes). De notar que, ao contrário do que o nome pode levar a pensar, "global" não significa que tenha de ter em conta todas as variáveis.

Quando o domínio de um nó respeita todas as suas restrições unárias, trata-se de um nó consistente ("node-consistent").

Quando o domínio de um nó respeita todas as suas restrições binárias (ou seja, se para cada aresta há pelo menos uma combinação de variáveis possível), diz-se que o nó é "arc-consistent".

Existem mais dois tipos de consistência ("path-consistent" e "k-consistent") que permitem simplificar ainda mais o grafo, mas podem ser demasiado pesadas para ser utilizadas.

Os problemas de restrições simbólicas (como o problema da coloração de países) são resolvidos da seguinte forma (algoritmo muito simplificado):

1. Tornam-se todas as variáveis "node-consistent" (simplesmente retirar valores do domínio que não sejam compatíveis com restrições unárias);
2. Tornam-se todas as variáveis "arc-consistent" (algoritmo AC3, não apresentado aqui);
3. Selecciona-se um nó não expandido do grafo de pesquisa (a forma como o nó é escolhido depende de uma heurística... Pode ser o com mais restrições por resolver, pode ser o com menos...);
4. Caso o domínio do nó seja vazio, faz-se backtrack;
5. Associa-se um dos valores possíveis a esse nó e actualiza-se o domínio possível dos seus vizinhos;
6. Repete-se o passo 3 até todos os nós terem um valor;

Capítulo 3

Representação do Conhecimento

A Inteligência Artificial encontra-se directamente relacionada tanto com a Engenharia do Conhecimento como com as Ciências da Cognição. Para obtermos agentes inteligentes é necessário compreendermos o que é o conhecimento, como o podemos representar e como é possível manipula-lo.

O desfio, neste caso, encontra-se em arranjar uma representação simbólica para os conhecimentos do mundo real. Existem vários tipos de representação possíveis:

- Exacta (Definida por processos de inferência e lógica clássica);
- Temporária (Definida de forma não monótona. O que é verdade hoje pode não o ser amanhã);
- Incerta (Definida de forma probabilística. Permite definir conceitos difusos, como "frio" ou "tarde");
- Incompleta (Definida com incertezas. Permite analisar casos em que nem tudo é conhecido);

Tendo em conta estas representações possíveis, o conhecimento pode ser representado de duas formas:

- Declarativa: Há uma colecção estática de factos e procedimentos genéricos para os manipular;
- Procedimental: Todo o conhecimento está codificado de forma operacional;

Tendo o conhecimento bem representado, é nos possível criar agentes baseados em conhecimento, que possuem uma base de conhecimentos ("Knowledge Base") e agem conforme os dados que façam parte dela.

Assim sendo, um sistema de conhecimento pode ser descrito como uma associação entre Objectos, Atributos e Valores. É importante ter em conta que, dependendo do nosso sistema, alguns atributos de um objecto podem possuir vários valores.

3.1 Lógica Formal

Uma das formas de representar o conhecimento é com recurso à lógica formal, criando agentes lógicos. O conhecimento destes agentes pode ser definido tanto por lógica proposicional como por lógica de primeira ordem (ou, em casos mais raros, lógica de ordem superior).

Esta representação permite que um agente, a partir de um conjunto de factos, consiga inferir mais factos, aumentando a sua base de conhecimentos de forma autónoma (de notar que este aumento não é um verdadeiro aumento de conhecimento, mas apenas corresponde à descoberta de conhecimento não explicito).

3.1.1 Lógica Proposicional

A lógica proposicional funciona da seguinte forma:

- A base de conhecimento possui frases atómicas (proposições);
- Cada proposição só pode ter um valor: Verdadeiro ou Falso;
- Uma ou mais proposições podem criar uma nova proposição (frase complexa) através de operadores lógicos;

Os principais operadores lógicos são os seguintes:

- $\overline{a}$ - Negação de a
  - por vezes representado como $\neg a$ ou $\sim a$;
- $a \vee b$ - a ou b (ou inclusivo)
  - por vezes representado como $a + b$;
- $a \underline{\vee} b$ - ou a ou b (ou exclusivo)
  - por vezes representado como $a \oplus b$ ou $a \neq b$;
- $a \wedge b$ - a e b
  - por vezes representado como $a \times b$, $a \cdot b$ ou $a \& b$;
- $a \Rightarrow b$ - a implica b;
  - por vezes representado como $a \rightarrow b$;

- $a \Leftrightarrow b$ - a é igual a b;
  - por vezes representado como $a \leftrightarrow b$, $a = b$ ou $a \equiv b$;

Com estes operadores é possível inferir novos factos a partir da base de conhecimentos.

3.1.2 Lógica de Primeira Ordem

Infelizmente, em problemas mais complexos, a lógica proposicional não é o suficiente para representar tudo, pois não tem nenhuma forma de representar vários objectos sem ter de criar uma proposição especifica para cada um (devidamente acompanhada com novas regras).

Como tal, é necessário utilizar uma lógica mais poderosa. A lógica de primeira ordem permite a existência de relações, que por sua vez levam à existência de funções e objectos. Uma relação é representada sob a forma de $relação(predicado1, predicado2...)$. Isto também faz com que apareçam dois novos operadores:

- $\exists x\ R(x)$ - Existe um x para o qual R(x) é verdadeiro;
- $\forall x\ R(x)$ - Para todos os valores de x da base de conhecimentos, R(x) é verdadeiro (normalmente usado em expressões do tipo $\forall x\ R(x) \Rightarrow S(x)$, ou seja, para todos os valores de x em que R(x) é verdadeiro, S(x) também é verdadeiro);

Com isto, é possível ter afirmações genéricas na base de dados, como "Todas as pessoas altas, com um peso inferior a 60Kg, são magras", o que não seria possível apenas com lógica proposicional.

3.2 Redes Associativas/Semânticas

Por vezes o nosso problema pode ter diversos termos que estão associados entre si por diferentes tipos de relações. Nestes casos, podemos utilizar redes associativas.

Estas redes não são mais que um grafo dirigido em que cada nó representa um termo ou objecto e cada aresta (devidamente etiquetada) representa a relação entre eles. No entanto, mesmo sendo um modelo tão simples, estas redes permitem representar modelos bastante complexos, representando categorias e objectos de uma forma muito próxima do conhecimento natural, como pode ser visto na figura 3.1.

Para além de ser uma representação poderosa, também é mais perceptível que a lógica de primeira ordem.

Por fim, é também uma representação bastante semelhante ao paradigma de programação orientada a objectos, o que pode facilitar a sua implementação nalguns casos.

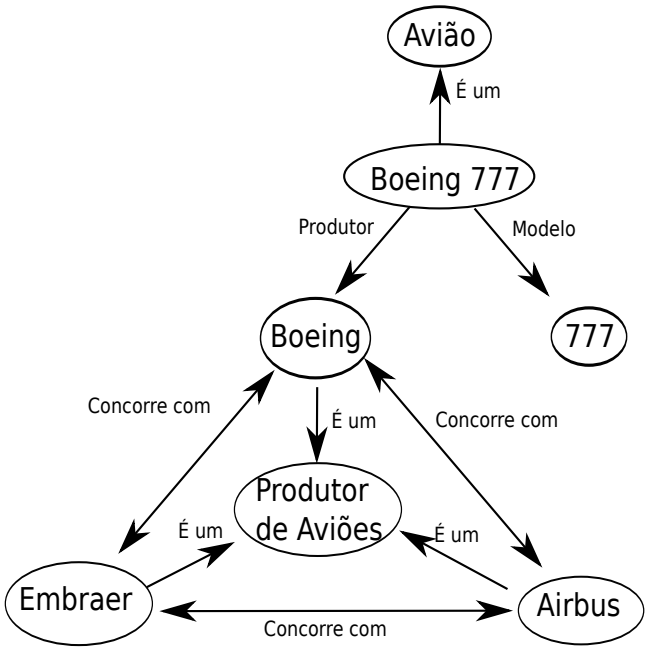


Figura 3.1: Exemplo de uma rede associativa

3.3 Produções

Para alguns problemas não possuímos um conjunto de conceitos, mas sim um conjunto de regras que nos permitem chegar a conclusões, por exemplo, para efectuar um diagnóstico. A este tipo de representações chamamos de "Regras de Produções".

Neste tipo de sistemas, o conhecimento é representado sobre a forma de regras do tipo "Se X, então Y, senão Z", sendo X uma condição booleana e Y e Z conclusões a tirar.

Estes sistemas, embora agora possam parecer bastante redutores, vão se provar muito úteis mais à frente quando forem abordados métodos de aprendizagem que permitem gerar regras de forma automática (nomeadamente, o método ID3 e o método C4.5).

3.4 Frames (Enquadramento)

Muitas vezes o tipo de dados que queremos representar já se encontra bem definido: Sabemos os atributos de cada objecto e o domínio destes, sendo que a única coisa que nos interessa é enquadrar os objectos nestes quadros bem definidos.

Os problemas de enquadramento, tal como as redes associativas, possuem

algumas parecenças com o paradigma orientado a objectos, no sentido em que cada objecto se encaixa num frame (algo análogo a uma classe). Para além disso, embora de uma forma talvez não tão intuitiva como nas redes associativas, este método também possui herança (por exemplo, um frame pode possuir um atributo "pai", cujo valor seja um objecto do qual são herdadas propriedades).

Exemplo

Supondo o seguinte frame para representar um animal:

- Animal:
  - nível (D={Espécie, Subespécie, Raça});
  - pai (D=Animal);
  - nome (D=String);
  - patas (D=Inteiro)

A nossa base de conhecimentos pode conter:

- Lobo = {Espécie,NULL,"Canis Lupus",4}
- Cão = {Subespécie,Lobo,"Canis Lupus Familiaris",NULL}
- Boxer = {Raça,Cão,"Boxer",NULL}

Ao implementar uma representação deste tipo, a aplicação deve ser capaz de responder a perguntas do tipo:"Que animais possuem 4 patas?".

3.5 Lógica Difusa e Raciocínio Incerto

No mundo real, nenhuma fonte de informação é 100% segura, há sempre um factor de incerteza associado a cada informação, mesmo que este seja infinitesimalmente pequeno. Nesses casos, podemos considerar que a informação que temos é de confiança, mas e nos outros casos? Como podemos dar respostas com um determinado factor de certeza associado?

Isto não se aplica só às respostas, mas também ao conhecimento. Como posso representar um conhecimento incerto como "frio", "tarde" ou "alto"? E se eu não tiver a certeza absoluta de um conhecimento que estou a transmitir à máquina?

Consideremos também o seguinte exemplo, um dentista está a fazer um diagnóstico, e ele sabe que:

$Dor\ de\ dentes \Rightarrow Cavidades$

No entanto, esta regra por si só está errada, pois uma dor de dentes pode estar associada a vários problemas:

$Dor\ de\ dentes \Rightarrow Cavidades \vee Gengivas\ Inflamadas \vee Abcesso...$

Como tal, para termos uma regra de diagnóstico verdadeira, necessitaríamos de uma lista praticamente ilimitada de problemas.

No entanto, podemos fazer o contrário, e tentar tornar o nosso diagnóstico (efeito->causa) num modelo causal (causa->efeito):

Cavidades $\Rightarrow$ *Dor de dentes*

Mas esta representação também não está correcta, pois nem todas as cavidades causam dor de dentes.

Como pode ser visto, a lógica proposicional não nos permite resolver de forma eficiente problemas de diagnóstico. Então, qual é a solução?

A solução encontra-se em modelos de lógica difusa.

3.5.1 Teoria dos Conjuntos Difusos

Para melhor representar este tipo de informação recorremos à teoria dos conjuntos difusos ("fuzzy sets"). Segundo esta teoria, cada variável possui um grau de pertença a um conjunto, podendo não pertencer completamente a este (pode ser 80% verdadeira e 20% falsa). Assim, é possível saber a probabilidade de a nossa solução ser verdadeira.

Por exemplo, o nosso conjunto de pertença pode ser:

$$alto(Pessoa) = \begin{cases} 1 & altura(Pessoa) > 1.85m \\ \frac{altura(Pessoa) - 1.65m}{1.85m - 1.65m} & 1.65m \leq altura(Pessoa) \leq 1.85m \\ 0 & altura(Pessoa) < 1.65m \end{cases}$$

Ou seja, temos aqui uma função muito simples para definir se uma pessoa é ou não alta (poderíamos ter, no entanto, uma função com um crescimento muito mais suave ou que dependesse de outros aspectos, como a idade).

