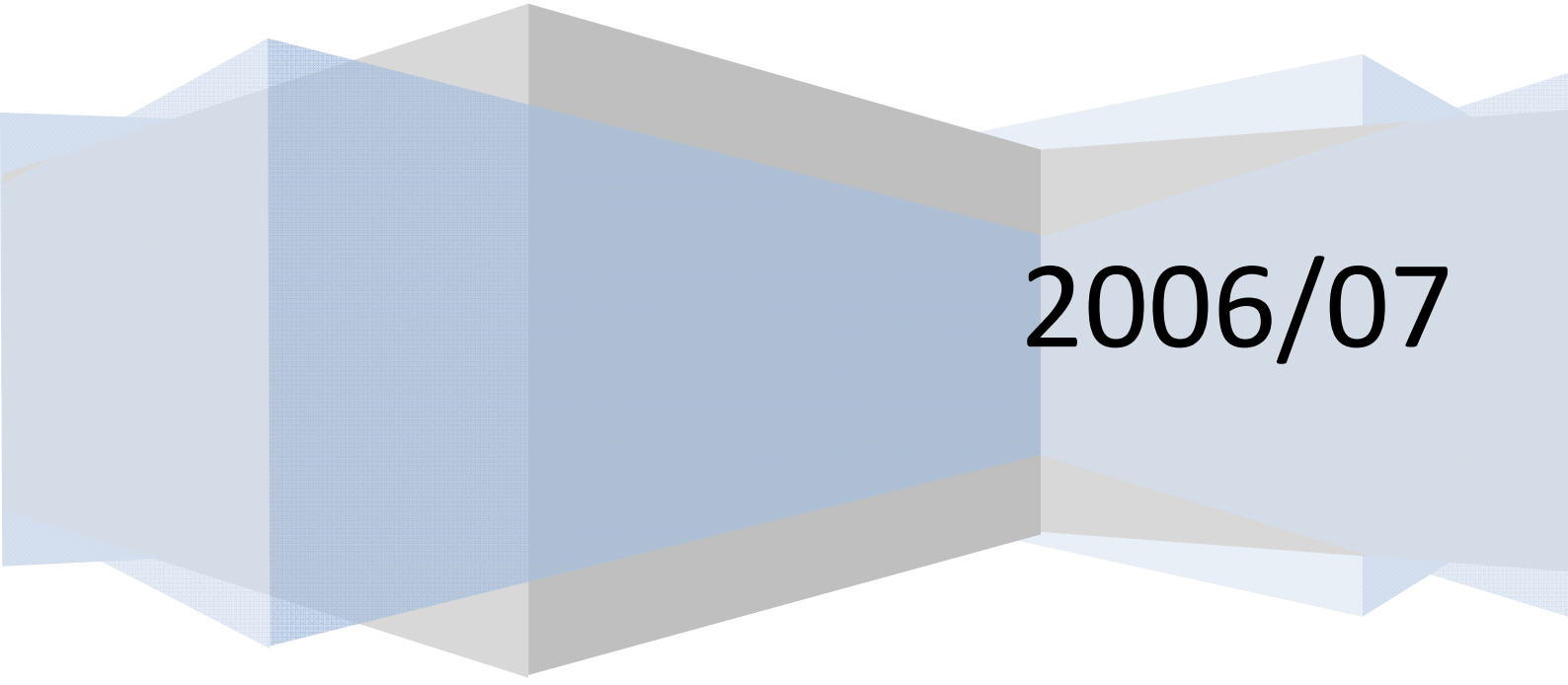
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Introdução à Inteligência Artificial

Apontamentos Teóricos



2006/07

Índice

I - Agentes Inteligentes 8
II - Procura 11
Algoritmos de procura em árvores 12
Estratégias de procura cegas 14
Procura em largura primeiro *(breadth-first)* 14
Procura de custo uniforme 15
Procura em profundidade primeiro 15
Procura de profundidade limitada 15
Procura por aprofundamento progressivo 16
Procura em grafos 17
III - Procura Heurística 19
Procura pelo melhor primeiro: 19
Procura Sôfrega 19
Procura A* 20
Heurísticas admissíveis 20
Heurísticas consistentes 21
IV - Procura Local 25
Algoritmos de procura local 25
Classes de Complexidade 26
Trepa Colinas (Hill Climbing) 27
Procura Local Estocástica 28
V - Algoritmos Genéticos 31
Operação de Recombinação 32
Outros operadores de selecção 34
O Caixeiro-viajante 34
Operador de recombinação PMX 35
Operador de recombinação OX 35
Outras variantes do OX (OX3) 36
Operador de recombinação CX 36
Operadores de mutação 37
Algoritmos meméticos 37
Programação Genética 38
VI - Problemas de Satisfação de Restrições 39

CSPs 39
Algoritmo de consistência de Arcos 41
CSPs com estrutura arbórea 41
CSPs de estrutura quasi-arbórea 42
Algoritmos locais para CSPs 42
Algoritmo min-conflitos 43
VII - Agentes Lógicos 44
Bases de conhecimento(s) 44
Um agente simples baseado em conhecimento 44
Noções de Lógica 45
Conclusão Lógica 45
Modelos 46
Inferência 46
Lógica Proposicional: Sintaxe 47
Lógica Proposicional: Semântica 47
Tabela de verdade para os conectivos 48
Posições no mundo do Wumpus 48
Inferência através de tabelas de verdade 49
Inferência por enumeração 49
Equivalência Lógica 50
Validade e Satisfatibilidade 50
Geografia das expressões Booleanas 51
Métodos de Prova 51
Encadeamento para a frente e para trás 52
Encadeamento para a frente 52
Algoritmo de Encadeamento para a frente 53
Demonstração de Completude 53
Encadeamento para trás 53
Encadeamento para a frente vs. Para trás 54
Resolução 54
Conversão para FNC 55
Algoritmo da Resolução 55
Um agente lógico no mundo do Wumpus 55
Limites de expressividade da lógica proposicional 56

Resumo das classes de complexidade (P) 56
Resumo das classes de complex. (NP e coNP) 57
Complexidade e Lógica proposicional 57
Verificação eficiente de satisfatibilidade 58
Algoritmo de Davis-Putnam 58
Propriedades do algoritmo de Davis-Putnam 59
WalkSAT 59
Propriedades do WalkSAT 60
Problemas de satisfatibilidade difíceis 60
VIII - Lógicas não monótonas 61
Cálculo de Extenções 61
Lógica Autopistémica 62
Programação em Lógica 62
Datalog (Database Logic) 62
Semântica de Programas em Lógica Definidos 63
Programas em Lógica Normais 63
Complexidade datalog 63
IX - Lógica de Primeira Ordem 64
Semântica da Lógica de Primeira Ordem 68
Consequência Lógica 69
Grupos Abelianos 69
Quantificação Universal 70
Quantificação Existencial 71
X - Inferência em Lógica de Primeira Ordem 72
Instanciação Universal 72
Instanciação Existencial 72
Redução à Inferência Proposicional 73
Redução 73
Unificação 73
Algoritmo de Encadeamento para a Frente 74
Algoritmo de Encadeamento para a Frente 75
Sistemas Prolog 76
XI - Planeamento 77
Procura vs. Planeamento 77

Planeamento Clássico 77
Cálculo de Situações 77
Tarefas que Pretendemos Efectuar 79
Descrevendo acções no cálculo de situações 79
Interrogando a Teoria 79
Problemas a enfrentar 80
Resolução do problema da quiescência 80
Axiomas de estado sucessor para o Wumpus 80
Problema da ramificação para o Wumpus 81
Axiomas de nomes únicos 81
Axiomas de acções únicas 81
Linguagem STRIPS 82
Operadores STRIPS 82
Tradução para Cálculo de Situações 82
Princípios da Linguagem STRIPS 83
Planeamento em espaços de estados 84
Heurísticas para planeamento em espaços de estados 85
Exemplo 85
Complexidade do planeamento 86
Planos parcialmente ordenados 86
Processo de Planeamento 87
Algoritmo POP 87
Ameaças e promoção/despromoção 88
Propriedades do POP 88
Ameaças e variáveis 88
Exemplo: Mundo dos blocos 89
Casa & Descasa 91
XII - Técnicas de Grafos 93
Grafo de Planeamento 93
Exclusão Mútua 95
Exemplo(continuação) 95
Extracção da Solução 96
Comparação com POP 99
Heurísticas para o POP 99

XIII - Aprender a partir de exemplos 100
Aprendizagem Indutiva 101
Aprendizagem Conceptual 101
Tratar Falsos Positivos e Negativos 102
Espaço de Hipóteses 102
Satisfação de Hipóteses 102
Ordenação parcial entre hipóteses 102
Espaço de hipóteses 103
Eliminação de Candidatos 103
Espaço de Versões 103
Evolução das Fronteiras 104
Algoritmo de eliminação de candidatos (Mitchell) 104
Aprender árvores de decisão 105
Utilizações típicas de Árvores de decisão 105
Indução de Árvores de Decisão 105
Algoritmo DTL (ou ID3) 106
Medida de desempenho 106
Aspectos Práticos de Utilização 107
Avaliar a Performance 107
Técnicas de Avaliação 107
XIV - Incerteza 108
Independência Condicional 108
Regra de Bayes 109
XV - Redes Bayesianas 110
XVI - Inferência em redes Bayesianas 111
Tarefas de inferência 111
Algoritmo de enumeração 111
Árvore de avaliação 112
Inferência por eliminação de variáveis 112
Algoritmo da eliminação de variáveis 113
Inferência por simulação estocástica 113
Amostragem por rejeição 114
Markov Chain Monte Carlo (MCMC) 115
XVII - Procura com adversários 116

Jogos Deterministas 116
Jogos não deterministas 120
XVIII - Redes Neuronais 122
Unidade de McCulloch-Pitts 122
Funções de activação 122
Implementação das funções lógicas 123
Topologia de redes 123
Exemplo de rede alimentada para a frente 123
Expressividade dos perceptrões 124
Regra Delta - Widrow e Hoff 124
Algoritmo de Aprendizagem do Perceptrão 125
Exemplo de aplicação 125
Perceptrão (regra delta generalizada) 126
Aprendizagem do perceptrão (cont.) 126
Perceptrões multicamada 127
Expressividade de redes multicamada 127
Aprendizagem por retropropagação 127
Derivação da regra de retropropagação 128
Derivação retropropagação outras camadas 128
O algoritmo de retropropagação 129
Esquema de funcionamento 129
Exemplo de aplicação (propagação) 129
Exemplo de aplicação (retropropagação) 130
Aprendizagem com Retropropagação (cont) 130
Utilização prática do Algoritmo 131
Sobreajustamento 131
Sobreajustamento (quando parar?) 132
Problemas de convergência 132

I - Agentes Inteligentes

Um agente é qualquer entidade que percepciona o seu ambiente através de sensores e age nele através de actuadores.

Agente biológico:

Sensores: olhos, ouvidos, pele, etc.
Actuadores: mãos, pernas, boca, etc.

Agente robótico:

Sensores: câmaras e sensores de proximidade
Actuadores: diversos motores

Agente computacional (software, vida artificial)

Sensores: teclado, rato, ligações TCP
Actuadores: ecrã, placa de controlo, drive disco, enviar email, etc.

Características dos agentes:

Corpo
Localização: Existem no espaço e no tempo
Capacidades: Sensores e actuadores
Decisão: Deliberativa ou não

Propriedades dos agentes:

Racionalidade: Maximizar o seu desempenho em função da informação disponível.
Reactividade: Responder em tempo útil a mudanças do ambiente**.**
Orientado pelos objectivos: Tem iniciativa, não se limita a reagir ao ambiente.
Comunicação: Comunica com outros agentes.
Autonomia: Aprende e adapta-se de acordo com a experiência anterior.
Mobilidade: Capaz de se transportar de um local para outro (e.g. de máquina para máquina).
Carácter: Tem estado emocional e sua própria personalidade (veracidade e benevolência).

Agentes racionais:

Um agente deve “fazer aquilo que é correcto", baseando-se no que percepciona e nas acções que pode efectuar. A acção correcta é aquela que trará mais sucesso ao agente.

Medida de desempenho: Critério objectivo que mede o sucesso do comportamento de um agente.

Um agente racional deve seleccionar a acção que maximiza o valor esperado da medida de desempenho, dada a evidência fornecida pela sequência de percepções e conhecimentos que o agente tenha.

Racionalidade ≠ omnisciência (tudo saber)
Racionalidade ≠ clarividência (tudo prever)
Racionalidade ≠ bem sucedido

Racionalidade pressupõe obtenção de informação e exploração, aprendizagem e autonomia.

PEAS: Performance measure, **E**nvironment, **A**ctuators, **S**ensors

Deve-se especificar completamente a situação no desenho de um agente inteligente.

Tipos de Ambientes:

Totalmente observável (vs. parcialmente observável): Os sensores do agente dão acesso ao estado completo do ambiente em cada instante de tempo.

Determinista (vs. estocástico): O estado seguinte do ambiente é completamente determinado pelo estado corrente e pela acção executada pelo agente. Se o ambiente é determinista exceptuando-se as acções de outros agentes, então o ambiente é estratégico.

Episódico (vs. sequencial): A experiência do agente é dividida em episódios atómicos (percepção seguida de única acção), em que a escolha da acção depende apenas do próprio episódio.

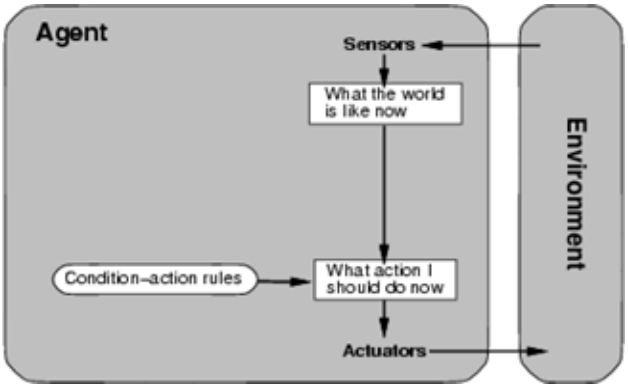
Estático (vs. dinâmico): O ambiente não se altera enquanto o agente delibera. O ambiente é semi-dinâmico quando o ambiente não se altera com a passagem do tempo, ao contrário da medida de desempenho.

Discreto (vs. contínuo): Um número limitado de acções e percepções, e claramente definidos.

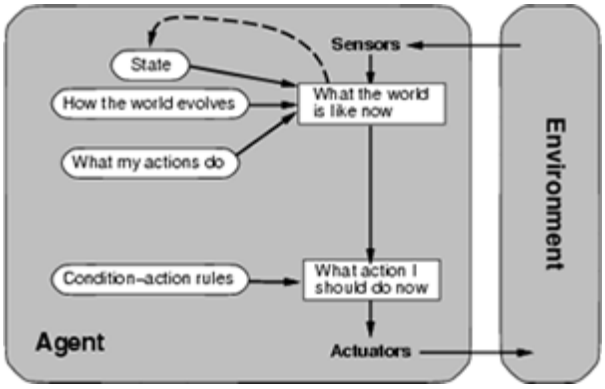
Agente único (vs. multi-agente): Agente a operar sozinho no ambiente.

Tipos de Agentes: Quatro tipos de agentes de generalidade crescente:

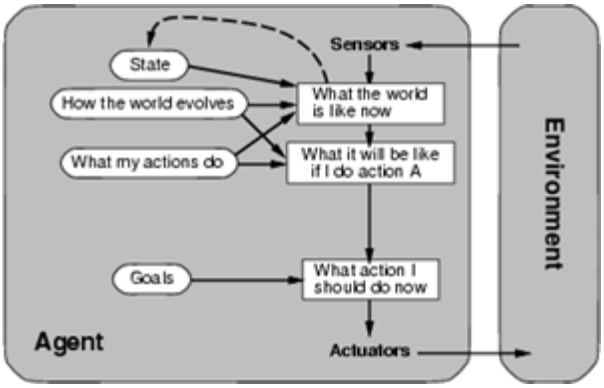
Agente reactivo puro:



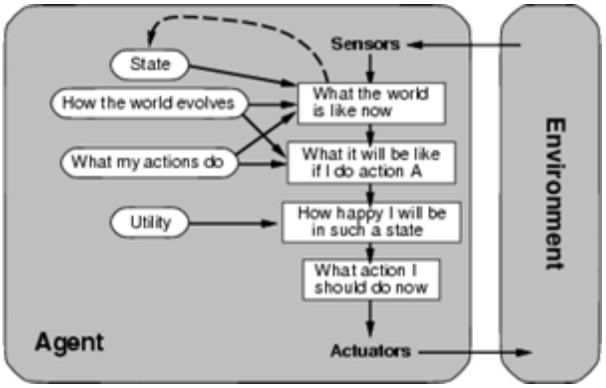
Agente reactivo com memória:



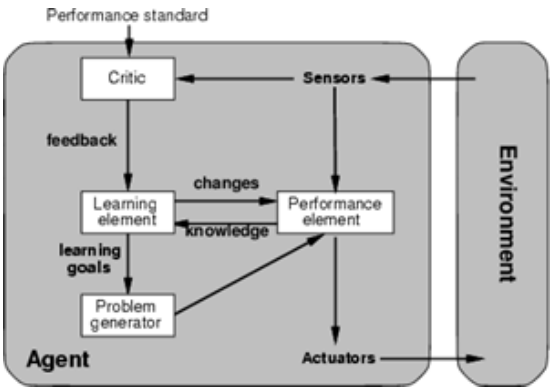
Agente deliberativo com objectivo:



Agente deliberativo com função de utilidade:



Agente aprendiz:



II - Procura

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Tipos de Problemas:

Determinista, observável: Problema com estado único. Agente sabe exactamente em que estado se encontrará. A solução é uma sequência de acções.

Não observável: Problema de conformidade. Agente pode não saber onde está. Caso exista, a solução é sequência de acções.

Não determinista e/ou parcialmente observável: problema de contingência. Percepção fornece nova informação acerca do estado corrente. A solução é uma árvore ou plano de acção. Habitualmente intercala procura com execução

Espaço de estados desconhecido: problema de exploração ("online").

Formulação de problema de estado único: Um problema de procura é definido por 4 constituintes:

1. Estado inicial;
2. Acções, operadores ou função sucessor;
3. Teste objectivo pode ser explícito ou implícito;
4. Custo do caminho (aditivo);

Uma **solução** é uma sequência de acções que partindo do estado inicial permite atingir o estado objectivo.

Selecção de um espaço de estados:

A realidade é absurdamente complexa; o espaço de estados deve ser abstraído para a resolução de problemas.

Estado (abstracto) = conjunto de estados reais

Acção (abstracta) = combinação complexa de acções reais

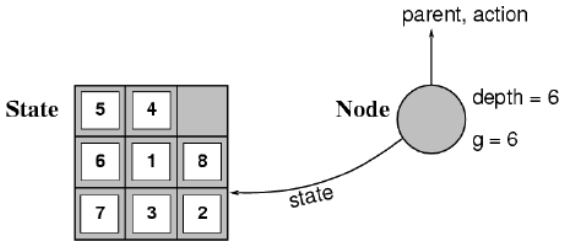
Para ser concretizável, qualquer estado real deve permitir chegar a algum estado real

Solução (abstracta) = Conjunto de caminhos reais que são soluções na realidade. Cada acção abstracta deverá ser mais simples do que no problema original!

Algoritmos de procura em árvores

Ideia básica: Simulação *offline* da exploração do espaço de estados através da geração de sucessores de estados já explorados.

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```



Implementação: estados vs nós.

Um **estado** é uma representação de uma configuração física. Estados não têm pais, profundidade ou custo do caminho!

Um **nó** é uma estrutura de dados constituinte da árvore de procura incluindo o estado, pai, nó, acção, profundidade e custo de caminho acumulado *g(x)*.

A função *Expand* cria novos nós, preenchendo os vários campos e recorrendo à função *SuccessorFn* do problema para criar os estados correspondentes.