Há que ter atenção que os operadores usados na lógica proposicional necessitam de ser adaptados para a lógica difusa:

- $\overline{a} = 1 - a$
- $a \wedge b = min(a, b)$
- $a \vee b = max(a, b)$
- (Os restantes operadores podem ser deduzidos a partir destes)

De notar que estes operadores continuam a funcionar com valores binários, tornando a lógica difusa numa generalização da lógica proposicional.

3.5.1.1 Desfusificação

Tendo a nossa informação devidamente agrupada em conjuntos difusos, é necessário "desfusificá-la" para não perdermos informação.

Supondo que temos um visor (a 1.70m do chão) que se deve ajustar à altura de um utilizador, com os seguintes conjuntos de pertença:

$$alto(Pessoa) = \begin{cases} 1 & altura(Pessoa) > 1.85m \\ \frac{altura(Pessoa) - 1.65m}{1.85m - 1.65m} & 1.65m \leq altura(Pessoa) \leq 1.85m \\ 0 & altura(Pessoa) < 1.65m \end{cases}$$

$$médio(Pessoa) = \begin{cases} 0 & altura(Pessoa) > 1.80m \\ \frac{1.80 - altura(Pessoa)}{1.80m - 1.70m} & 1.70m \leq altura(Pessoa) \leq 1.80m \\ \frac{altura(Pessoa) - 1.60m}{1.70m - 1.60m} & 1.60m \leq altura(Pessoa) \leq 1.70m \\ 0 & altura(Pessoa) < 1.60m \end{cases}$$

$$baixo(Pessoa) = \begin{cases} 0 & altura(Pessoa) > 1.75m \\ \frac{1.75m - altura(Pessoa)}{1.75m - 1.55m} & 1.55m \leq altura(Pessoa) \leq 1.75m \\ 1 & altura(Pessoa) < 1.55m \end{cases}$$

E as seguintes regras:

- Se alto → sobe visor 15cm;
- Se médio → mantém visor;
- Se baixo → desce o visor 15cm;

Se apenas considerarmos a que conjunto a pertença é maior, acabamos por perder informação valiosa: Imagine-se uma pessoa com 1.76m, vai ser baixo com um grau de pertença de 0, médio com um grau de pertença de 0.4 e alto com um grau de pertença de 0.55, sendo que o visor vai subir 15 cm, sendo que ficará demasiado alto para o utilizador.

A forma mais simples de desfusificação é atribuir um peso a cada decisão. No exemplo anterior, o visor passaria a subir $-15 \times 0 + 0 \times 0.4 + 15 \times 0.55 = 8.25cm$, passando a cerca de 1.78m, o que é muito mais agradável.

Outro método mais correcto de fazer a desfusificação (mas mais complexo) é encontrar o centroide (ou centro de massa, fazendo uma analogia com a física) no gráfico dos nossos conjuntos difusos (ver figura 3.2). Isso é feito da seguinte forma:

- Divide-se a área de pertença em áreas elementares (triângulos e rectângulos);
  - Nota: a área de pertença de um conjunto difuso é a área da função com o topo limitado pelo grau de pertença a esse conjunto.
  - A área de pertença total é a união da área de pertença de todos os conjuntos difusos.
- Para cada uma destas áreas i, calcular a sua área (Ai) e o seu centro ({Xi,Yi});
- Calcula-se a área total: $At = \sum_i Ai$
- O centroide é dado por $\left\{ Xc = \frac{\sum_i Xi \times Ai}{At}, Yc = \frac{\sum_i Yi \times Ai}{At} \right\}$
- Utilizando a coordenada Xc, e tendo em conta a sua distância ao "pico" de cada um dos conjuntos em questão (máximo do conjunto, onde a nossa regra supostamente se aplica melhor), calcula-se a decisão a tomar.

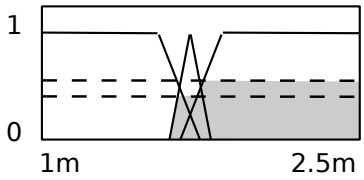


Figura 3.2: Exemplo de áreas de pertença de conjuntos difusos

Infelizmente, muitas vezes as funções não são tão simples como as aqui apresentadas (por exemplo, o grau de pertença a um conjunto pode seguir uma distribuição normal), sendo que passa a ser necessário, para esses conjuntos, calcular o centroide através de integração. Felizmente, regra geral, podemos considerar que os nossas funções de pertença são todas funções continuas do tipo f(x) (ou, pelo menos, é possível dividi-las em funções com estas propriedades), pelo que o calculo do centroide é simplesmente:

$C = \left\{ Xc = \frac{1}{A} \int_{x_0}^{x_1} x\ f(x)\ dx, Yc = \frac{1}{A} \int_{x_0}^{x1} \frac{f(x)^2}{2}\ dx \right\} \text{com } A = \int_{x_0}^{x_1} f(x)\ dx.$

É de notar, no entanto, que estas fórmulas continuam a ter de ser adaptadas tendo em conta o grau de pertença a cada conjunto.

Por outro lado, há que ter em conta que, se o cálculo do integral for efectuado com recurso a métodos numéricos, pode ser preferível simplesmente tentar dividir tudo em áreas elementares.

3.5.2 Incerteza e Teorema de Bayes

Para podermos entender a teoria do raciocínio incerto é necessário saber como calcular uma determinada incerteza.

A melhor forma de quantificar uma incerteza é através da teoria de probabilidades, nomeadamente com recurso ao teorema de Bayes, que nos permite calcular a probabilidade de X sabendo Y.

O teorema é o seguinte:

$P(X|Y) = \frac{P(X \cap Y)}{P(Y)}$

A partir deste teorema, é possível fazer algo muito útil: Sabendo a probabilidade efeito->causa, calcular a probabilidade causa->efeito!

Isto vem de:

- $P(C \cap E) = P(C|E) \times P(E)$
- $P(E \cap C) = P(E|C) \times P(C)$
- $P(C \cap E) = P(E \cap C)$

logo $P(C|E) = \frac{P(E|C) \times P(C)}{P(E)}$ e $P(E|C) = \frac{P(C|E) \times P(E)}{P(C)}$ (caso haja mais evidencias, torna-se $P(X|Y,e) = \frac{P(Y|X,e) \times P(X|e)}{P(Y|e)}$)

Sendo assim, basta-nos ter uma probabilidade condicional e duas probabilidades à priori para ser possível resolver problemas no sentido de diagnóstico ou no sentido causal.

Porque é que isto é importante? Voltando ao exemplo do dentista, é muito mais fácil calcular a probabilidade de uma cavidade causar dor de dentes (sentido causal) do que de uma dor de dentes ser causada por uma cavidade (sentido de diagnóstico), no entanto, a probabilidade que interessa ao dentista é exactamente a de diagnóstico (ele sabe o que o paciente sente, quer saber qual é a causa).

No entanto, isto ainda não resolve todos os nossos problemas, voltando ao caso do dentista, ele precisa de calcular:

$P(Cavidade|Dor\ de\ dentes) = \frac{P(Dor\ de\ dentes|Cavidade)\times P(Cavidade)}{P(Dor\ de\ dentes)}$

O dentista pode saber a probabilidade de um paciente ter uma cavidade dos dentes, tendo em conta a frequência com que esse problema ocorre, mas e a probabilidade de ter dor de dentes?

Outra forma de realizar este calculo é a seguinte, tendo em conta que a soma de uma probabilidade com o seu inverso é sempre 1, podemos usar um factor de normalização:

$P(Cavidade|Dor\ de\ dentes) = \alpha\times P(Dor\ de\ dentes|Cavidade)\times P(Cavidade)$

$P(\overline{Cavidade}|Dor\ de\ dentes) = \alpha\times P(Dor\ de\ dentes|\overline{Cavidade})\times P(\overline{Cavidade})$

onde o factor de normalização é dado por:

$\alpha = \frac{1}{P(Dor\ de\ dentes|Cavidade)\times P(Cavidade)+P(Dor\ de\ dentes|\overline{Cavidade})\times P(\overline{Cavidade})}$

Embora isto possa funcionar bem para casos pequenos como estes, este método não escala, sendo que, para problemas com vários factos, é necessário ter em conta a independência destes.

Na teoria de probabilidades, A independente de B se e só se $P(A) = P(A|B) \wedge P(B) = P(B|A)$.

Tendo isto em conta, somos agora capazes de compreender os principais métodos de lógica difusa.

3.5.3 Factores de Certeza

O modelo de factores de certeza é um modelo relativamente informal de quantificar o grau de certeza numa solução, dado um conjunto de evidências. Consideremos então o seguinte:

Dado um conjunto de evidências E, o nosso factor de certeza de um acontecimento A ocorrer será $-1 \leq FC(A,E) \leq 1$, sendo -1 a descrença total, 1 a crença total e 0 o desconhecimento.

Este factor é calculado tendo em conta a medida de crença "MB" e a medida de descrença "MD", sendo que $0 \leq MB(A,E) \leq 1$ e $0 \leq MD(A,E) \leq 1$, no entanto $MB(A,E) > 0 \Rightarrow MD(A,E) = 0$ e $MD(A,E) > 0 \Rightarrow MB(A,E) = 0$, sendo que, na realidade, só temos de considerar um dos factores de cada vez.

É também importante considerar MB' e MD', que representam o factor de crença (ou descrença) considerando que as evidências são verdadeiras (a utilidade disto será vista mais à frente).

Sendo assim, o factor de certeza é calculado através da fórmula $FC(A,E) = MB(A,E) - MD(A,E)$.

Para este modelo funcionar correctamente, todas as regras necessitam de ser ou puramente de diagnóstico ou puramente causais, tornando-se assim num modelo equivalente a uma rede Bayesiana. Na prática, isto é muito complicado de conseguir, sendo que pode levar a problemas que serão vistos mais à frente.

3.5.3.1 Combinação de Factores de Certeza

As funções de combinação de factores de certeza possuem as seguintes propriedades:

- São associativas e comutativas (independentes da ordem a que as evidências chegam);
- Um acumular de evidências deve levar a um aumento médio da crença;
- Um encadeamento de raciocínios com incerteza deve levar ao aumento da incerteza do resultado final.

Existem 3 formas de combinar factores de certeza:

- Várias regras que levam à mesma conclusão ($A \rightarrow C, B \rightarrow C$):
  - $MB(C, A \cap B) = MB(C, A) + MB(C, B) \times (1 - MB(C, A))$
  - $MD(C, A \cap B) = MD(C, A) + MD(C, B) \times (1 - MD(C, A))$
- Há uma crença numa colecção de factos em conjunto:
  - $MB(A \cap B, C) = min(MB(A, C), MB(B, C))$ e $MB(A \cup B, C) = max(MB(A, C), MB(B, C))$
  - $MD(A \cap B, C) = min(MD(A, C), MD(B, C))$ e $MD(A \cup B, C) = max(MD(A, C), MD(B, C))$
- Há um encadeamento de regras ($A \rightarrow B \rightarrow C$):
  - $MB(C, B) = MB'(C, B) \times MB(B, A)$
  - $MB'(C, B)$, como dito anteriormente, reprezenta o factor de crença em C supondo que B é verdadeiro
  - A aplicação mais comum desta regra é, no entanto:$MB(A, B) = MB'(A, B) \times MB(B)$, considerando que há uma incerteza associada a B
  - Exemplificando, se nos for dito que "Se amanhã chover, o chão vai estar molhado com FC=0.9" e "sabe-se que amanhã vai chover com FC=0.3", então: $MB(Molhado, Chover) = MB'(Molhado, Chover) \times MB(Chover) = 0.9 \times 0.3 = 0.27$.