Implementação: Procura genérica em árvores

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Estratégias de procura: Uma estratégia de procura é definida pela ordem de expansão dos nós. As estratégias são avaliadas segundo as dimensões:

Completude: encontra garantidamente uma solução, caso exista?
Complexidade temporal: número de nós gerados
Complexidade espacial: número máximo de nós em memória
Optimalidade: encontra sempre uma solução de custo mínimo?

A complexidade temporal e espacial são avaliadas em função de:

b: factor de ramificação máximo da árvore de procura
d: profundidade da solução de custo mínimo
***m*:** profundidade máxima do espaço de estados (pode ser ∞)

Estratégias de procura cegas

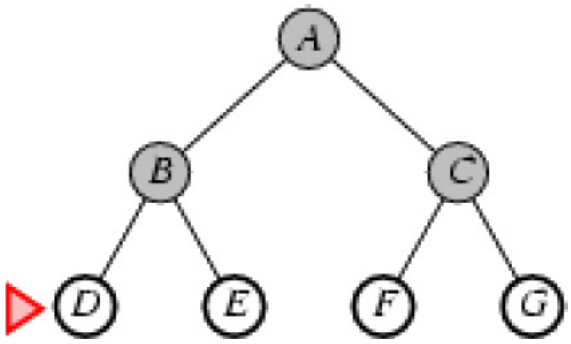
As estratégias de procura cegas (ou não informadas) recorrem apenas à informação disponibilizada no problema.

- ✓ **Procura em largura primeiro** *(breadth-first)*
- ✓ **Procura de custo uniforme** *(uniform- cost)*
- ✓ **Procura em profundidade primeiro** *(depth-first)*
- ✓ **Procura em profundidade limitada**
- ✓ **Procura por aprofundamento progressivo**
- ✓ **Procura bidireccional**

Procura em largura primeiro *(breadth-first)*

Expandir um nó de menor profundidade.

Implementação: *fringe* é uma fila FIFO; novos sucessores vão para o fim.



Completa? Sim (se *b* é finito);
Tempo? *1 + b + b2 + b3* + … + *bd* + *b(bd-1)* = O(bd+1);
Espaço? *O(bd+1)*(mantém todos os nós);
Optimal? Sim (se custo = 1 por passo).

Espaço é o maior problema (mais do que o tempo).

Procura de custo uniforme

Expandir o nó por tratar de menor custo.

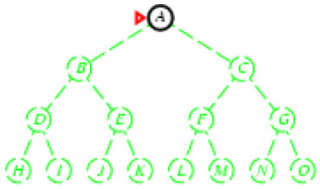
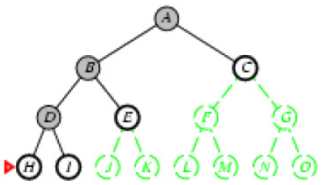
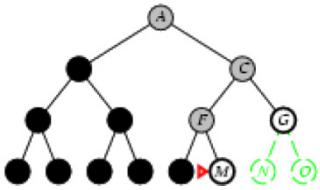
Implementação: *fringe* = fila ordenada pelo custo do caminho. Equivale à procura em largura se custos forem constantes.

Completa? Sim, se custo do passo ≥ ε;
Tempo? Número de nós com $g \leq$ custo da solução óptima, $O(b^{ceiling(C^*/\varepsilon)})$ em que C^* é o custo da solução óptima;
Espaço? Número de nós com $g \leq$ custo da solução óptima, $O(b^{ceiling(C^*/\varepsilon)})$;
Óptima? Sim; nós expandidos por ordem crescente de *g(n)*.

Procura em profundidade primeiro

Expandir um dos nós mais profundos

Implementação: *fringe* = pilha LIFO, i.e., colocar sucessores à frente.

Completa? Não: falha em espaços de profundidade infinita, espaços com ciclos. Modificação para evitar espaços repetidos no mesmo caminho (completa para espaços finitos).
Tempo? *O(bm)*: terrível se ***m*** muito maior do que ***d***, mas se as soluções são densas, pode ser muito mais eficiente do que a *procura em largura primeiro*.
Espaço? *O(bm),* i.e., espaço linear!
Óptima? Não.

Procura de profundidade limitada

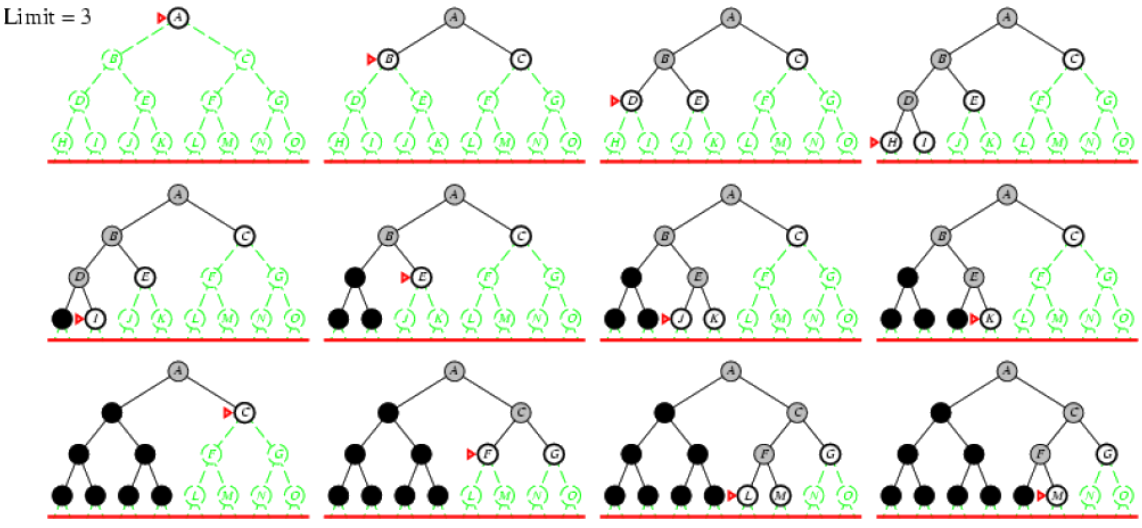
Procura em largura primeiro, com limite de profundidade ***l***, i.e., nós à profundidade *l* não têm sucessores.

Implementação recursiva:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Procura por aprofundamento progressivo

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```



Completa? Sim.
Tempo? *(d+1)b0 + db1 + (d-1)b2 + …+ bd = O(bd)*.
Espaço? *O(bd)*.
Óptima? Sim, se custos constantes.

Sumário dos Algoritmos:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes^{+}	Yes^{+}	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes*	Yes	No	No	Yes*

+ se o factor de ramificação for finito
***** Óptima se o custo total for monótono na profundidade da árvore (e.g. custos constantes e idênticos).

Nota: A não detecção de estados repetidos pode tornar um problema linear num problema exponencial!

Procura em grafos

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Optimalidade de procura em grafos:

O algoritmo de procura em grafos ignora novos caminhos para o mesmo estado, portanto a questão da optimalidade não é imediata. Para grafos com custos de passos constantes, quer a *procura em largura primeiro* quer a *procura de custo uniforme* com lista fechada são óptimas. Se a inserção na lista aberta adicionar apenas nós correspondentes a estados não expandidos e mantiver o nó com o menor custo total, então a procura de custo uniforme com lista fechada é óptima. Muito semelhante ao algoritmo de *Dijkstra* para encontrar o melhor caminho num grafo dirigido.

Complexidade de procura em grafos:

Claramente, no pior caso, a complexidade espacial para qualquer dos algoritmos básicos de procura passa a ser da ordem bd + 1 quando se utiliza a lista fechada.

Recorrendo à lista fechada, a complexidade dos algoritmos de procura é limitada pelo número de estados no espaço de procura e não pelo número de caminhos nesse espaço. Para alguns problemas, pode resultar em diminuições exponenciais em tempo e espaço. Contudo, em espaços de procura muito grandes pode continuar a ser proibitivo.

Implementação dos algoritmos:

Obviamente, deve-se ter algum cuidado na selecção das estruturas de dados para implementaras listas fechadas e abertas. Habitualmente a lista fechada é implementada com uma tabela de dispersão (*hashtable*). Quanto à lista aberta, normalmente opta-se por:

Fila de prioridade (priority queue) quando o grafo de estados é esparso número reduzido de nós sucessores limitados por uma constante pequena. Complexidade temporal $O(N * logN + L * logN)$, em que N o número de estados e L o número de arcos. Esta é a situação habitual.

Quando o grafo é denso, então deve-se utilizar uma lista ou tabela de dispersão. Complexidade temporal da ordem de $O(N2+ L)$.

III - Procura Heurística

- ✓ **Procura pelo melhor primeiro**
- ✓ **Procura sôfrega**
- ✓ **Procura A***

Procura pelo melhor primeiro:

Ideia: Aplicar uma função de avaliação *f(n)* a cada nó. Indica-nos se o nó é promissor ou não. Expandir o nó que aparenta ser mais promissor.

Implementação: Ordenar os nós na lista aberta por ordem decrescente da função de avaliação

Casos especiais: Procura sôfrega e Procura A*.

Procura Sôfrega

Função de avaliação *f(n) = h(n)* (**heurística**) = estimativa do custo do menor caminho para ir de *n* até um estado objectivo. A *Procura Sôfrega* expande o nó que *aparenta* estar mais próximo do objectivo.

Propriedades da Procura Sôfrega:

Completa? Não, pode ficar presa em ciclos. Completa em espaços finitos com verificação de estados repetidos.
Tempo? *O(bm)*, mas uma boa heurística pode ter melhorias espectaculares.
Espaço? *O(bm)*, mantém todos os nós em memória.
Óptima? Não.

Procura A*

Ideia: evitar expandir caminhos que já têm elevado custo.

Função de avaliação: $f(n) = g(n) + h(n)$

- ✓ ***g(n)*** = custo actual para atingir *n*.
- ✓ ***h(n)*** = custo estimado para atingir o objectivo a partir de *n*.
- ✓ ***f(n)*** = custo total estimado do caminho até ao objectivo passando por *n*.

Propriedades do A*:

- ✓ O A* expande todos os nós com ***f(n) < C****
- ✓ O A* expande alguns nós com ***f(n) = C****
- ✓ O A* nunca expande nós com ***f(n) > C****

Completo? Sim (a não ser que haja um número infinito de nóscom f $\leq f(G)$)
Tempo? Exponencial no erro relativo de *h* x o tamanho da solução.
Se |h(n) –h*(n)| ≤ O (logh*(n)) o algoritmo A* tem um comportamento sub-exponencial.
Espaço? Mantém todos os nós em memória
Óptimo? Sim, se a heurística for admissível

O algoritmo A* é optimalmente eficiente para qualquer heurística dada; não há outro algoritmo **óptimo** que garantidamente expanda um menor número de nós!

A* expande nós por ordem crescente de valores da função de avaliação; adiciona gradualmente contornos aos nós (c.f. procura em largura adiciona níveis).

Contorno *i* tem todos os nós $f=f_i$, em que $f_i < f_i + 1$.

Heurísticas admissíveis

Uma heurística *h(n)* é admissível se para todo o nó *n*, $h(n) \leq h^*(n)$, em que $h^*(n)$ é o custo real de atingir o objectivo a partir de *n* . Uma heurística admissível nunca sobrestima o custo de alcançar o objectivo, i.e., é optimista.

Teorema: Se *h(n)* é admissível, então o algoritmo A* usando *TREE-SEARCH* é óptimo.

Heurísticas consistentes

A demonstração de optimalidade do A* não se generaliza para o algoritmo de procura em grafos (eliminação de estados já explorados). Uma heurística é consistente se para todo o nó *e* todo o seu sucessor *n', gerado pela acção a*:

$$h(n) \leq c(n,a,n') + h(n')$$

Se *h* é consistente, temos

$$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$$

Ou seja, *f(n)* é não decrescente ao longo de qualquer caminho (é monótona).

Teorema: Se *h(n)* for consistente, então o A* recorrendo à procura GRAPH-SEARCH é óptimo.

O que fazer com heurísticas inconsistentes?

Solução simples, a lista fechada mantém nós em vez de estados.

Seja *n* um novo nó gerado pelo algoritmo. Se existir um nó na lista fechada *m* para o mesmo estado de *n* tal que *f(n) < f(m)* então retira-se o nó *m* da lista fechada e coloca-se o novo nó *n* na lista aberta.

Estimativa PathMax:

Existe uma optimização que tenta manter a heurística consistente (estimativa PathMax) mas mais complexa.

A ideia consiste em utilizar como valor da função de avaliação

$$f^{\wedge}(m) = \max \{f(n); f(m)\}$$

em que *m* é sucessor de *n*. O valor de $f^{\wedge}(m)$ é obtido em tempo de execução e depende do caminho seguido para atingir *m*.

Poderá ser necessário remover, na mesma, nós da lista fechada.

Problemas Relaxados: Um problema com menos restrições nas acções é designado por problema relaxado. O custo exacto de uma solução óptima para o problema relaxado é uma heurística admissível para o problema original!

Procura informada com memória limitada:

Algoritmo IDA*
Algoritmo recursivo de procura pelo melhor primeiro (RBFS)
Algoritmo A* de memória limitada simplificado

A* por aprofundamento progressivo:

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  local variables: f-limit, the current f- COST limit
                   root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f- COST[root]
  loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution
sequence and a new f- COST limit
  inputs: node, a node
          f-limit, the current f- COST limit
  local variables: next-f, the f- COST limit for the next contour, initially ∞

  if f- COST[node] > f-limit then return null, f- COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

Propriedades do IDA*:

Completo, espaço linear e óptimo. Prático se os custos do passos forem unitários. Dificuldade em lidar com custos reais, podendo acarretar grande tempo de processamento motivado por regenerações sucessivas de nós. A análise de sobrecarga efectuada para as versões cegas não é válida aqui!

RBFS:

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  RBFS(*problem*,MAKE-NODE(INITIAL-STATE[*problem*]),∞)

---

function RBFS(*problem*,*node*, *f-limit*) **returns** a solution, or failure and a new *f*-cost limit
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
  successors ← EXPAND(*node*,*problem*)
  if *successors* is empty **then return** *failure*,∞
  for each *s* **in** *successors* **do**
    f[*s*] ← MAX(g(*s*)+h(*s*), f[*node*])
  repeat
    best ← the lowest *f*-value node in *successors*
    if *f*[*best*] > *f-limit* **then return** failure, *f*[*best*]
    alternative ← the second lowest *f*-value node among *successors*
    result, *f*[*best*] ← RBFS(*problem*, *best*, min(*f-limit*,alternative))
    if *result*≠*failure* **then return** *result*

Propriedades do RBFS:

Completo, espaço linear e Óptimo (se a heurística for admissível). Melhor que o IDA*. Continua a ter problemas com regenerações sucessivas de nós: utiliza pouca memória...

SMA*:

Tal como no A*, expande-se a melhor folha até ficar com a memória cheia. Quando a memória fica toda ocupada, esquece a folha mais antiga com o pior valor, e guarda no pai o seu valor, para possível regeneração. Um nó só é regenerado quando todos os outros caminhos se mostrarem piores do que aqueles do nó esquecido.

```
function SMA*(problem) returns a solution sequence
 inputs: problem, a problem
 local variables: Queue, a queue of nodes ordered by f-cost

 Queue ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
 loop do
     if Queue is empty then return failure
     n ← deepest least-f-cost node in Queue
     if GOAL-TEST(n) then return success
     s ← NEXT-SUCCESSOR(n)
     if s is not a goal and is at maximum depth then f(s) ← ∞
     else f(s) ← MAX(f(n), g(s)+h(s))
     if all of n's successors have been generated then
         update n's f-cost and those of its ancestors if necessary
     if SUCCESSORS(n) all in memory then remove n from Queue
     if memory is full then
         delete shallowest, highest-f-cost node in Queue
         remove it from its parent's successor list
         insert its parent on Queue if necessary
     insert s on Queue
 end
```

Propriedades do SMA*

O SMA* utiliza *toda* a memória disponível para levar a cabo a procura. É completo se existir uma solução alcançável (cujo caminho caiba em memória). É Óptimo se existir uma solução óptima alcançável, caso contrário devolve a melhor solução cujo caminho cabe em memória.

Retira da fronteira nós superficiais com valores elevados da função de avaliação. Um nó retirado da fronteira só é regenerado se todos os irmãos forem piores do que ele. É o melhor algoritmo para procurar soluções óptimas, nomeadamente quando o espaço de estados é um grafo, os custos não são uniformes e a geração de nós é mais dispendiosa do que manter listas de nós abertos e fechados, mas as limitações de memória podem tornar um problema intratável...

IV - Procura Local

Algoritmos de procura local

Em muitos problemas de optimização, o caminho para a solução é irrelevante. O estado objectivo é a própria solução. O espaço de estado é igual ao conjunto de configurações "completas" (soluções candidatas).

Nestas situações, podem-se usar **algoritmos de procura local**: mantém-se um único estado "corrente", e tenta-se alterá-lo para melhorar a sua qualidade.

Espaço constante, apropriado para procura *online* e *offline* em espaços discretos e contínuos, assim como para resolução de problemas de optimização.

Problema das *n rainhas*, satisfatibilidade booleana, caixeiro-viajante, problemas combinatórios (problemas que tipicamente envolvem encontrar grupos, ordenações ou atribuições de um número finito de objectos discretos que satisfazem um conjunto de condições ou restrições, habitualmente, o espaço de soluções candidatas para uma instância particular é pelo menos exponencial no tamanho dessa instância).

Tipos de problemas combinatórios:

Problemas de Decisão (têm resposta sim/não).

Variante de decisão: saber se existe ou não solução para o problema;
Variante de procura: encontrara solução, caso exista.

Problemas de Optimização (encontrar solução que optimiza valor de função objectivo)

Variante de procura: encontrar a solução óptima;
Variante de avaliação: indicar qual o valor mínimo/máximo da função objectivo.

Qualquer problema de optimização origina um problema de decisão associado, a partir de um limite *b* fornecido. Se o problema é de minimização, saber se existe uma solução com valor inferior ou igual ao *b* dado. Se o problema é de maximização, saber se existe uma solução com valor superior ou igual ao *b* dado.

Classes de Complexidade

Habitualmente definidas a partir de problemas de decisão e para o pior caso.

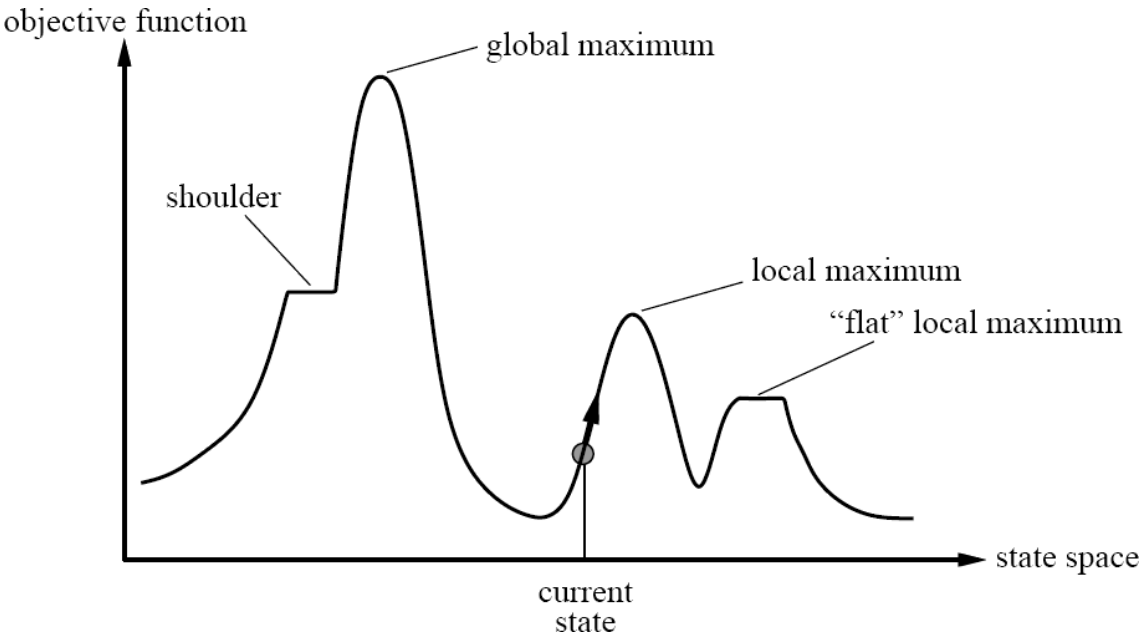
Classe de problemas P: Problemas de decisão que podem ser resolvidos por uma máquina determinista em tempo polinomial no tamanho da instância.