Exemplo

Um fabricante de automóveis deseja criar um sistema de diagnóstico automático, sendo que, recebendo um input de um utilizador (com um determinado factor de certeza associado), retorna qual o problema mais provável.
As regras são as seguintes:

- Motor Funciona ∩ Bateria Fraca →Bateria Má (FC=1.0)
- Luzes Fracas ∩ Bateria Fraca →Bateria Má (FC=0.8)
- Motor Funciona ∩ Cheiro a Gasolina →Motor Encharcado (FC=0.8)

(Obviamente que um sistema real seria muito mais complexo, no entanto, precisamos de um caso que possa ser rapidamente analisado)
O utilizador sabe que o motor funciona (FC=0.9), não sabe bem o estado da bateria, mas presume que pode estar fraca (FC=0.2), as luzes parecem estar mais fracas que o normal (FC=0.5) e há um leve cheiro a gasolina no ar (FC=0.6), com que certeza o problema é a bateria estar má e o motor estar encharcado?
Para simplificação, vamos considerar as variáveis:

- MF - Motor Funciona
- BF - Bateria Fraca
- BM - Bateria Má
- LF - Luzes Fracas
- CG - Cheiro a Gasolina
- ME - Motor Encharcado

Começando por definir o problema:

1. $MB(MF) = 0.9$
2. $MB(BF) = 0.2$
3. $MB(LF) = 0.5$
4. $MB(CG) = 0.6$
5. $MB'(BM, MF \cap BF) = 1.0$
6. $MB'(BM, LF \cap BF) = 0.8$
7. $MB'(ME, MF \cap CG) = 0.8$

Continuação do Exemplo

Tentando então resolver o problema com as regras indicadas acima:

1. $MB(MF \cap BF) = min(MB(MF), MB(BF)) = 0.2$
2. $MB(LF \cap BF) = min(MB(LF), MB(BF)) = 0.2$
3. $MB(MF \cap CG) = min(MB(MF), MB(CG)) = 0.6$
4. $MB(BM, MF \cap BF) = MB'(BM, MF \cap BF) \times MB(MF \cap BF) = 0.2$
5. $MB(BM, LF \cap BF) = MB'(BM, LF \cap BF) \times MB(LF \cap BF) = 0.8 \times 0.2 = 0.16$
6. $MB(BM, MF \cap LF \cap BF) = max(MB(BM, MF \cap BF), MB(BM, MF \cap LF)) = 0.2$
   (a) Nota: Supondo que tinhamos 2 diagnósticos intermédios BM1 (dado pelo passo 4) e BM2, (dado pelo passo 5), podiamos considerar este passo como $MB(BM, BM1 \cup BM2) = max(MB(BM, BM1), MB(BM, BM2)) = 0.2$.
7. $MB(ME, MF \cap CG) = MB'(ME, MF \cap CG) \times MB(MF \cap CG) = 0.8 \times 0.6 = 0.48$
8. $MB(BM, MF \cap BF \cap LF \cap CG) = MB(BM, MF \cap LF \cap BF) + MB(BM, CG) \times (1 - MB(BM, MF \cap LF \cap BF)) = 0.2$
9. $MB(ME, MF \cap BF \cap LF \cap CG) = MB(ME, MF \cap CG) + MB(ME, BF \cap LF) \times (1 - MB(ME, MF \cap CG)) = 0.48$
10. $MB(BM \cap ME, MF \cap BF \cap LF \cap CG) = min(MB(BM, MF \cap BF \cap LF \cap CG), MB(ME, MF \cap BF \cap LF \cap CG)) = min(0.2, 0.48) = 0.2$
11. $FC(BM \cap ME, MF \cap BF \cap LF \cap CG) = MB(BM \cap ME, MF \cap BF \cap LF \cap CG) = 0.2$

Logo, a nosso factor de certeza 0.2

Este modelo, no entanto, possui alguns problemas nomeadamente quando:

- Há sobreposição de condições:
  - Consideremos o seguinte exemplo:
    - Se um liquido é vermelho, é sangue com FC=0.5;
    - Se um liquido é encarnado, é sangue com FC=0.5;
    - Sabendo que um liquido é vermelho(FC=1.0) e é encarnado(FC=1.0), com que FC é sangue?
    - $FC(S, V \cap E) = MB(S, V) + MB(S, E)(1 - MB(S, V)) = 0.75$
  - Isto é claramente um resultado que não corresponde à realidade.

- Se as regras não forem independentes:
  - Consideremos o seguinte exemplo:
    * Se choveu ontem, a relva está molhada com FC=0.9;
    * Se a relva está molhada, o sistema de rega trabalhou ontem com FC=0.7;
    * Sabendo que choveu ontem (FC=1.0), com que FC o sistema de rega trabalhou?
    * $FC(SR, C) = MB(RM, C) * MB(SR, RM) = 0.63$
  - Mais uma vez, isto não faz muito sentido, pois a chuva não influencia o sistema de rega

3.5.4 Redes de Bayes (Redes Bayesianas)

Uma outra forma de descrever acontecimentos, mais poderosa que o modelo de factores de certeza, são as redes de Bayes. Estas redes podem ser representadas sob a forma de um grafo acíclico dirigido, em que cada nó representa um facto e as arestas representam uma relação de causalidade, com uma probabilidade associada. Pode ser visto um exemplo na figura 3.3.

Estas redes baseiam-se no teorema de Bayes, visto anteriormente.

Este modelo resolve o problema da independência e sobreposição de condições dos factores de certeza, no entanto, apresenta outro problema: Em casos como o da figura 3.3, em que o nosso problema é um grafo multi-conexo ("Cancelar jogo" possui várias causas), este torna-se num problema NP-hard.

A solução nestes casos é tentar unir todas as causas ("Piso Escorregadio" e "Inundação") numa só e calcular as novas probabilidades ($P(PE \cup I|C)$ e $P(CJ|PE \cup I)$), tornando o nosso problema num problema resolvível em tempo polinomial (Perdendo-se, no entanto, algumas informações).

3.5.5 Teoria de Dempster-Shaffer

Na teoria de Dempster-Shaffer, um conjunto de proposições deixa de ter associado só um factor de certeza, passando este a ser substituído por um par {Crença, Plausibilidade}, onde:

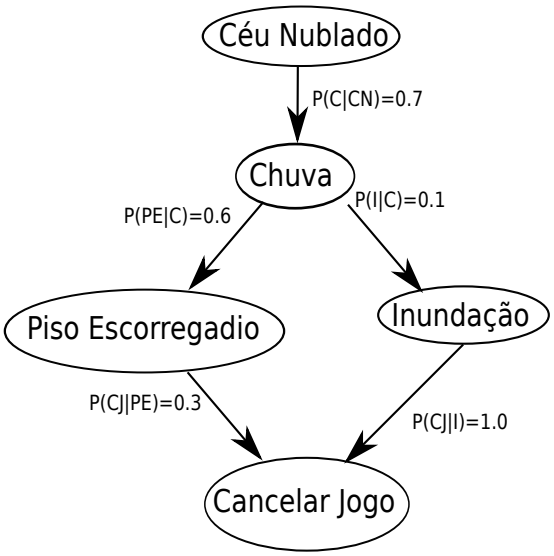
$0 \leq Crença(S) \leq 1$

$Plausibilidade(S) = 1 - Crença(\overline{S})$

Onde a Crença(S) representa a nossa crença numa proposição S e Plausibilidade(S) representa o valor máximo a que essa confiança pode chegar adicionando mais evidências.

À primeira vista, estas formulas não fazem grande sentido, pois a nossa intuição pode-nos levar a pensar que $Crença(S) + Crença(\overline{S}) = 1$, no entanto, isso não é verdade neste modelo, como pode ser visto mais à frente.

Assim, ao contrário dos outros modelos, esta teoria permite-nos distinguir entre incerteza e ignorância. O que estamos a calcular deixa de ser apenas a



Considere-se um grupo de amigos que quer reservar um pavilhão para jogar futsal.
O pavilhão é um bocado antigo, por isso nos dias de chuva o piso costuma ficar escorregadio, e às vezes ocorrem mesmo inundações.
Caso haja uma inundação, são obrigados a cancelar o jogo, caso o piso só esteja escorregadio, o mais certo é jogarem, embora também o possam decidir cancelar.
Com esta rede, sabendo a probabilidade de o céu estar nublado num determinado dia, é possível calcular a possibilidade de o jogo ser ou não cancelado.

Figura 3.3: Exemplo de uma rede de Bayes

probabilidade de S ser verdadeiro, mas também se é ou não necessário obter mais evidências.

A melhor forma de explicar este conceito é apresentando um exemplo:

Exemplo

Pessoa	Nº de Carros	Entre 3 e 5 carros?
A	1 ou 2	Não
B	1	Não
C	4 ou 5	Sim
D	5 ou 6	?
E	6	Não

Tendo em conta a seguinte tabela, com a informação que possuímos, sendo S="Escolhida uma pessoa aleatória, essa pessoa tem entre 3 e 5 carros"
$Crença(S) = \frac{1}{5}$
$Crença(\overline{S}) = \frac{3}{5}$
$Plausibilidade(S) = 1 - \frac{3}{5} = \frac{2}{5}$
Ou seja, por mais evidências que usemos, a nossa crença nunca será superior a 2/5.

A teoria de Dempster-Shaffer é, normalmente, implementada da seguinte maneira:

- Inicialmente, precisamos de um conjunto Θ, que contém todas as hipóteses possíveis (sendo estas mutuamente exclusivas). Este conjunto vai representar o "desconhecido" até ao momento;
- Cada evidência suporta um subconjunto de Θ;
- Inicialmente, apenas se sabe que a crença se encontra entre 0 e 1, não se fazendo mais nenhuma suposição (por exemplo, não se pode assumir que inicialmente, tendo Θ N hipóteses, cada hipótese tem probabilidade 1/N).
- Temos uma função $m(conjunto)$ que mede a densidade de probabilidade de um conjunto, ou seja, a crença nesse conjunto ($Crença(X) = \sum_{Y|Y \subseteq X} m(Y)$, ou seja, a crença é a soma da densidade de todos os subconjuntos desse conjunto).
  - Por exemplo, $Crença(\{A, B\}) = m(\{A\}) + m(\{B\})$.
- Sendo assim, a plausibilidade pode ser calculada através de $Plausibilidade(X) = \sum_{Y|Y \cap X \neq \phi} m(Y)$, ou seja, a crença em todos os conjuntos (incluindo o "desconhecido") que não anulem X.
  - Por exemplo, $Plausibilidade(\{A\}) = m(\{A\}) + m(\{A, B\}) + m(\{\Theta\})$.
- Inicialmente, $m(\Theta) = 1$, no entanto, este valor vai sendo alterando conforme são adicionadas evidências (diminuindo assim o nosso "desconhecido");
- Duas funções de crença m1(X) e m2(Y) podem ser combinadas da seguinte forma:

- $m1(X) \oplus m2(Y) = m1,2(Z) = \frac{\sum_{X \cap Y = Z} m1(X)m2(Y)}{1-K}$
  - $\sum_{X \cap Y = Z} m1(X)m2(Y)$: ou seja, para todas as intersecções dos conjuntos X e Y, o valor da sua multiplicação das suas densidades;
  - $K = \sum_{X \cap Y = \phi} m1(X)m2(Y)$: ou seja, para todos as intersecções vazias dos conjuntos X e Y, o valor das suas multiplicação (utilizado para normalizar os valores);
- ou seja, obtem-se $m1,2(Z)$ onde Z representa as intersecções dos conjuntos X e Y em que, para cada conjunto Z, o seu valor é igual à multiplicação $m1(X)m2(Y)$ ponderada tendo em conta as intersecções vazias.

De notar que, quando não há intersecções vazias (ou seja, não há ignorância), a teoria de Dempster-Shaffer é equivalente ao modelo dos factores de certeza.