Classe de problemas NP: Problemas de decisão que podem ser resolvidos por uma máquina não determinista em tempo polinomial no tamanho da instância. Equivalentemente, se "adivinhar"a solução consigo verificá-la em tempo polinomial, logo P contido em NP.

Classe de problemas NP-difíceis: Aqueles que são mais difíceis do que todos os problemas em NP. Se o problema pertencer a NP, então o problema diz-se NP-completo. Os problemas NP-completos são os mais difíceis da classe NP!

Como atacar problemas difíceis?

Encontrar subclasses do problema que sejam interessantes e de resolução eficiente. Usar algoritmos de aproximação eficientes e recorrer a aproximações estocásticas.



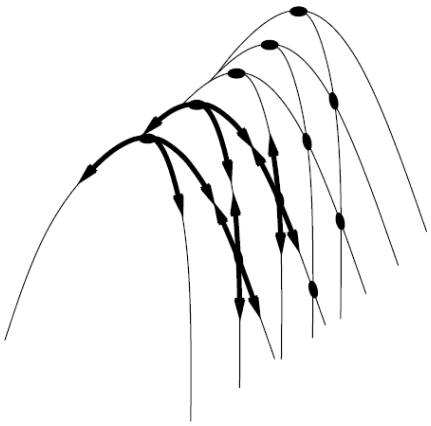
Tanto podemos definir o problema como maximizar a função objectivo (proveito) ou minimizar o custo (heurística).

Trepa Colinas (Hill Climbing)

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node
   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

A escolha é normalmente aleatória entre os sucessores com o mesmo valor para a função objectivo.

Problemas: pode ficar preso facilmente em máximos locais; travessia difícil em cristas de máximos locais; passeios aleatórios em planaltos. Em espaços contínuos, problemática na escolha do tamanho do passo, convergência lenta.



Trepa-colinas com recomeços aleatórios: Pode-se utilizar um número limitado de movimentos alterais para se tentar sair de planaltos, mas não funciona para planaltos que correspondem a máximos locais (substituir teste de ≤ por < e contar a sua utilização).

Para evitar ficarmos presos em máximos locais, tenta-se novamente com um novo estado inicial gerado aleatoriamente e guarda-se o melhor deles ao fim de um número determinado de iterações. O sucesso deste algoritmo depende muito da paisagem do espaço de estados, poucos máximos locais e planaltos é a situação ideal.

Procura Local Estocástica

O trepa-colinas é um exemplo elementar de um algoritmo de procura local estocástica. A procura é efectuada no espaço de soluções candidatas. Inicia-se a partir de uma solução candidata. O processo continua movendo-se iterativamente de uma solução candidata para outra na sua vizinhança. A decisão em cada passo é tomada tendo em conta apenas informação local limitada. A inicialização e a decisão podem ser aleatórias (probabilísticas). Pode-se utilizar memória adicional, por exemplo para guardar um número limitado de soluções candidatas visitadas recentemente.

SLS Decision:

```
procedure SLS-Decision(π)
   input: problem instance π in Π
   output: feasible solution in S'(Π) or ∅
   (s,m) := init(π);
   while not terminate(π,s,m) do
        (s,m) := step(π,s,m);
   end
   if s in S'(Π) then
        return s
   else
        return ∅
   end
```

SLS-Maximisation:

```
procedure SLS-Maximisation(π)
   input: problem instance π in Π
   output: feasible solution in S'(Π) or ∅
   (s,m) := init(π);
   incumbent := s;
   while not terminate(π,s,m) do
        (s,m) := step(π,s,m);
        if f(π,s) > f(π,incumbent) then
           incumbent := s;
        end
   end
   if incumbent in S'(Π) then
        return incumbent
   else
        return ∅
   end
```

Como evitar ficar preso em mínimos/máximos locais?

Exige um equilíbrio entre estratégias de:

Intensificação: tentativa de melhoramento da solução na vizinhança da solução candidata corrente.

Diversificação: tenta evitar a estagnação em zonas do espaço de procura que não contêm soluções de alta qualidade.

As duas estratégias definem os inúmeros algoritmos existentes. Maiores vizinhanças contêm mais e melhores soluções candidatas mas requerem mais tempo para as encontrar (normalmente o tempo cresce exponencialmente…).

Exemplos: Uninformed Random Picking, Uninformed Random Walking e Randomised Iterative Improvement.

Recristalização Simulada: Selecciona-se um movimento aleatoriamente em cadaiteração. Transita-se para esse estado se este for melhor do que o estado actual. A probabilidade de se movimentar para um estado pior decresce exponencialmente com o valor da mudança, ou seja, a temperatura T altera-se de acordo com o escalonamento definido, Temperatura inicial T0 (pode depender das propriedades da instância).
Actualização da temperatura, número de passos a cada temperatura (normalmente múltiplo da dimensão da vizinhança). Terminação normalmente baseada no rácio entre estados propostos versus aceites.

Se a temperatura for diminuída suficientemente devagar atinge-se o melhor estado, com probabilidade que se aproxima de 1.

$$e^{\frac{\Delta E}{T}}$$

Procura local em feixe: Mantêm-se *k* soluções candidatas em vez de apenas 1**.** Começa-se com *k* soluções geradas aleatoriamente. Em cada iteração, as vizinhanças das k soluções candidatas são geradas. Se alguma é o objectivo, parar; se não seleccionam-se as *k* melhores e repete-se.

Procura local em espaços contínuos: Muitos problemas podem ser representados como maximização/minimização de funções multi-dimensionais em Rn. Existem inúmeras técnicas mas iremos abordar brevemente aquelas baseadas no gradiente de uma função *f(x1,…, xn)* que se supõe diferenciável em Rn.

O gradiente da função f, denotado por ∇f define-se por:

$$\nabla f = \left(\frac{df}{x_1}, \ldots, \frac{df}{x_n}\right)$$

Subida pelo gradiente: Partindo de uma solução inicial x0, executa-se iterativamente o algoritmo até que a variação entre |xn+1–xn| atinja um erro predeterminado. Podemos variar em cada passo o parâmetro γn, que deve ser suficientemente pequeno.

Método de Newton-Rampson: É utilizado para obter as raízes de f(x)=0. O método para funções reais corresponde a iterar $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Para encontrar os extremos tem de se resolver a equação f'(x) = 0.

V - Algoritmos Genéticos

Algoritmos Genéticos são técnicas de optimização estocásticas inspiradas no processo de evolução por selecção natural. Começam com uma população de indivíduos (ou cromossomas) gerados aleatoriamente. Fazem evoluir em simultâneo a população (soluções alternativas) por intermédio de operadores de selecção, reprodução e mutação.

Os operadores genéticos permitem manter a diversidade da população e evitam que os AG convirjam prematuramente para um máximo local.

Com uma modelação correcta do problema, a qualidade média das gerações sucessivas será aumentada, em particular a do seu melhor indivíduo.

A população é constituída por um número geralmente fixo de indivíduos. Cada cromossoma (ou indivíduo) é representado por uma cadeia de símbolos de um alfabeto finito, normalmente uma cadeia binária de 0s e 1s. Os cromossomas são constituídos por genes e os valores que cada gene pode tomar são os alelos.

A qualidade de cada indivíduo é avaliada por uma função de mérito (*fitness function*) que influencia a probabilidade de selecção dos indivíduos para reprodução

A reprodução combina dois indivíduos para gerar novo(s) indivíduo(s). Os filhos podem sofrer mutações genéticas aleatórias no seu cromossoma antes de serem colocados na população (geração seguinte). Uma geração pode ser substituída integralmente pelos seus descendentes ou manter alguns dos seus indivíduos mais aptos.

Operação de Recombinação

O locus (ponto de corte) é escolhido aleatoriamente. Outros métodos permitem a reprodução com N pontos de corte, mas normalmente melhores resultados são obtidos com a recombinação uniforme, em que é criada aleatoriamente uma máscara para copiar o material genético.

Progenitores	Descendência
32752411	32748552
24748552	24752411

1 ponto de corte

Progenitores	Descendência
32752411	22758451
24748552	34742512

Uniforme com máscara 01110101

Operação de Mutação: Alteração aleatória de um valor dos genes, com uma probabilidade baixa (0.001 valor típico) Evita que o algoritmo genético fique preso em máximos locais.

Implementações de Algoritmos Genéticos

```
function GENETIC-ALGORITHM( problem, FITNESS-FN) returns an individual
   inputs: population, a set of individuals
           FITNESS-FN, a function that measures the fitness of an individual
  repeat
     new_population ← empty set
     loop for i from 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (random number ≤ mutation probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough or enough time has elapsed
  return best individual in population, according to FITNESS-FN

function REPRODUCE( x, y) returns an individual
   inputs: x, y, parent individuals
   n ← LENGTH(x)
   c ← random number from 1 to n
   return APPEND(SUBSTRING(x,1,c), SUBSTRING(y,c+1,n))
```

```
function GENETIC-ALGORITHM( problem, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual
  repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population)/2 do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      if (random number ≤ crossover probability) then
        child1, child2 ← REPRODUCE(x, y)
      else
        child1, child2 ← x, y
      endif
      if (random number ≤ mutation probability) then child1 ← MUTATE(child1)
      if (random number ≤ mutation probability) then child2 ← MUTATE(child2)
      add child1 and child2 to new_population
    population ← new_population
  until some individual is fit enough or enough time has elapsed
  return best individual in population, according to FITNESS-FN
```

Valores empíricos:

Para grandes populações (100 indivíduos)

Probabilidade de recombinação: 0.6
Probabilidade de mutação: 0.001

Para populações pequenas (30 indivíduos)

Probabilidade de recombinação: 0.9
Probabilidade de mutação: 0.01

O teorema do esquema: Um esquema é um padrão da forma H = (1 * 1 0 * 1 *) representado um conjunto de cromossomas (* = don't care). O número total de símbolos fixo 0 ou 1 é a ordem do esquema, O(H) = 4 para o caso anterior.

O comprimento definidor é a distância entre a menor e maior posição definidas no esquema, exemplo d(H) = 5. O número de instâncias de um determinado esquema cresce exponencialmente ao longo do tempo se estes possuírem três características:

Qualidade acima da média;
Baixa ordem;
Baixo comprimento definidor.

Os algoritmos genéticos apresentam paralelismo implícito: uma população de n indivíduos processa n^3 esquemas.

Outros operadores de selecção

Amostragem universal estocástica: Mecanismo para obter de uma vez só todos os indivíduos para reprodução.

Amostragem por posição: Ordenam-se os indivíduos por posição, utilizando esta posição para atribuir as probabilidades de selecção.

Amostragem por torneio: Seja *k* um parâmetro entre 0 e 1. Seleccionam-se dois indivíduos aleatoriamente da população. Gera-se um número aleatório *r*. Se $r < k$ fica-se com o melhor indivíduo; caso contrário selecciona-se para reprodução o pior deles.

Amostragem por estado estável: Substitui-se uma percentagem dos piores indivíduos.

Amostragem por truncatura: Só se deixa reproduzir uma percentagem dos melhores indivíduos.

Amostragem elitista: Retém-se sempre um pequeno número dos melhores indivíduos.

O Caixeiro-viajante

O problema do caixeiro-viajante requer cuidados especiais para a sua resolução com algoritmos genéticos.

Representação dos cromossomas;
Operador de recombinação;
Operador de mutação.

A representação natural recorre a uma lista ordenada de cidades para representar o circuito.

1	8	3	4	6	2	7	9	8

A dificuldade essencial é o operador de recombinação dado que qualquer dos métodos vistos anteriormente pode criar indivíduos inviáveis (abortos…). Existem duas aproximações possíveis, reparar os indivíduos ou garantir que a recombinação só gera indivíduos válidos.

Operador de recombinação PMX

Partially mapped crossover (PMX): escolhe uma subsequência a partir de um dos pais e preserva a ordem, na medida das possibilidades, do outro progenitor.

Seleccionam-se dois pontos de corte

p1 = **1 2 3 | 4 5 6 7 | 8 9**
p2 = **4 5 2 | 1 8 7 6 | 9 3**

A selecção induz o conjunto de equivalências **4↔1**, **5↔8**, **6↔7** e **7↔6**.

Os segmentos entre os pontos são trocados:

f1 = **x x x | 1 8 7 6 | x x**
f2 = **x x x | 4 5 6 7 | x x**

Copiam-se os restantes valores que não introduzam repetições

f1 = **x 2 3 | 1 8 7 6 | x 9**
f2 = **x x 2 | 4 5 6 7 | 9 3**

Recorre-se às equivalências para substituir os restantes valores

f1 = **4 2 3 | 1 8 7 6 | 5 9** (porque 4↔1 e 5 ↔8)
f2 = **1 8 2 | 4 5 6 7 | 9 3** (porque 4↔1 e 5 ↔8)

Operador de recombinação OX

Order crossover (OX): Selecciona-se uma subsequência e mantém-se a ordem do outro progenitor.

Seleccionam-se dois pontos de corte

p1 = **1 2 3 | 4 5 6 7 | 8 9**
p2 = **4 5 2 | 1 8 7 6 | 9 3**

Copiam-se os segmentos para os descendentes:

f1 = **x x x | 4 5 6 7 | x x**
f2 = **x x x | 1 8 7 6 | x x**

Copiam-se os valores do outro progenitor sequencialmente a partir do segundo ponto de corte, evitando repetições de cidades:

f1 = **2 1 8 | 4 5 6 7 | 9 3**
f2 = **3 4 5 | 1 8 7 6 | 9 2**

Outras variantes do OX (OX3)

Seleccionam-se dois pontos de corte:

p1 = **1 2 3 | 4 5 6 7 | 8 9**
p2 = **4 5 2 | 1 8 7 6 | 9 3**

Copiam-se os segmentos para os descendentes:

f1 = **x x x | 4 5 6 7 | x x**
f2 = **x x x | 1 8 7 6 | x x**

Em vez de se começar a partir do segundo ponto de corte começa-se pela ordem do outro progenitor desde o início:

f1 = **2 1 8 | 4 5 6 7 | 9 3**
f2 = **2 3 4 | 1 8 7 6 | 5 9**

Em vez de utilizar uma subsequência, pode-se gerar uma máscara aleatória para seleccionaras cidades a manter para a geração seguinte.

Operador de recombinação CX

Cycle crossover (CX): Seleccionam-se dois pontos de corte:

p1 = **1 2 3 4 5 6 7 8 9**
p2 = **4 1 2 8 7 6 9 3 5**

Começa-se na primeira cidade de p1:

f1 = **1 x x x x x x x x**

A respectiva em p2 é a 4, colocando-se essa cidade na posição de p1:

f1 = **1 x x 4 x x x x x**

A respectiva em p2 é a 8, colocando-se essa cidade na posição de p1:

f1 = **1 x x 4 x x x 8 x**

Itera-se até introduzir um ciclo (cidade já colocada):

f1 = **1 2 3 4 x x x 8 x**

Preenchem-se as restantes a partir da sequência no outro progenitor:

f1 = **1 2 3 4 7 6 9 8 5**

Operadores de mutação

Inversão (inverte subsequência):

1 2 | 3 4 5 6 | 7 8 9
1 2 | **6 5 4 3**| 7 8 9

Inserção (move arbitrariamente 1 cidade):

1 2 3 4 5 6 7 8 9
1 2 **7** 3 4 5 6 8 9

Deslocamento (move subsequência):

1 2 3 4 5 | 6 7 8 | 9
1 | **6 7 8**| 2 3 4 5 9

Troca (troca duas cidades):

1 2 3 4 5 6 7 8 9
7 2 3 4 5 6 **1** 8 9

Algoritmos meméticos

Após aplicação de qualquer um dos operadores genéticos e mesmo da inicialização da população, efectua-se uma procura local. Estes algoritmos são designados por algoritmos meméticos. Por exemplo, pode-se aplicar o algoritmo trepa-colinas para melhorar o indivíduo.

Considerando todas as trocas de duas ou três cidades e escolhendo a melhor alternativa até atingir um mínimo local (2-opt e 3-opt). É preciso ter cuidado na escolha do operador de mutação para não efectuaras mesmas operações que a procura local, uma possibilidade é não efectuar procura local após mutação.

Outros algoritmos de inspiração evolucionária:

Programação Genética: Evolução de programas

Estratégias Evolutivas: Evolução de vectores reais com distribuições normais para resolução de problemas de Engenharia. O operador dominante é a mutação

Programação Evolutiva: os indivíduos são máquinas de estados

Programação Genética

Cada indivíduo é um programa. A qualidade de um indivíduo é obtida por execução do programa relativamente a um conjunto de testes. O material genético encontra-se estruturado em árvore. É de tamanho variável durante o processo evolutivo.

O operador de recombinação troca sub-árvores entre os dois progenitores, preservando a sintaxe.

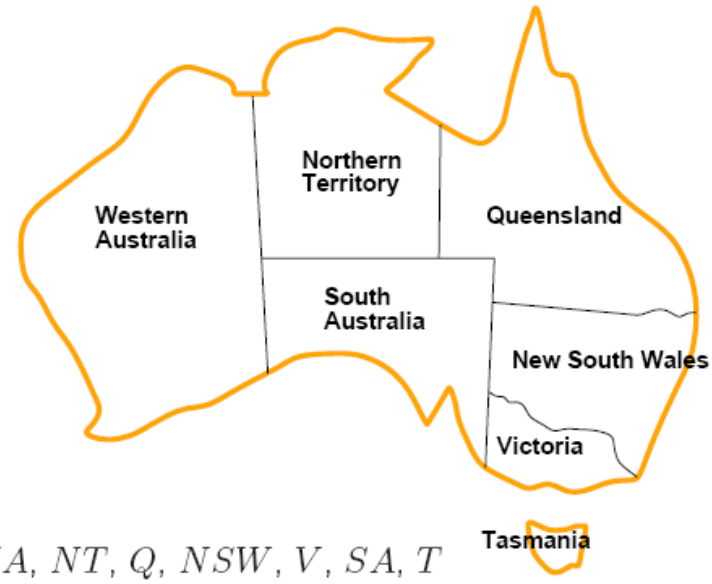
O operador de mutação substitui uma sub-árvore por outra construída aleatoriamente.

A população inicial é obtida pela geração aleatória de funções e terminais que constituem os programas.

VI - Problemas de Satisfação de Restrições

CSPs

Num csp o estado é definido por variáveis pertencentes a um domínio e o teste objectivo é um conjunto de restrições especificando as combinações permitidas para subconjuntos de variáveis. Permite recorrer a algoritmos genéricos melhores que os de procura.



Variáveis WA, NT, Q, NSW, V, SA, T
Domínios $D_i = \{red, green, blue\}$
Restrições: regiões adjacentes devem ter cores diferentes
e.g., $WA \neq NT$ (se a linguagem o permitir), ou
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

***CSP binário*:** cada restrição relaciona no máximo duas variáveis
***Grafo de restrições*:** nós são variáveis e arcos correspondem a restrições

Variáveis discretas

- ✓ domínios finitos; cardinalidade $d \Rightarrow O(dn)$ atribuições completas
  ex.CSPs Booleanos, incl. satisfatibilidade Booleana (NP-completo)
- ✓ domínios infinitos (inteiros, cadeias de caracteres, etc.)
- ✓ escalonamento: variáveis representam início/fim das tarefas
- ✓ utilizam linguagem de restrições: *StartJob*1 + 5 ≤ *StartJob*3
- ✓ restrições lineares são solúveis, não-lineares indecidíveis

Variáveis contínuas

- ✓ e.g., tempos de início/fim das observações do Telescópio Hubble

- ✓ restrições lineares resolúveis em tempo polinomial por métodos de programação linear

Restrições:

- ✓ **Unárias** restrições envolvendo apenas uma variável, e.g., *SA _= green*
- ✓ **Binárias** restrições envolvendo pares de variáveis, e.g., *SA _= WA*
- ✓ **Ordem superior** restrições envolvendo 3 ou mais variáveis, e.g., restrições das colunas em problemas cripto-aritméticos
- ✓ **Preferências** (restrições suaves), por exemplo, *red é* melhor do que *green.* Habitualmente representado atribuindo um custo a cada atribuição de variáveis → problemas de optimização com restrições.