Posto desta maneira, a teoria pode parecer bastante confusa, por isso vamos a mais um exemplo prático:

Exemplo

Considerando o conjunto $\Theta = \{Alergia, Gripe, Constipação, Pneumonia\}$
Havendo as seguintes evidências:

- Febre (m1)
  - $m1(\{G, C, P\}) = 0.6$
  - $m1(\Theta) = 0.4$
- Pingo no nariz (m2)
  - $m2(\{A, G, C\}) = 0.8$
  - $m2(\Theta) = 0.2$

Combinando estas duas evidências em m3:

	$m2(\{A, G, C\})$	$m2(\Theta)$
$m1(\{G, C, P\})$	$\{G, C, P\} \cap \{A, G, C\}$	$\{G, C, P\} \cap \Theta$
$m1(\Theta)$	$\Theta \cap \{A, G, C\}$	$\Theta \cap \Theta$

Fazendo as contas:

	$m2(\{A, G, C\}) = 0.8$	$m2(\Theta) = 0.2$
$m1(\{G, C, P\}) = 0.6$	$\{G, C\} = 0.48$	$\{G, C, P\} = 0.12$
$m1(\Theta) = 0.4$	$\{A, G, C\} = 0.32$	$\Theta = 0.08$

Ou seja, a nossa função m3 é definida por:

- $m3(\{G, C\}) = 0.48$
- $m3(\{G, C, P\}) = 0.12$
- $m3(\{A, G, C\}) = 0.32$
- $m3(\Theta) = 0.08$

De notar que não há nenhum subconjunto vazio, logo K=0, o que faz com que, para já, esta teoria seja semelhante ao modelo de factores de certeza. Apresentando um exemplo:
$FC(G|m1 \cap m2) = MB(G|m1) + MB(G|m2) \times (1 - MB(G|m1)) = 0.6 + 0.8 \times 0.4 = 0.92$
$m3(\{G\}) = m3(\{G, C\}) + m3(\{G, C, P\}) + m3(\{A, G, C\}) = 0.92$

Continuação do Exemplo

Suponha-se agora que os sintomas desapareceram depois de o paciente viajar, o que leva a uma nova evidência:

- Viajar (m4)
  - $m4(\{A\}) = 0.9$
  - $m4(\Theta) = 0.1$

Aplicando este novo conhecimento a m3, resulta (esta tabela apresentará directamente os resultados para poupar espaço):

	$\{A\}$	Θ
$\{G, C\}$	$\emptyset = 0.432$	$\{G, C\} = 0.048$
$\{G, C, P\}$	$\emptyset = 0.108$	$\{G, C, P\} = 0.012$
$\{A, G, C\}$	$\{A\} = 0.288$	$\{A, G, C\} = 0.032$
Θ	$\{A\} = 0.072$	$\Theta = 0.008$

Neste caso já ocorrem subconjuntos vazios, sendo que temos de computar K primeiro:
$K = 0.432 + 0.108 = 0.54$
$1 - K = 0.46$
Logo, a nossa função m5 passa a ser:

- $m5(\{A\}) = \frac{0.288+0.072}{0.46} = 0.78$
- $m5(\{G, C\}) = \frac{0.048}{0.46} = 0.10$
- $m5(\{G, C, P\}) = \frac{0.012}{0.46} = 0.03$
- $m5(\{A, G, C\}) = \frac{0.032}{0.46} = 0.07$
- $m5(\Theta) = \frac{0.008}{0.46} = 0.02$

Ou seja, daqui podemos concluir:
$Crença(\{A\}) = 0.78$
$Crença(\overline{\{A\}}) = 0.1 + 0.03 = 0.13$
$Plausibilidade(\{A\}) = 1 - 0.13 = 0.87$

Capítulo 4

Sistemas Periciais

Agora que já temos a capacidade de representar o conhecimento, é importante saber como usá-lo. Os sistemas periciais são os sistemas baseados em conhecimento mais típicos, constituindo algumas das aplicações mais conhecidas e mais práticas da inteligência artificial.

Mas antes de mais, o que é um sistema pericial? Um sistema pericial é um sistema computacional que incorpora o conhecimento simbólico de um ou mais especialistas em um ou mais domínios, podendo depois emular (ou assistir) os especialistas. Estes sistemas normalmente possuem os seguintes módulos:

- Conhecimento simbólico (base de conhecimentos);
- Capacidade de inferência (motor de inferência);
- Heurísticas;
- Explicabilidade (capacidade de explicar o seu processo dedutivo/indutivo);
- Meta-Conhecimento (conhecimento que o sistema tem do seu próprio conhecimento);
- Interface amigável (quanto mais próximo da linguagem natural, melhor);
- Capacidades evolutivas (aprendizagem máquina);

No entanto, há que ter atenção para não confundir sistemas periciais com aplicações que apenas efectuam cálculo numérico ou programas que apenas tratam grandes quantidades de dados. Essas aplicações não possuem muitos dos módulos referidos em cima, como a capacidade de evolução. É importante saber quando utilizar um tipo de sistema ou outro:

- Se a complexidade do tratamento de informação for maior que a quantidade de dados, utilizar sistemas periciais;
- Se a quantidade de dados é muito maior que a complexidade do tratamento de informação, utilizar bases de dados (ou semelhantes);

Sendo assim, os sistemas periciais são uma forma prática de unir o conhecimento de várias fontes e especialidades, permitindo não só aceder a esse conhecimento de uma forma mais natural que o normal acesso a bases dados relacionais, como a evoluir o conhecimento graças às capacidades de aprendizagem.

4.1 Motor de Inferência

O motor de inferência de um sistema pericial tem de ser capaz de, tendo uma base de conhecimentos com conjunto de regras (com premissas e conclusões que se podem retirar) e factos (que podem ter uma incerteza associada), inferir novos factos (eventualmente com uma nova incerteza associada). Para além dos factos da base de conhecimentos e os factos induzidos, o motor de inferência pode, se achar necessário, perguntar novos factos ao utilizador.

A inferência pode ser feita de duas maneiras:

- Encadeamento Directo ("Forward-chaining"):
  - Seleccionar as regras cujas premissas unificam com o estado corrente;
  - Executar as acções correspondentes à conclusão das regras;
  - Repetir enquanto existirem regras aplicáveis;
- Encadeamento Inverso ("Backwards-chaining"):
  - Utiliza um sistema de backtracking (semelhante ao motor de inferência de Prolog);

4.2 Explicabilidade

O sistema pericial tem de conseguir explicar como chegou à solução. Isto é muito importante, pois um especialista que esteja a ser auxiliado por um destes sistemas pode duvidar da origem de um resultado, podendo pretender analisar o raciocínio do sistema para saber se algo está errado ou não.

Existem 2 tipos de explicações:

- Porquê?
  - Explica o que já é conhecido à partida e as regras que foram aplicadas;
- Como?
  - Trace do caminho percorrido durante a inferência;

4.3 Aprendizagem

A implementação de métodos de aprendizagem máquina serão explicados mais à frente, no entanto, de forma muito sucinta, os sistemas periciais possuem 3 formas de adquirir novo conhecimento:

- Por comunicação directa (através de uma linguagem próxima da natural, sendo o input processado e transformado numa representação intermédia que o sistema pericial compreenda);
- Por indução (usando indução lógica ou algoritmos como ID3 ou C4.5)
- Por adaptação (usando técnicas como aprendizagem por reforço)

É de notar, no entanto, que as redes neuronais (também a estudar mais à frente) não são consideradas como uma forma válida para aprendizagem em sistemas baseados em conhecimento. Isto deve-se ao seu funcionamento não ser propriamente baseado no processamento do conhecimento em si mas sim em valores que, quando aplicados ao conhecimento, retornam os valores desejados, sendo assim um sistema "estúpido" que apenas reage da forma correcta.

4.4 Meta-Conhecimento

O meta-conhecimento é o conhecimento que o sistema pericial possui sobre o seu próprio conhecimento, por exemplo:

- Quais são os conceitos?
- Quais são os factos actuais?
- Quais são as regras?
- Quais são as fórmulas para calcular a aptidão?
- Quais as heurísticas usadas?

Normalmente, este conhecimento é dividido em 3 níveis (do mais abstracto para o mais concreto):

- Arquétipos (os conceitos);
- Categorias (os moldes que cada facto pode ter);
- Instâncias concretas (os factos em si);

4.5 Shells

Uma shell é um programa que inclui:

- Motor de Inferência;
- Tratamento de regras;
- Interface (normalmente com menus);
- Capacidade explicativa do tipo "Porquê?" (algumas incluem também o tipo "Como?");
- Editor de regras e, eventualmente, frames;

Estes programas permitem a ligação com linguagens de programação comuns, como C, Java ou Prolog, permitindo muito rapidamente desenvolver sistemas periciais ou sistemas semelhantes.

Capítulo 5

Linguagem Natural

Como foi falado anteriormente, num sistema pericial é importante haver alguma forma de um especialista comunicar com a máquina, sendo que quanto mais natural for esta comunicação, mais facilidade o especialista tem em exprimir-se, havendo menor perda de informação.

Não só nos sistemas periciais, como também noutros campos da inteligência artificial, é importante haver um sistema que seja capaz de compreender a linguagem natural humana, permitindo-nos assim transmitir informação $Humano \rightarrow Máquina$ e/ou $Máquina \rightarrow Humano$, sendo capaz de compreender frases e textos.

As aplicações do processamento da linguagem natural são as mais variadas, por exemplo:

- Interfaces com bases de dados de uma forma natural;
- Sistemas periciais;
- Sistemas de perguntas e respostas para aceder à informação;
- Classificação do texto (correcção gramatical e sugestões);
- Recuperação de informação perdida;
- "Text Mining"/"Data Mining" (muito em voga recentemente em sistemas como "twitómetros" ou medidores de popularidade).

Existem dois tipos de aproximação há linguagem natural:

- Aproximação estatística:
  - Exemplo 1: Analisar todos os textos de Shakespeare e anotar a probabilidade de depois da palavra A aparecer a palavra X ou mesmo de depois de aparecer a sequência A, B e C aparecer a palavra Y. Com isto é possível classificar um texto como sendo ou não de Shakespeare com um determinado factor de certeza.

  - Exemplo 2: Sabendo a probabilidade de na língua portuguesa uma letra A ser seguida de outra letra X (ou mesmo de uma sequência de caracteres A, B e C ser seguida de uma letra Y), calcular o factor de certeza de uma frase ser portuguesa.
  - O exemplo 1 não tem grande interesse prático (até porque não é capaz de analisar o contexto dos textos, podendo assumir muitas coisas erradamente), embora possa vir a ter as suas utilidades, nomeadamente, sabendo quais as palavras mais prováveis de se seguir a outras em determinada linguagem, é possível decifrar palavras-chave baseadas em frases (normalmente longas, sendo o brute-force normal impraticável), passando-se a fazer um brute-force por palavra e não por letra.
  - O exemplo 2 permite-nos, por exemplo, detectar automaticamente a linguagem de uma página web, fazer brute-force a uma cifra de césar sabendo em que idioma foi escrita a mensagem original (testar todas as combinações e retornar a com maior factor de certeza) ou mesmo adaptar um algoritmo de brute-force de passwords letra a letra de forma a escolher primeiro os conjuntos de letras mais prováveis.

- Aproximação linguística:

  - Esta aproximação é baseada na capacidade de análise sintáctica com recurso a gramáticas com contexto (ou gramáticas livres de contexto, se apenas nos interessar uma linguagem próxima da natural).

5.1 Representação

Existem várias formas de o nosso sistema de linguagem natural representar as frases:

- Análise superficial (apenas analisa o que está escrito e responde de acordo);
- Semântica restrita (analisa a frase e o contexto desta, no entanto esta tem de estar de acordo com uma semântica restrita);
- Compreensão profunda (analisa tanto a frase como o seu contexto, suportando frases ambíguas);

Uma forma de testar um sistema de linguagem natural é o famoso teste de Turing, que consiste no seguinte:

- Colocam-se várias pessoas em salas isoladas, com um computador à frente;
- Cada pessoa comunica ou com o nosso sistema de linguagem natural ou com outra pessoa (sem saber com qual delas está a falar);
- No final, cada pessoa deve dizer se acha que estava a falar com uma máquina ou com outra pessoa;

Obviamente que este teste não testa só as capacidades de linguagem natural, obrigando o nosso sistema a ter módulos extra, como uma base de conhecimentos e um motor de inferência. No entanto, este teste é importante para identificar o poder da nossa representação. Considerando um exemplo de um teste de Turing para cada uma das representações:

- Análise Superficial:
  - Humano: "Quem é o presidente dos Estados Unidos?"
  - Máquina: "Porque quer saber quem é o presidente dos Estados Unidos?"
  - Humano: "Para saber se estou a falar com uma máquina ou não"
  - Máquina: "Saber se está a falar com uma máquina é importante para si?"
- Semântica Restrita:
  - Humano: "Quem é o presidente dos Estados Unidos?"
  - Máquina: "Barack Obama"
  - Humano: "Qual é o nome completo dele?"
  - Máquina: "Por 'dele' posso presumir que quer dizer 'de Barack Obama' ?"
  - Humano: "Sim"
  - Máquina: "Barack Hussein Obama II"
- Compreensão Profunda:
  - Humano: "Quem é o presidente dos Estados Unidos?"
  - Máquina: "Barack Obama"
  - Humano: "Qual é o nome completo dele"
  - Máquina: "Barack Hussein Obama II"

Como se pode ver pelo exemplo, a análise superficial apenas tem a informação que vem da frase anterior, tentando "fazer de conta" que possui conhecimento. Pode conseguir essa ilusão ao fazer perguntas, mas perde-se completamente quando tem de responder a algo.

No exemplo da semântica restrita, o nosso sistema já possui algum conhecimento, mas tem problemas com ambiguidades. Já é um modelo mais realista, mas continua a ser fácil de detectar que é uma máquina. É de notar, no entanto, que dependendo da implementação deste comportamento, ele também poderia responder correctamente à pergunta (visto que possui um estado interno) ou simplesmente responder "Não sei a quem se refere com 'dele'.", obrigando o humano a reformular a pergunta.

Com a compreensão profunda a conversa já se torna muito mais próxima de uma conversa natural.

5.2 Processamento de Linguagem Natural

Usando a aproximação linguística para o processamento de linguagem natural, esta encontra-se dividida em 3 fases: Análise Lexical, Análise Sintáctica e Análise Semântica. Para quem já tiver tido aulas de Compiladores ou semelhantes, já devem ter alguma ideia de como é que isto vai funcionar.

5.2.1 Análise Lexical

Antes de mais, precisamos de ter definidos na nossa base de conhecimentos os seguintes elementos:

- Nomes;
- Pronomes;
- Verbos (e respectivas conjugações);
- Adjectivos;
- Artigos;
- Proposições;
- Conjunções;

Para ajudar mais à frente na análise semântica, é aconselhável já ter alguma informação associada a cada palavra (género, número...). Se possível, o ideal é esta classificação ter o máximo de informação possível (semelhante à TLEBS).

Embora a análise lexical seja feita normalmente com recurso a palavras, pode ser boa ideia utilizar expressões regulares para classificar alguns tipos especiais de nomes (como números de telefone, sites, emails...).

5.2.2 Análise Sintáctica

Para já, vamos apenas considerar gramáticas livres de contexto ("Context-Free Grammars" ou "CFGs") para a nossa análise sintáctica, representadas através da forma de Backus-Naur ("BNF").

A forma de Backus-Naur funciona da seguinte forma: A nossa gramática possui um conjunto de símbolos terminais (palavras, normalmente representadas em letras minúsculas) e símbolos não terminais (compostas por produções: conjuntos de palavras e símbolos não terminais, normalmente representados por letras maiúsculas ou palavras capitalizadas em "upper camel case"). Apresentando o exemplo simples da gramática portuguesa:

$Frase \rightarrow SintagmaNominal\ SintagmaVerbal$
$SintagmaNominal \rightarrow Artigo\ SintagmaNominal1$
$SintagmaNominal \rightarrow SintagmaNominal1$
$SintagmaNominal1 \rightarrow ConjuntoAdjectivos\ Nome$
$SintagmaNominal1 \rightarrow Nome$

$ConjuntoAdjectivos \rightarrow Adjectivo|Advérbio\ Adjectivo$
$SintagmaVerbal \rightarrow Verbo|Verbo\ SintagmaNominal$
$Artigo \rightarrow a|o$
$Nome \rightarrow joao|pedro|maria|bola$
$Advérbio \rightarrow muito$
$Adjectivo \rightarrow grande|pequena$
$Verbo \rightarrow corre|chuta$

(Antes de mais, como pode ser visto, na representação "BNF", uma regras com várias produções possíveis pode ser representada de duas formas: ou através de várias regras com o mesmo nome, ou através do operador "|")

Com esta gramática, já conseguimos validar frases como:"o grande pedro chuta a muito pequena bola" ou "a maria corre", como pode ser visto na figura 5.1.

No entanto este tipo de gramáticas possuem alguns problemas:

- As gramáticas normalmente possuem inúmeras excepções, sendo que para serem representadas, é necessário recorrermos a uma CFG extremamente complexa:
  - Por exemplo, este caso aceita o sintagma nominal "pequena bola", mas não aceita "bola pequena". Isto podia ser resolvido adicionando a regra $SintagmaNominal1 \rightarrow Nome\ ConjuntoAdjectivos$, o que é bastante simples.
  - Agora imagine-se que pretendemos adicionar cores à nossa gramática. As cores são um exemplo interessante, pois tanto são adjectivos como nomes (tanto podemos dizer "o amarelo é bonito" como "o carro é amarelo"), no entanto, isto faria com que a nossa gramática não só aceitasse o sintagma nominal "amarelo amarelo", como ficasse com uma produção ambígua (amarelo amarelo é um Nome seguindo de um Adjectivo ou um Adjectivo seguido de um Nome?). A solução para isto seria não considerar as cores nem um nome nem um adjectivo, mas um grupo à parte, alterando radicalmente a gramática e aumentando em muito a sua complexidade.
- Aceita frases como "os maria corre o bola", que não fazem sentido pois não há concordância de género/número:
  - Possível de resolver com gramáticas com contexto, a ver mais à frente.
- Pode possuir ambiguidades:
  - Embora a gramática acima não seja o caso, uma gramática mais completa aceitaria as frases equivalentes, como "o joao chuta a bola" e "a bola é chutada pelo joao", havendo duas árvores distintas com o mesmo significado (ou seja, duas representações diferentes para o mesmo valor).

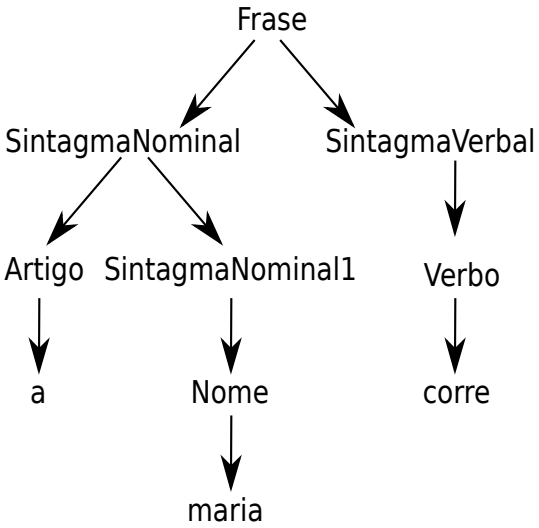


Figura 5.1: Exemplo da decomposição da frase "a maria corre"

5.2.3 Análise Semântica

Caso o a gramática que pretendamos usar não seja tão completa quanto a gramática portuguesa, podemos aproveitar o modelo de gramáticas livres de contexto para retirar informações de uma frase. Por exemplo, se pretendermos apenas implementar um sistema que nos dê informações de um determinado navio, a nossa gramática pode ser:

$Questão \rightarrow QuestãoPropriedade|QuestãoMembroTripulação$
$QuestãoPropriadade \rightarrow qual\ é\ Propriedade\ do\ Navio\ ?$
$QuestãoMembroTripulação \rightarrow quem\ é\ Membro\ do\ Navio\ ?$
$Propriedade \rightarrow o\ tamanho|a\ tonelagem|a\ idade...$
$Membro \rightarrow o\ capitão|a\ captiã|o\ dono...$
$Navio \rightarrow gil\ eanes...$

Estes modelos são bons por duas razões:

- São muito simples, logo tornam-se fáceis de processar (o processamento pode ser feito ao mesmo tempo que a interpretação);
- Não possuem ambiguidade, pois estas deixam de fazer sentido;

No entanto, possuem o inconveniente de, ou serem pouco poderosas (como é o caso acima) ou de precisarem de muitas regras, o que torna a interpretação demorada.

5.2.4 Gramáticas de Caso

Como foi dito acima, um dos problemas das CFG é o tratamento de frases ambíguas, como "o pedro chutou a bola" e "a bola foi chutada pelo pedro", pois

obtemos duas estruturas diferentes (normalmente árvores) que representam o mesmo valor.

A solução das gramáticas de caso é a seguinte: recebendo a árvore da análise (com recurso a uma CFG), identificar qual o agente, o objecto e o verbo (entre outras informações), passando a representar as frases agrupadas pelo verbo, na seguinte forma (considerando as duas frases acima):

$chutar(Agente(pedro), Objecto(bola))$

Sendo assim, a representação de ambas as árvores torna-se idêntica, o que é muito mais agradável de processar.

Num exemplo mais complexo, como "A Liliana cozinhou a galinha durante 30 minutos" o resultado deveria ser algo do género:

$cozinhar(Agente(liliana), Objecto(galinha), Tempo(30minutos))$

5.2.5 Gramáticas Lógicas

Como foi dito antes, é necessário a implementação de gramáticas que não sejam livres de contexto para obtermos frases correctas. Uma das soluções para este problema é utilizar gramáticas baseadas em lógica de primeira ordem, em que, em vez de termos apenas:

$SintagmaNominal \rightarrow Artigo\ SintagmaNominal1$

Temos agora algo como:

$SintagmaNominal(G, N) \rightarrow Artigo(G, N)\ SintagmaNominal1(G, N)$ (Sendo G o género e N o número).

Sendo assim, deixamos de ter uma CFG, tendo agora uma Gramática de Clausulas Definidas ("Definite Clause Grammar" ou "DCG");

Obviamente que, para esta nova gramática funcionar, cada artigo e cada membro do SintagmaNominal1 vai precisar de ter um número e um género associado (como foi dito no final da análise lexical). Para além disso, pode ser conveniente ter mais informações, por exemplo, se um nome é animado ou não, para podermos validar frases do tipo "O urso chuta a bola" e invalidar frases do tipo "A pedra chuta a bola".

A implementação destas gramáticas é feita da seguinte maneira:

- Inicialmente, implementar um analisador sintáctico descendente de forma funcional (para mais informações, procurar por analisadores LL(1) ou LL(K));
- Caso já haja um género/número definido, testar esses argumentos com os novos tokens/funções. Caso contrário, apenas verificar o tipo de token.
- Cada função receba argumentos extra (como género e número) deve também retornar esses argumentos;
- Durante a verificação de tokens, verificar se há concordância.

Um exemplo em pseudo-código:

Exemplo

Nota: Considera-se que a lista de tokens é global.

```
sintagma_nominal() {
Token t=TokenList.nextToken();
Genero g=NULL;Numero n=NULL;
if (t.type==ARTIGO) {
g=t.genero;n=t.numero;
t=TokenList.nextToken();
if (t.type==NOME && t.genero==g && t.numero==n) {
// Retorna verdadeiro
}
else {
// Retorna falso ou lança uma excepção
}
}
// Retorna falso ou lança uma excepção
}
```

Esta implementação pode ser um bocado estranha, sendo que pode ajudar a ver uma implementação em Prolog, onde se evita o problema de ter de argumentos desconhecidos devido à natureza da linguagem:

Exemplo

Nota: o primeiro "argumento" de cada regra são os tokens a processar, e o segundo argumento são os tokens a processar pela próxima regra.

```
sintagma_nominal(X,Y) :- artigo(X,Z,Gen,Num),nome(Z,Y,Gen,Num).
artigo([o|X],X,masculino,singular).
artigo([os|X],X,masculino,plural).
artigo([a|X],X,feminino,singular).
artigo([as|X],X,feminino,plural).
nome([pedro|X],X,masculino,singular).
```

Na prática, a maior parte dos interpretadores de Prolog possui o operador $-->$ que nos permite tornar o código acima em:

Exemplo

```
sintagma_nominal --> artigo(Gen,Num),nome(Gen,Num).
artigo(masculino,singular) --> [o].
artigo(masculino,plural) --> [os].
artigo(feminino,singular) --> [a].
artigo(feminino,plural) --> [as].
nome(masculino,singular) --> [pedro].
```

5.2.6 Quantificação

No entanto, estas gramáticas continuam sem resolver todos os problemas, nomeadamente os problemas de quantificação. Suponha-se a seguinte frase: "Todas as

pessoas estão a ver um barco".

Recorrendo à lógica de primeira ordem, esta afirmação pode ser descrita de duas formas:

- Para todas as pessoas, há uma barco que é visto:$\forall p \in Pessoas \Rightarrow \exists b \in Barcos \wedge Ve(p, b)$
- Existe um barco que todas as pessoas estão a ver:$\exists b \in Barcos : \forall p \in Pessoas \Rightarrow Ve(p, b)$

A solução para este problema passa normalmente por criar representações intermédias "quase lógicas" entre a análise sintáctica e a análise semântica, usando regras de preferência sobre os quantificadores.

5.2.7 Transformações Semânticas

Estando terminada a análise da frase, é necessário que esta seja transformada para que possa ser executada.

5.2.7.1 Three Branched Quantifiers

Este método consiste em agrupar as frases em quantificadores com 3 ramos, na forma quantificador(variável, sintagma nominal, sintagma verbal). Por exemplo, a frase "O professor chumba todos os alunos que não aprendem" resulta em todo(X, e(aluno(X), not(aprende(X)), chumba(professor,X)).

5.2.7.2 Definite Closed-World Clauses

Este método associa respostas a perguntas na forma: *Resposta* $\Leftarrow$ *Condições da pergunta*, por exemplo a pergunta "Quais os oceanos que banham pelo menos 3 países europeus?" torna-se em $resposta(Oc) \Leftarrow oceano(Oc) \wedge cardinalidade(P, pais(P) \wedge oceano(Oc) \wedge banha(Oc, P)) > 2$

Para isto, no entanto, é necessário definir operadores para representar a pergunta, por exemplo:

- Operadores lógicos;
- Operadores de comparação;
- Operadores matemáticos;
- existe(X,T) - Existe um valor de X tal que T seja verdadeiro;
- cardinalidade(X,T) - Para quantos elementos de X T é verdadeiro;
- etc.