Os estados são definidos pelos valores atribuídos até ao momento às variáveis

- ✓ **Estado inicial:** a atribuição vazia, { }
- ✓ **Função sucessor:** atribuir um valor a uma variável livre que não entre em conflito com a atribuição actual ⇒ falha se não existirem atribuições possíveis
- ✓ **Teste objectivo:** a atribuição não tem variáveis livres

Notas:

Toda a solução ocorre à profundidade *n* com *n* variáveis ⇒ utilização de procuras em profundidade primeiro.
O caminho é irrelevante, pode-se usar formulação de estado completo

Procura em profundidade primeiro para CSPs com atribuições a uma única variável é designada por procura com retrocesso (algoritmo básico para CSPs).

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING([], csp)

function RECURSIVE-BACKTRACKING(assigned, csp) returns solution/failure
  if assigned is complete then return assigned
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assigned, csp) do
      if value is consistent with assigned according to CONSTRAINTS[csp] then
          result ← RECURSIVE-BACKTRACKING([var = value|assigned], csp)
          if result ≠ failure then return result
  end
  return failure
```

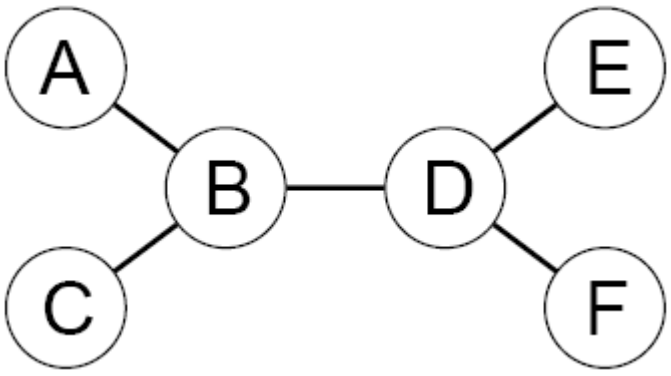
Métodos de melhoramento de algoritmos genéricos (Heurísticas):

- ✓ Escolha da variável mais constrangida (menos valores possíveis). Em caso de empate, seleccionar a variável com maior número de restrições nas restantes variáveis).
- ✓ Escolha do valor menos restritivo (o que elimine menos valores nas respectivas variáveis) ⇒ Importante quando queremos apenas uma solução, irrelevante caso contrário.
- ✓ Vigiar os valores possíveis das variáveis por atribuir e terminar a procura quando uma deixe de os ter.

Algoritmo de consistência de Arcos

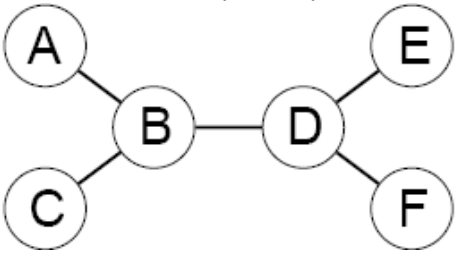
```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue

---------------------------------------------------------------

function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
      then delete x from DOMAIN[Xi]; removed ← true
  return removed
```

CSPs com estrutura arbórea



Teorema: se o grafo de restrições não tem ciclos, então o CSP pode ser resolvido em tempo *O*(*nd*2)
Comparar com CSPs genéricos, cujo pior caso temporal é *O*(*dn*)
Esta propriedade também se aplica ao raciocínio lógico e probabilístico: um exemplo importante da relação entre restrições sintácticas e a complexidade do raciocínio.

1. Escolher uma variável como raiz, ordenar variáveis da raiz para as folhas tal que o pai de um nó aparece primeiro do que todos os seus filhos na ordenação

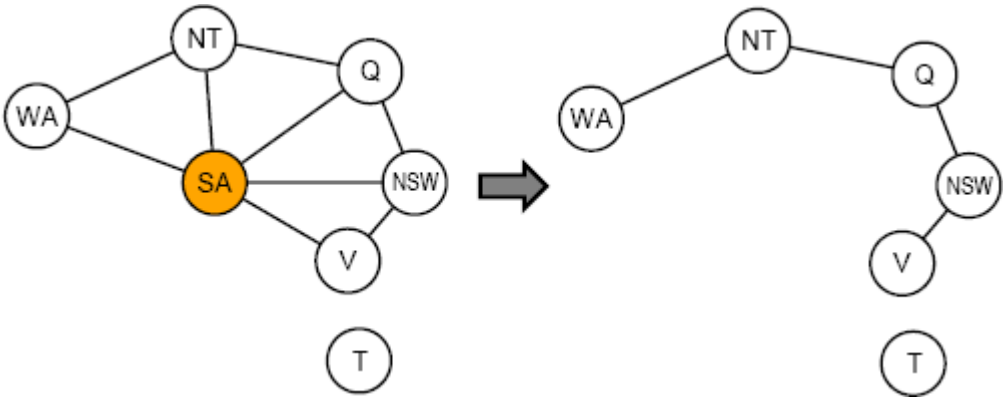


2. Para *j* de *n* até 2, aplicar *RemoveInconsistent*(*Parent*(*Xj*),*Xj*)
3. *j* de 1 até *n*, atribuir X_j consistentemente com *Parent*(*Xj*)

Nota: Resolvidos em tempo linear

CSPs de estrutura quasi-arbórea

Condicionamento: instanciar uma variável, reduzir os domínios dos nós vizinhos



Condicionamento por conjunto de corte: instanciar (de todas as maneiras) um conjunto de variáveis tal que o grafo resultante seja árvore
Conjunto de corte de tamanho c ⇒ tempo de execução $O(dc \cdot (n-c)d2)$, muito rápido para c pequeno.

Algoritmos locais para CSPs

Trepa-colinas, recristalização simulada funcionam habitualmente com estados "completos", i.e., todas as variáveis atribuídas

Aplicação a CSPs:

permitir estados com restrições por satisfazer
os operadores *reatribuem* valores a variáveis

Selecção de Variáveis: escolher aleatoriamente qualquer variável conflituante

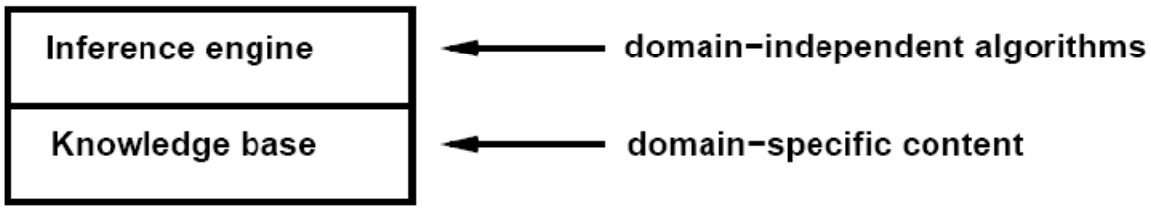
Selecção de Valores por intermédio da heurística *min-conflitos*: escolher valor que viola o menor número de restrições, i.e., trepa-colinas com $h(n)$ = número total de restrições violadas.

Algoritmo min-conflitos

```
function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max-steps, the number of steps allowed before giving up
  local variables: current, a complete assignment
                   var, a variable
                   value, a value for a variable