Sendo assim, cada determinante corresponde a um quantificador, a um domínio D e um âmbito A.

- Determinantes do tipo "o", "a", "existe"... - $existe(X, D \wedge A)$

- Determinantes do tipo "todo", "qualquer" - $qualquer = not(existe(X, D \wedge not(A)))$

Por exemplo, na frase "Algum professor é português":

- D=professor(X)
- A=português(X)
- Resultado: $existe(X, professor(X) \wedge português(X))$

Capítulo 6

Aprendizagem Simbólica Automática

A aprendizagem humana é algo muito complexo, sendo que não se trata de um processo de aprendizagem só, divide-se em vários tipos de aprendizagem, sendo normalmente dividida em dois tipos:

- Aprendizagem por Aquisição de Conhecimentos (Simbólica):
  - Conhecimento adquirido através da aquisição de novas informações, sendo capaz de aplicar o novo conhecimento em situações novas.
  - Exº: Aprender Inteligência Artificial por um livro.
- Aprendizagem por Refinamento de Habilidade (Não-simbólica):
  - Processo de refinamento de uma habilidade por sucessivas tentativas.
  - Exº: Aprender a andar de bicicleta.

O objectivo da aprendizagem simbólica automática é criar sistemas dotados de capacidades de aprendizagem simbólica, tal como os humanos.

6.1 Taxonomia dos Métodos de Aprendizagem Simbólica Automática

A aprendizagem simbólica automática pode ser classificada de várias formas:

6.1.1 Classificação quanto à Estratégia Usada

- Rote Learning:

  - Consiste simplesmente em aprender por ser programado ("by being programmed") ou por memorização de novos factos ou dados (funciona como uma base de dados apenas).

- Aprendizagem por Conselho/Instrução ("by being told" ou "by instruction"):

  - Um agente exterior, de alguma forma (directamente ou indirectamente utilizável pelo sistema), fornece um conselho. O sistema é depois responsável de actualizar o seu conhecimento (alterando a representação do conhecimento ou por inferência) de acordo com o conselho.

- Aprendizagem por Analogia:

  - Consiste na capacidade de adquirir novos conceitos, factos ou planos por transformação ou ampliação de conceitos que já façam parte do sistema.

- Aprendizagem por Exemplos:

  - Dado um conjunto de exemplos e contra-exemplos, o sistema deve conseguir induzir um conceito geral que verifique os exemplos e descarte todos os contra-exemplos.

- Aprendizagem não Supervisionada (por Observação/por Descoberta)

  - É um método de aprendizagem indutiva capaz de formar teorias e criar critérios de classificação para formar hierarquias taxonómicas sem intervenção externa.

6.1.2 Classificação quanto à Representação do Conhecimento Adquirido

- Parâmetros em Expressões Algébricas;
- Árvores de Decisão;
- Gramáticas Formais;

  - Autómatos finitos, expressões regulares, CFGs...

- Regras de Produção;
- Expressões Baseadas em Lógica Formal e Semelhantes ;
- Grafos e Redes;

6.2 Aprendizagem Dedutiva/Indutiva por Métodos Baseados em Explicações

Para implementar a nossa aprendizagem dedutiva/indutiva por métodos baseados em explicações, vamos utilizar uma estratégia de aprendizagem por exemplos, ou seja, dando um conjunto de exemplos positivos ao nosso sistema de um determinado conceito, este deve ser capaz de deduzir/induzir um conceito genérico definido pela "Teoria do Domínio" que englobe os exemplos.

Existem 3 métodos para isto:

- Generalização Baseada nas Explicações (EBG);
- EBG de exemplos múltiplos (mEBG);
- Indução Sobre as Explicações (IOE);

No entanto, tanto o EBG como o mEBG possuem algumas desvantagens, nomeadamente:

- Raramente identificam a definição correcta;
- O sucesso depende muito de como as regras estão definidas na teoria do domínio.

6.2.1 Generalização Baseada nas Explicações (EBG)

Este método funciona da seguinte maneira: O nosso resolvedor de problemas começa já com uma teoria do domínio correcta, capaz de definir o conceito genérico. No entanto, esta teoria pode ser muito ineficiente, contendo informações desnecessárias (Por exemplo, se estivermos a classificar uma chávena, a nossa teoria inicial pode dizer que uma chávena tem de ser ou branca, ou preta, ou vermelha, ou verde, ou às bolinhas amarelas...).

Depois disto, o algoritmo funciona da seguinte maneira:

1. Recebe um conjunto de exemplos de treino E;
2. Aplica a teoria de domínio para provar que um único exemplo Ei é uma instância do conceito-genérico, criando uma árvore de explicação para esse exemplo;
3. Extraem-se as pré-condições mais fracas WP (as essenciais para cada exemplo continuar a ser uma instância do conceito-genérico);
   (a) Normalmente, caso a árvore de explicação seja uma árvore simples, WP é a conjunção dos literais das folhas.
4. Forma-se uma regra eficiente;
5. O conceito-genérico é convertido numa forma operacional.

Se as pré-condições mais fracas WP forem as condições necessárias e suficientes de um conceito C mais especifico, então $WP(E) = C(E)$.

Consideremos um exemplo, para entender melhor como isto funciona:

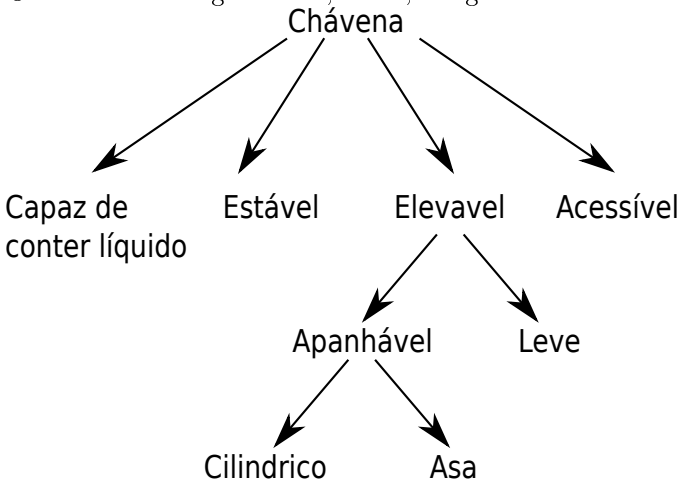
Exemplo

Começando com a nossa teoria de domínio inicial sobre o conceito "Chávena":

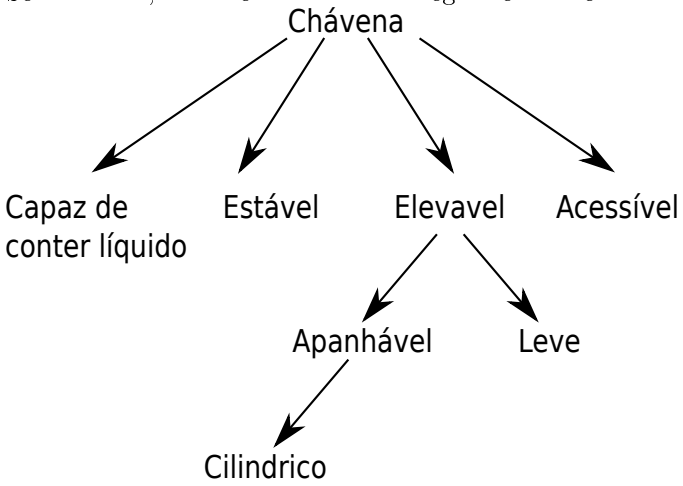
- Regra "Chávena"
  - $chávena(Obj) \leftarrow capaz\ de\ conter\ liquido(Obj) \wedge estável(Obj) \wedge elevavel(Obj) \wedge acessível(Obj)$
- Regra "Capaz de conter líquido"
  - $capaz\ de\ conter\ liquido(Obj) \leftarrow superfície\ não\ porosa(Obj) \wedge fundo\ não\ poroso(Obj)$
  - $superfície\ não\ porosa(Obj) \leftarrow superfície(Obj, Sup) \wedge feito\ de(Sup, Material) \wedge não\ poroso(Material)$
  - $fundo\ não\ poroso(Obj) \leftarrow fundo(Obj, Fundo) \wedge feito\ de(Fundo, Material) \wedge não\ poroso(Material)$
- Regra "Estável"
  - $estável(Obj) \leftarrow fundo(Obj, Fundo) \wedge plano(Fundo)$
- Regra "Elevavel"
  - $elevavel(Obj) \leftarrow leve(Obj) \wedge apanhável(Obj)$
- Regra "Acessível"
  - $acessível(Obj) \leftarrow tem(Obj, Parte) \wedge concavidade(Parte) \wedge para\ cima(Parte)$
- Regra "Leve"
  - $leve(Obj) \leftarrow pequeno(Obj) \wedge superfície\ leve(Obj) \wedge fundo\ leve(Obj)$
  - $superfície\ leve(Obj) \leftarrow superfície(Obj, Sup) \wedge feito\ de(Sup, Material) \wedge material\ leve(Material)$
  - $fundo\ leve(Obj) \leftarrow fundo(Obj, Fundo) \wedge feito\ de(Fundo, Material) \wedge material\ leve(Material)$
- Regra "Apanhável Cilíndrico"
  - $apanhável(Obj) \leftarrow pequeno(Obj) \wedge superfície(Obj, Sup) \wedge cilindrico(Sup) \wedge feito\ de(Sup, Material) \wedge isolante\ térmico(Material)$
- Regra "Apanhável Asa"
  - $apanhável(Obj) \leftarrow pequeno(Obj) \wedge tem(Obj, Parte) \wedge asa(Parte)$
- Sendo:
  - $não\ poroso(plástico) \wedge não\ poroso(aluminio) \wedge não\ poroso(porcelana) \wedge material\ leve(plastico) \wedge material\ leve(porcelana) \wedge material\ leve(aluminio) \wedge isolante\ térmico(plastico) \wedge isolante\ térmico(porcelana)$

Continuação do Exemplo

O nosso conceito genérico é, então, o seguinte:



Consideremos então, que o nosso exemplo de testes (c1) possui as seguintes propriedades:
$superfície(c1, sup1) \wedge feito\ de(sup1, plástico) \wedge fundo(c1, fundo1) \wedge feito\ de(fundo1, plástico) \wedge plano(fundo1) \wedge tem(c1, parte1) \wedge convavidade(parte1) \wedge para\ cima(parte1) \wedge pequeno(c1) \wedge cilindrico(c1)$
Sendo assim, a chávena c1 forma a seguinte árvore:



Sendo as pré-condições mais fracas de c1 obtidos pela conjunção dos literais das folhas:

- Regra "Capaz de conter líquido":$superfície(Obj, S) \wedge feito\ de(S, Ms) \wedge não\ poroso(Ms) \wedge fundo(Obj, F) \wedge feito\ de(F, Mf) \wedge não\ poroso(Mf)$
- Regra "Estável":$fundo(Obj, F) \wedge plano(F)$
- Regra "Elevável":$pequeno(Obj) \wedge material\ leve(Ms) \wedge materia\ leve(Mf) \wedge cilindrico(S) \wedge isolante\ térmico(Ms)$
- Regra "Acessível":$tem(O, P1) \wedge concavidade(P1) \wedge para\ cima(P1)$

Como pode ser visto pelo exemplo, esta implementação tem bastantes problemas, visto que só tem em conta um exemplo, o que nos leva ao próximo método:

6.2.2 EBG com Múltiplos Exemplos (mEBG)

Este método funciona da seguinte maneira:

1. Para cada exemplo, prova-se que faz parte do conceito-genérico e gera-se a sua árvore de prova;
2. Comparam-se todas as árvores de prova geradas e gera-se uma nova árvore A que seja a maior sub-árvore partilhada por todos os exemplos;
3. Aplica-se o EBG como apresentado anteriormente, só que em vez de o aplicar a um exemplo, aplica-se à árvore A.

6.2.3 Indução Sobre Explicações (IOE)

Este algoritmo é bastante semelhante ao mEBG, no entanto, a árvore A é gerada de forma diferente.

Em vez de utilizar uma politica de generalizar a prova o mais possível sem alterar a sua estrutura, o que o IOE pretende fazer é utilizar uma politica mais conservativa, de forma a generalizar a prova apenas o necessário para desenvolver uma prova comum.

Neste caso, para gerar a melhor sub-árvore partilhada, o nosso objectivo é associar a cada constante c_i do domínio e dos exemplos (por exemplo, "plástico") uma variável v_i (por exemplo, "Mat"), criando assim uma substituição Θ.

Uma substituição Θ é gerada da seguinte forma:

1. Inicialmente, o nosso conjunto Θ encontra-se vazio:
2. Para cada variável v_i do nosso conceito C (no exemplo do EBG, "Chávena"), tenta-se fazer uma associação $v_i = c_j$ e adicionar essa associação a Θ, onde c_j é obtido da seguinte maneira:
   (a) Começa-se por considerar que constantes substituíram v_i em cada um dos exemplos cv_{ei} (sendo e o exemplo);
   (b) $c_j$passa a ter o valor da generalização dessas constantes, sendo essa generalização dada por:
      i. Se para todos os exemplos, o valor de cv_{ei} é igual, a generalização é cv_{ei} .
      ii. Se os valorescv_{ei}são todos termos do mesmo functor (exº. elevável(cv_{ei})) a generalização é elevável(generalização(cv_{ei}))
      iii. Caso nada disto se verifique, a generalização é uma nova variável V, que ainda não tenha sido usada em nenhuma transformação.

Vejamos um pequeno exemplo do resultado deste algoritmo:

Exemplo

Atributo	Ex. 1: Bola de Futebol	Ex. 2: Bola de Hóquei	Resultado
Forma	esférica	esférica	esférica
Material	couro	madeira	Mat
Tamanho	médio	pequeno	Tam
Exterior	leve	pesado	Peso
Interior	leve	pesado	Peso

Assim, o nosso novo conceito é "Bola esférica com um exterior igual ao interior".

6.3 Aprendizagem Indutiva

A aprendizagem indutiva é baseada em algoritmos que permitem criar árvores de decisão (ID3, C4.5...). O objectivo destes métodos é adquirir conhecimento através da construção automática de novas regras de conhecimento.

Tendo em conta um grande conjunto de dados com conclusões associadas, se tivermos em conta apenas os atributos significativos, é nos possível construir árvores de decisão e, a partir destas, obter novas regras.

6.3.1 ID3 (Algoritmo de Quinlan)

O algoritmo ID3 é uma forma de criarmos estas árvores de decisão de uma forma sistemática, sendo muito usado para tornar cenários complexos como fundos de investimento em árvores de decisão simples.

O algoritmo ID3 baseia-se no cálculo da entropia E para uma conclusão C sabendo um atributo A , sendo esta dada pela fórmula:

$E(C|A) = \sum_{j=1}^{M} p(a_j) \times \left[- \sum_{i=1}^{N} p(c_i|a_j) \times log_2 \left(p(c_i|a_j) \right) \right]$

Sendo:

- M - número de valores possíveis para o atributo A;
- N - número de valores possíveis para a conclusão C;
- a_j - valor possível j para o atributo A;
- c_i - valor possível i para a conclusão C;

Sabendo isto, o algoritmo funciona da seguinte forma:

1. Calcula-se a entropia para cada atributo;
2. Escolhe-se como raiz da árvore o atributo com menor entropia (ou seja, menor variação de valores), fazendo-se uma divisão de acordo com esse atributo (A);
3. Para cada valor possível do atributo A:

(a) Se só for possível chegar a uma conclusão, essa conclusão é uma folha da árvore.

(b) Se for possível chegar a várias conclusões, repete-se o algoritmo com o sub-conjunto de dados referente a esse valor e unem-se as duas árvores.

Apresentando um exemplo:

Exemplo

Tendo em conta o cenário de um fundo de investimento, em que nos interessa saber qual o valor deste de acordo com a seguinte tabela:

Tipo de Fundo	Taxa Juro	Disp. Monet.	Tensão	Valor
Circ. Interno	Alta	Alta	Média	Médio
Circ. Interno	Baixa	Alta	Média	Alto
Circ. Interno	Média	Baixa	Alta	Baixo
Mina	Alta	Alta	Média	Alto
Mina	Baixa	Alta	Média	Médio
Mina	Média	Baixa	Alta	Médio
Construção	Alta	Alta	Média	Baixo
Construção	Baixa	Alta	Média	Alto
Construção	Média	Baixa	Alta	Baixo

Vamos então, calcular as entropias:

- $E(Valor|Tipo) = p(CI) \times (-p(Baixo|CI)log_2(p(Baixo|CI)) - ...) + ...$
  - $E(Valor|Tipo) = \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{1}{3}log_2(\frac{1}{3}) - \frac{1}{3}log_2(\frac{1}{3})\right) + \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{2}{3}log_2(\frac{2}{3})\right) + \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{2}{3}log_2(\frac{2}{3})\right) = 1.140$
- $E(Valor|Taxa) = p(Alta) \times (-p(Baixo|Alta)log_2(p(Baixo|Alta)) - ...) + ...$
  - $E(Valor|Taxa) = \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{1}{3}log_2(\frac{1}{3}) - \frac{1}{3}log_2(\frac{1}{3})\right) + \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{2}{3}log_2(\frac{2}{3})\right) + \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{2}{3}log_2(\frac{2}{3})\right) = 1.140$
- $E(Valor|Disp) = p(Alta) \times (-p(Baixo|Alta)log_2(p(Baixo|Alta)) - ...) + ...$
  - $E(Valor|Disp) = \frac{6}{9} \times \left(-\frac{3}{6}log_2(\frac{3}{6}) - \frac{2}{6}log_2(\frac{2}{6}) - \frac{1}{6}log_2(\frac{1}{6})\right) + \frac{3}{9} \times \left(-\frac{1}{3}log_2(\frac{1}{3}) - \frac{2}{3}log_2(\frac{2}{3})\right) = 1.279$

Assim sendo, podemos fazer a divisão tanto no tipo de fundo como na taxa de juro.

Continuação do Exemplo

Considerando que efectuamos a divisão na taxa de juro.

- Se a taxa de juro for alta, os valores possíveis para o valor do fundo são: {Alto,Médio,Baixo}
- Se a taxa de juro for média, os valores possíveis para o valor do fundo são: {Médio,Baixo}
- Se a taxa de juro for baixa, os valores possíveis para o valor do fundo são: {Alto,Médio}

Como tal, nenhum dos casos se torna numa folha da nossa árvore, sendo que temos de repetir o algoritmo para as seguintes tabelas:

- Taxa de juro alta:

_

Tipo de Fundo	Disp. Monet.	Tensão	Valor
Circ. Interno	Alta	Média	Médio
Mina	Alta	Média	Alto
Construção	Alta	Média	Baixo

- Taxa de juro média:

_

Tipo de Fundo	Disp. Monet.	Tensão	Valor
Circ. Interno	Baixa	Alta	Baixo
Mina	Baixa	Alta	Médio
Construção	Baixa	Alta	Baixo

- Taxa de juro baixa:

_

Tipo de Fundo	Taxa Juro	Disp. Monet.	Tensão	Valor
Circ. Interno	Baixa	Alta	Média	Alto
Mina	Baixa	Alta	Média	Médio
Construção	Baixa	Alta	Média	Alto

Em todos os casos, o valor da entropia do tipo de fundo vai ser 0 (os cálculos foram omitidos por simplificação, mas isto poderia ser visto "a olho" pelo facto de, em cada tabela, sabendo o tipo de fundo é possível saber o valor).
Assim sendo, a nossa árvore de decisão seria a seguinte:

$$
Taxa\ de\ Juro \begin{cases} Alta & Tipo\ de\ Fundo \begin{cases} C.I. & Valor = Médio \\ Mina & Valor = Alto \\ Const. & Valor = Baixo \end{cases} \\ Média & Tipo\ de\ Fundo \begin{cases} C.I. & Valor = Baixo \\ Mina & Valor = Médio \\ Const. & Valor = Baixo \end{cases} \\ Baixa & Tipo\ de\ Fundo \begin{cases} C.I & Valor = Alto \\ Mina & Valor = Médio \\ Construção & Valor = Baixo \end{cases} \end{cases}
$$

6.3.2 C4.5

O algoritmo C4.5 é uma evolução do ID3, que, para além das funcionalidades do ID3, também nos permite trabalhar com valores contínuos e desconhecidos, assim como permite podar as árvores de decisão, visto que o ID3 por vezes pode retornar árvores enormes. Para além disso, o ID3 apenas nos permite chegar a uma conclusão final, quando por vezes pode ser desejado retirar conclusões intermédias.

Para já, vamos considerar que já temos um algoritmo que nos permite pegar num conjunto de valores contínuos e, tendo em conta o seu domínio e frequência, classificá-los de forma discreta (pouco, médio, muito...). Mais à frente este algoritmo será explicado.

Assim, o funcionamento do C4.5 passa por, em vez de usarmos o nosso conjunto de treino C, passamos a usar um sub-conjunto $SC \subseteq C$.

Para além disso, em vez de ser usada a entropia, é usada a razão de ganho de informação de um atributo A, sendo que precisamos das seguintes fórmulas:

- Razão de ganho de uma conclusão C sabendo um atributo A: $RG(C|A) = \frac{G(C|A)}{InfoSeparação(A)}$
- Ganho de uma conclusão C sabendo um atributo A: $G(C|A) = fc(A) \times (info(C) - info(C|A))$
- Informação:$info(C) = -\sum_i^N p(c_i) \times log_2(p(c_i))$
- Informação dado um atributo: $info(C|A) = \sum_{j=1}^{M} p(a_j) \times \left[-\sum_{i=1}^{N} p(c_i|a_j) \times log_2\left(p(c_i|a_j)\right) \right]$(semelhante à entropia)
- Informação de Separação:$InfoSeparação(A) = -\sum_{i=0}^{N} p(a_i) \times log_2(p(a_i))$
- Frequência de Atributos Conhecidos:$fc(A) = \frac{conhecidos(A)}{conhecidos(A)+desconhecidos(A)}$
- Nota: Os valores das variáveis são análogos aos vistos no algoritmo ID3.

O algoritmo, muito por alto, funciona da seguinte forma:

1. Gera-se uma árvore de decisão sob o nosso conjunto SC (de forma semelhante ao ID3, mas dividindo no atributo com maior razão de ganho e não no com menor entropia), obtendo uma árvore A;
2. Testa-se o conjunto C na árvore A:
   (a) Para cada teste que falhe, esse teste é adicionado a SC
3. Se SC foi alterado, volta-se a 1;
4. Poda-se a árvore A, transformando sub-árvores complexas e com pouco benefício em folhas.

Depois de repetir este algoritmo várias vezes com sub-conjuntos iniciais SC diferentes, escolhe-se o resultado mais promissor.

A poda da árvore pode ser feita de duas maneiras:

- Algoritmo C4.5 original:
  - Para cada árvore A, todas as sub-árvores S de A que possuam razão inferior ao aumento de erro para a complexidade e razão inferior à média das outras sub-árvores de A são podadas (tornadas em folhas);
- Poda pessimista:
  - Estando uma folha de uma árvore A a classificar N elementos de um conjunto e seja E o número de erros de classificação (por exemplo, se a folha representa "alto", mas está a classificar {"alto","alto","baixo"}, E=1), a confiança nessa folha é dada por $\frac{E+1}{N+2}$.
  - Assim, podam-se todas as sub-árvores cuja poda não aumente significativamente o erro previsível da árvore.

6.3.2.1 Tratamento de Valores Contínuos

O C4.5, para poder trabalhar com valores de domínio continuo, necessita de os agrupar.

Tendo um atributo A com valores possíveis $\{v_1, v_2, ..., v_m\}$, existem m-1 formas possíveis de dividir estes valores em dois grupos, divididos nos pontos médios ($\frac{v_i+v_{(i+1)}}{2}$, se os valores estiverem ordenados).

Sendo assim, a divisão é feita da seguinte forma:

1. Ordenam-se os valores possíveis do atributo por ordem crescente;
2. Testam-se todos os pontos médios e calcula-se a razão de ganho (ou simplesmente o ganho, para simplificação dos cálculos) do atributo caso se dê essa separação;
3. Escolhe-se o ponto médio com melhor razão de ganho e efectua-se aí a separação.

Capítulo 7

Redes Neuronais

> "Redes Neuronais são redes massivamente paralelas, constituídas por elementos simples interligados (usualmente adaptativos), interagindo com o mundo real, tentando simular o que o sistema nervoso biológico faz." - Kohonen, 1987

No conexionismo (onde se encaixam as redes neuronais) ao contrário de na Inteligência Artificial clássica, o conhecimento deixa de ser representado sob a forma de expressões declarativas, sendo estas substituidas pela estrutura e activações de estados numa rede. Passamos agora a ter uma enorme capacidade de paralelismo e processamento distribuido, com a capacidade de transformar um processador num conjunto (ou até, em apenas um) de neurónios e interligar vários processadores entre si.

As redes neuronais computacionais são uma forma de caracterizar o conexionismo, que implicam:

- Aprendizagem: Os sistemas não são "programados", apenas é definida a rede e ela aprende os conceitos de forma autónoma (eventualmente com reforço externo).
- Capacidade Adaptativa: O sistema tem de tolerar a passagem de atributos com valores pouco precisos.

As redes neuronais computacionais tentam simular o sistema nervoso biológico, sendo que temos agora um grafo dirigido em que cada nó é um neurónio, as arestas são axónios e a cada aresta é associado um peso sináptico.

Nesta rede, cada elemento tenta ou manter ou modificar o seu estado, de acordo com os elementos a que está ligado.

Este tipo de redes é caracterizada por 3 elementos:

1. O elemento de processamento (unidade básica ou neurónio artificial);
2. A estrutura das ligações (topologia da rede);
3. A lei de aprendizagem;

7.1 Elemento de Processamento

O neurónio da nossa rede é caracterizado por cinco partes fundamentais:

1. Entradas (de onde vem a informação);
2. Pesos das conexões com outros neurónios (influência de cada entrada);
3. Função de combinação de entradas:
   (a) Normalmente, é a soma ponderada das entradas tendo em conta o seu peso: $e_i = \sum_{j=1}^{N} w_{ji} s_j$, sendo:
      i. e_i - entrada do neurónio i;
      ii. N - número de neurónios que se ligam a i;
      iii. w_{ji} - peso da ligação que sai do neurónio j e se liga ao neurónio i;
      iv. s_j - saída do neurónio j;
   (b) Outras funções usadas são o máximo e o mínimo das entradas ponderadas
4. Função de transferência (função que permite calcular o estado de um neurónio):
   (a) Normalmente, é a função sigmóide: $s_i = \frac{1}{1+e^{-ce_i}}$ (ver figura 7.1).
      i. Sendo c uma constante, sendo que, para $c \to \infty$, a função converge para uma função de heaviside (função de interruptor). Por conveniência, vamos usar c=1, tornando apenas $s_i = \frac{1}{1+e^{-e_i}}$.
      ii. Esta é uma função agradável exactamente por se comportar como uma função de heaviside (dominio bem definido entre 0 e 1 e simetria) tendo, no entanto, um crescimento muito mais suave.
      iii. A derivada desta função em respeito à entrada é dada por: $\frac{d}{de_i} s_i = s_i(1 - s_i)$ (isto vai ser necessário mais à frente).
5. Saída (resultado da função de transferência).

7.2 Estrutura das Ligações

Uma rede neuronal pode ser de 3 tipos:

1. Totalmente conectada (todos os nós estão ligados entre si)
2. Camada única (existe uma camada de neurónios E que recebem a entrada, e uma camada de neurónios S que enviam a saida, estando os neurónios da camada E apenas ligados aos neurónios da camada S)
3. Múltiplas camadas:

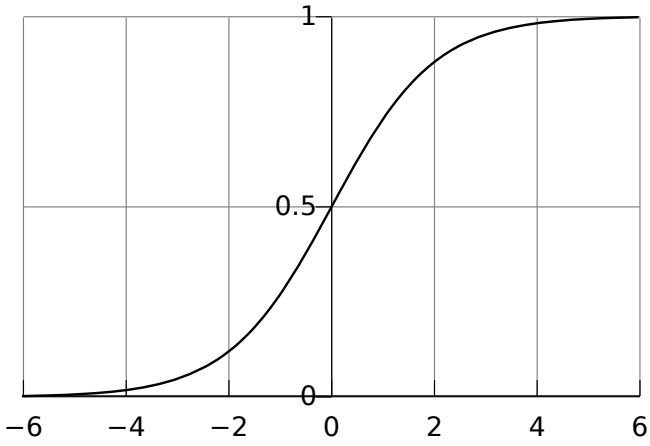


Figura 7.1: Função sigmóide com c=1 (função logistica)

(a) Possuem 3 tipos de ligações:

i. Directa - Ligação a 1 neurónio da camada seguinte;

ii. Inter-directa - Ligação a vários neurónios da camada seguinte;

iii. Intra-directa - Ligação a neurónios da mesma camada

(b) Se a rede apena possuir ligações à camada seguinte (directas e inter-directas), trata-se de uma rede de do tipo "feedforward"

7.3 Lei de Aprendizagem

Sendo as redes neuronais computacionais inspiradas nos trabalhos de neurofisiologia, a aprendizagem é feita de maneira análoga à aprendizagem humana, ou seja, por alteração dos contactos sinápticos (no nosso caso, alteração do peso das ligações entre neurónios).

Esta aprendizagem pode ser tanto por reforço, supervisionada ou não supervisionada.

7.3.1 Aprendizagem por Reforço

Em vez de se fornecerem exemplos entrada->saida correcta, são atribuidos "prémios" e "castigos" de acordo com o facto de a saida ser correcta ou não.

Neste método, as alterações dos pesos são apenas baseadas nos níveis de actividade entre as unidades conectadas. Sendo esta uma informação local, ao alterar um peso não é conhecido o desempenho global do sistema.

A cada iteração da aprendizagem, $W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij}$.

A aprendizagem termina quando a actividade for suficientemente baixa ou mesmo nula.

7.3.1.1 Aprendizagem de Hebb

O estado de um neurónio pode ter dois valores: ou está activo, ou não está activo (1 ou 0), sendo que a eficácia de uma ligação está relacionada com o correlacionamento do valor dos neurónios, ou seja:

$\Delta W_{ij} = s_i \times s_j$

O que faz com que o peso entre dois neurónios só aumente se eles estiverem activos ao mesmo tempo (caso contrário, o peso mantém-se).

7.3.1.2 Aprendizagem de Hopfield

Este método é semelhante ao anterior, mas baseia-se numa fórmula mais agressiva:

$\Delta W_{ij} = (2 \times s_i - 1) \times (2 \times s_j - 1)$

Ou seja, isto faz com que, caso as saidas sejam iguais, o peso aumenta, caso sejam diferentes, o peso diminui (de notar que agora o peso já não se mantém).

Uma propriedade das redes de Hopfield é que todas as ligações são bidireccionais, o que faz com que não sejam redes "feedforward".

7.3.2 Aprendizagem Supervisionada

Neste tipo de aprendizagem, a rede produz uma resposta para uma determinada entrada, sendo que um supervisor (humano ou automático) apresenta a resposta correcta. Se ambas forem idênticas, os pesos mantêm-se iguais, caso contrário, volta-se a aplicar a fórmula $W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij}$.

7.3.2.1 Perceptron

Este é um método apenas usado em redes neuronais de camada única.

Os pesos passam a ser actualizados seguindo:

$\Delta W_{ij} = K \times (s_j - d_j) \times s_i$

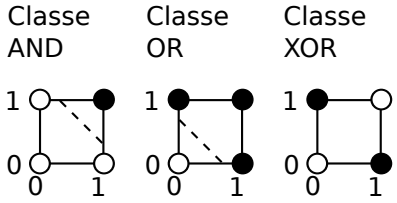
Onde K é uma constante e d_j representa o valor correcto para uma saída j.

Esta regra de aprendizagem converge garantidamente para a representação correcta do problema se e só se a nossa classe de modelos for separável linearmente (ou seja, se existir uma função linear capaz de separar os valores 0 dos valores 1), como pode ser visto na figura 7.2.

7.3.2.2 Retro-propagação do Gradiente (Backpropagation)

Ao contrário do método "perceptron", este novo método já pode ser utilizado em redes neuronais de múltiplas camadas, tentando minimizar a função de custo quadrático: $\frac{1}{2 \times M} \sum_{k=1}^{M} \sum_{i=1}^{N} (s_{ki} - d_{ki})^2$, onde:

- M - número de exemplos apresentados.
- N - número de neurónios na camada de saída.



Como pode ser visto na figura, as classes AND e OR são linearmente separáveis, enquanto que as classes XOR não o são.

Figura 7.2: Exemplo de classes separáveis e não separáveis linearmente

Vejamos agora a implementação deste algoritmo (Supondo uma rede com uma camada de entrada com A únidades, uma camada intermédia com B unidades, e uma camada de saída com C unidades):

1. Apresenta-se um exemplo E, constituido por valores e_i (um valor para cada entrada, logo i=1...A);
2. Propagam-se os estados da camada de entrada para a camada de interméida (usando a função sigmoide):
   (a) $m_j = \frac{1}{1+e^{-\sum_i W_{ij} \times e_i}}$ (sendo m_j as saída do nó j da camada de entrada, ou seja, as entradas da camada intermédia)
3. Propagam-se agora os estados da camada intermédia para a camada de saída (mesma fórmula, com variáveis alteradas):
   (a) $s_k = \frac{1}{1+e^{-\sum_j W_{jk} \times m_j}}$
4. Por fim, calcula-se o erro nas unidades de saída, tendo em conta a reposta certa:
   (a) Sendo a derivada parcial da nossa função de transferência (sigmoide) em função de uma entrada dada por $s_k \times (1 - s_k)$.
   (b) Usando a derivada como um peso para ponderar o erro (de forma a haver alterações mais pequenas quando s_k se apróxima de 1 ou 0), consideramos o erro da saída como: $\delta s_k = s_k \times (1 - s_k)(d_k - s_k)$.
5. Acabando a propagação, retropropaga-se o erro para as únidades intermédias:
   (a) $\delta m_j = m_j \times (1 - m_j) \sum_k W_{jk} \times \delta s_k$
6. Alteram-se os pesos entre a camada intermédia e a camada de saída:
   (a) $W_{jk}(t+1) = W_{jk}(t) + \mu(t) \times \Delta W_{jk}(t)$
   (b) $\Delta W_{jk}(t) = \delta s_k \times m_j$

(c) $\mu(t) = K \times \mu(t-1)$

i. Sendo K<1 (por exemplo, 0.7) se $\Delta W_{jk}(t)$ e $\Delta W_{jk}(t-1)$ tiverem sinais oposotos;

ii. Sendo K>1 (por exemplo, 1.2) se $\Delta W_{jk}(t)$ e $\Delta W_{jk}(t-1)$ tiverem o mesmo sinal;

iii. Quanto mais afastado de 1 for K, mais rápida é a aprendizagem, quanto mais próximo de 1, mais precisa e menos predisposta a oscilações é a aprendizagem;

A. Uma solução é usar $\Delta W_{jk}(t) = \delta s_k \times m_j + \beta \times \Delta W_{jk}(t-1)$, sendo $0 < \beta < 1$. Assim evitam-se oscilações acentuadas.

7. Alteram-se os pesos entre a camada intermédia e a camada de entrada (semelhante ao passo 6 mas com variáveis diferentes):

(a) $W_{ij}(t+1) = W_{ij}(t) + \mu(t) \times \Delta W_{ij}(t)$

(b) $\Delta W_{ij}(t) = \delta m_j \times e_i$

(c) $\mu(t) = K \times \mu(t-1)$

8. O algoritmo retorna 1 caso sejam necessários mais exemplos ou se se pretender diminuir o erro.

Este algoritmo já é capaz de tratar exemplos que não são linearmente separáveis.

7.3.3 Aprendizagem não Supervisionada

Ao contrário da aprendizagem supervisionada, a aprendizagem não supervisionada, como o nome indica, não necessita de um supervisor. Estes métodos são úteis para quando não dispomos informação à priori sobre possíveis classificadores.

7.3.3.1 Aprendizagem Competitiva

A aprendizagem competitiva baseia-se no seguinte conceito: Sendo apresentado um exemplo a uma rede, todas as unidades devem concorrer pelo direito à resposta. A célula mais activa será aquela que responder mais fortemente, ajustando-se os pesos de forma a que a sua resposta seja reforçada (facilitando a identificação de entradas semelhantes).

Este tipo de aprendizagem faz com que unidades distintas passem a representar caracteristicas distintas, permitindo classificar configurações de entrada.

7.3.3.2 Método de Kohonen

Mais uma vez, este é um método que só funciona em redes de camada única.

No fim da aprendizagem, cada unidade deve responder selectivamente a uma classe, ou seja, para cada configuração, uma unidade encontra-se mais activa que as outras, permitindo-nos classificar a entrada. Este método é idêntico ao

anterior, mas quando uma unidade vence a competição, não só são alterados os seus pesos, como também são alterados os pesos dos vizinhos.

Calculando para cada unidade de saída: $c_j = \sum_{i=0}^{n}(s_i - W_{ij})^2$, a unidade mais activa será aquela que possuir o menor valor de c_j.