  current ← an initial complete assignment for csp
  for i = 1 to max-steps do
      var ← a randomly chosen, conflicted variable from VARIABLES[csp]
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var=value in current
      if current is a solution for csp then return current
  end
  return failure
```

VII - Agentes Lógicos

Bases de conhecimento(s)



Base de conhecimento = conjunto de **frases** numa linguagem **formal**

Aproximação **declarativa** na construção de um agente (ou outro sistema):

TELL ← informar o sistema do que precisa de saber

Seguidamente, o sistema pode perguntar a si próprio (ASK) o que deve fazer – As respostas obtidas (implicitamente) a partir da Knowledge Base.

Os agentes podem ser analisados quanto ao seu **nível de conhecimento** – i.e., aquilo que sabem, independentemente da sua implementação.

Ou quanto ao seu **nível de implementação** – i.e., estruturas de dados na Knowledge Base e algoritmos que a manipulam,

Um agente simples baseado em conhecimento

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

O agente deve ser capaz de:

- ✓ Representar estados, acções, etc.
- ✓ Incorporar novas percepções
- ✓ Actualizar representações internas do mundo
- ✓ Deduzir propriedades escondidas do mundo
- ✓ Deduzir acções apropriadas

Noções de Lógica

Lógicas são linguagens formais para a representação de informação que permitem a extracção de conclusões

Sintaxe define as frases permitidas da linguagem

Semântica define o "significado" das frases – i.e., define **verdade** de uma frase num mundo

Ex, a linguagem da aritmética

$x + 2 \geq y$ é uma frase (proposição); $x2 + y >$ não é uma frase

$x + 2 \geq y$ é verdade sse o número $x + 2$ não é menor do que o número y

$x + 2 \geq y$ é verdade num mundo em que $x = 7, y = 1$

$x + 2 \geq y$ é falso num mundo em que $x = 0, y = 6$

Conclusão Lógica

Conclusão significa que algo **segue** de outrem: KB $\vDash \alpha$

Da base de conhecimento KB conclui-se a frase α α se e somente se α é verdade em todos os mundos em que KB é verdade.

Da base de conhecimento KB contendo "o Boavista ganhou" e "o Porto ganhou" conclui-se "o Boavista ganhou ou o Porto ganhou"

Ex. De $x + y = 4$ conclui-se $4 = y + x$

Conclusão Lógica é uma relação entre frases (i.e. **sintaxe**) que se encontra baseada na **semântica**

Nota: o cérebro processa **sintaxe** (de algum tipo)

Modelos

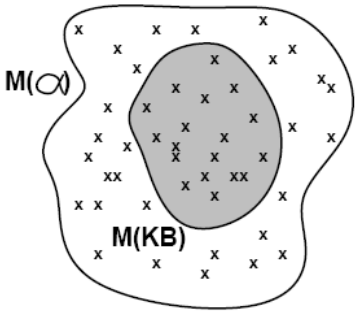
Os lógicos pensam normalmente em termos de modelos, que são mundos formalmente estruturados relativamente aos quais se pode avaliar a veracidade.

Diz-se que *m* é modelo de uma proposição α se α é verdade em *m*

M(α) é o conjunto de todos os modelos de α

Logo KB $\models \alpha$ se e somente se *M(KB)* $\subseteq$ *M(α)*

Ex. KB = Sporting ganhou e Benfica ganhou α = Benfica ganhou



Inferência

KB $\vdash_i \alpha$ = proposição α pode ser derivada de KB pelo procedimento i

Consequências de KB são o palheiro; α é a agulha. Conclusão Lógica = agulha no palheiro; inferência = encontrá-la

Fidedigno: i é fidedigno (ou sólido) se quando KB $\vdash_i \alpha$, então também é verdade que KB $\models \alpha$

Completo: i é completo se quando KB $\models \alpha$, então também é verdade que KB $\vdash_i \alpha$

Antecipação: definiremos uma lógica (lógica de primeira ordem) que é suficientemente expressiva para dizer quase tudo o que é interessante, e para a qual existe um procedimento de inferência fidedigno e completo. Ou seja, o procedimento responderá a qualquer questão que se segue daquilo que é conhecido pela KB.

Lógica Proposicional: Sintaxe

A lógica proposicional é a lógica mais simples – ilustra os conceitos básicos

Os símbolos (ou variáveis) proposicionais P_1, P_2 etc. são proposições (frases)

Se S é uma proposição, $\neg S$ é uma proposição (**negação**)

Se S_1 e S_2 são proposições, $(S_1 \wedge S_2)$ é uma proposição (**conjunção**)

Se S_1 e S_2 são proposições, $(S_1 \vee S_2)$ é uma proposição (**disjunção**)

Se S_1 e S_2 são proposições, $(S_1 \Rightarrow S_2)$ é uma proposição (**implicação**)

Se S_1 e S_2 são proposições, $(S_1 \Leftrightarrow S_2)$ é uma proposição (**bicondicional**)

Lógica Proposicional: Semântica

Cada modelo atribui verdadeiro/falso a cada símbolo proposicional

E.g. $P_{1,2}$ $P_{2,2}$ $P_{3,1}$
verdadeiro *verdadeiro* *falso*

(Com estes símbolos, 8 modelos possíveis, podem ser enumerados automaticamente)

Regras para avaliar a veracidade relativamente a um modelo *m*:

$\neg S$	é verdade sse	S	é falso		
$S_1 \wedge S_2$	é verdade sse	S_1	é verdade **e**	S_2	é verdade
$S_1 \vee S_2$	é verdade sse	S_1	é verdade **ou**	S_2	é verdade
$S_1 \Rightarrow S_2$	é verdade sse	S_1	é falso **ou**	S_2	é verdade
i.e.,	é falso sse	S_1	é verdade **e**	S_2	é falso
$S_1 \Leftrightarrow S_2$	é verdade sse	$S_1 \Rightarrow S_2$	é verdade **e**	$S_2 \Rightarrow S_1$	é verdade

Um processo recursivo simples avalia uma proposição arbitrária, e.g.,

$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = \mathit{verd} \wedge (\mathit{falso} \vee \mathit{verd}) = \mathit{verd} \wedge \mathit{verd} = \mathit{verd}$

Tabela de verdade para os conectivos

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$falso$	$falso$	$verd$	$falso$	$falso$	$verd$	$verd$
$falso$	$verd$	$verd$	$falso$	$verd$	$verd$	$falso$
$verd$	$falso$	$falso$	$falso$	$verd$	$falso$	$falso$
$verd$	$verd$	$falso$	$verd$	$verd$	$verd$	$verd$

Posições no mundo do Wumpus

Seja $P_{i,j}$ verdade se existir um poço em [i,j]
Seja $B_{i,j}$ verdade se existir um poço em [i,j]

$\neg P_{1,1}$
$\neg B_{1,1}$
$B_{2,2}$

"Poços causam brisa em casas adjacentes"

$B_{1,1} \quad (P_{1,2} \quad P_{2,1})$
$B_{2,1} \quad (P_{1,1} \quad P_{2,2} \quad P_{3,1})$

"Uma casa é ventosa *sse* existir um poço adjacente"

Inferência através de tabelas de verdade

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
false	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*
false	*false*	*false*	*false*	*false*	*false*	*true*	*false*	*true*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*true*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*false*	*true*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*false*

Inferência por enumeração

Enumeração em profundidade primeiro dos modelos todos é sólido e completo

```
function TT-ENTAILS?(KB, α) returns true or false
    symbols ← a list of the proposition symbols in KB and α
    return TT-CHECK-ALL(KB, α, symbols, [])

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
    if EMPTY?(symbols) then
        if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
        else return true
    else do
        P ← FIRST(symbols); rest ← REST(symbols)
        return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model)) and
                TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model))
```

Para *n* símbolos, complexidade temporal de $O(2^n)$ e espacial $O(n)$; problema é coNP-completo

Equivalência Lógica

Duas proposições são **logicamente equivalentes** sse forem verdadeiras nos modelos:
$\alpha \equiv \beta$ sse $\alpha \vDash \beta$ e $\beta \vDash \alpha$

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$ Comutatividade de ∧

$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$ Comutatividade de ∨

$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$ Associatividade de ∧

$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$ Associatividade de ∨

¬(¬α) ≡ α a eliminação da dupla negação

(α ⇒ β) ≡ (¬β ⇒ ¬α) contraposição

(α ⇒ β) ≡ (¬α ∨ β) eliminação da implicação

(α ⇔ β) ≡ ((α ⇒ β) ∧ (β ⇒ α)) eliminação da bicondicional

$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ de Morgan

$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ de Morgan

(α ∧ (β ∨ γ)) ≡ ((α ∧ β) ∨ (α ∧ γ)) Distributividade de ∧ sobre ∨

(α ∨ (β ∧ γ)) ≡ ((α ∨ β) ∧ (α ∨ γ)) Distributividade de ∨ sobre ∧

Validade e Satisfatibilidade

Uma proposição é **válida** se for verdadeira em *todos* os modelos,

Ex, True, A ∨ ¬A, A ⇒ A

Validade está relacionada com consequência através do **Teorema da Dedução**:

KB ⊨ α se e somente se (KB ⇒ α) é válida

Uma proposição é **satisfazível** se é verdadeira em *algum* modelo

Ex, A ∨ B, C

Uma proposição é **insatisfazível** se for verdadeira em *nenhum* modelo

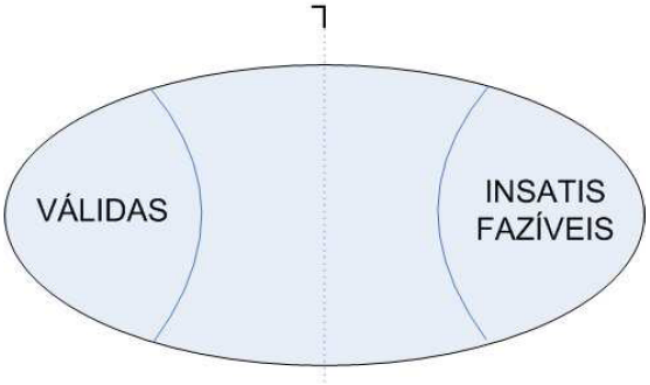
Ex, A ∨ ¬A

Insatisfatibilidade relaciona-se com consequência através de:

KB ⊨ α se e somente se (KB ∧ ¬α) é insatisfazível

i.e., demonstrar α por *reductio ad absurdum*

Geografia das expressões Booleanas



Eixo de simetria = negação

Métodos de Prova

Os métodos de prova agrupam-se em dois tipos:

Aplicação de regras de inferência

- ✓ Geração legítima (sólida) de novas proposições a partir das antigas
- ✓ **Prova** = uma sequência de aplicação de regras de inferência (Regras de inferência podem ser operadores em algoritmos de procura)
- ✓ Habitualmente obrigam à tradução das frases para uma **forma normal**

Verificação de modelos

- ✓ Enumuração por tabelas de verdade (sempre exponencial em *n*)
- ✓ Melhoramentos ao retrocessom, e.g., Davis-Putnam-Logemann-Loveland

Encadeamento para a frente e para trás

Forma de Horn (restrita)

KB = *conjunção* de *cláusulas de Horn*

Cláusula de Horn =

- Símbolo proposicional; ou
- (conjunção de símbolos) ⇒ símbolo

E.g., C ⋀ (B ⇒ A) ⋀ (C ⋀ D ⇒ B)

Modus Ponens (para a forma de Horn): completa para KB de Horn

$$\frac{\alpha_1, \ldots, \alpha_n, \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

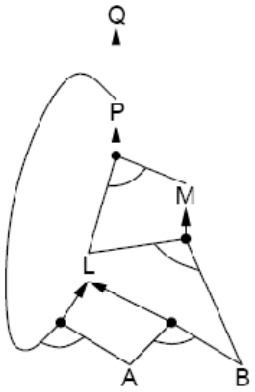
Pode ser utilizada com **encadeamento para a frente** ou **para trás**.

Estes algoritmos são muito naturais e executam em tempo *linear*

Encadeamento para a frente

Ideia: disparar toda a regra cujas premissas estão satisfeitas na KB, adicionar a sua conclusão á KB, até se chegar à pergunta

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



Algoritmo de Encadeamento para a frente

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known to be true

  while agenda is not empty do
      p ← POP(agenda)
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
              if count[c] = 0 then do
                  if HEAD[c] = q then return true
                  PUSH(HEAD[c], agenda)
  return false
```

Demonstração de Completude

EF deriva toda a proposição atómica que é concluída a partir de KB

1. EF atinge um ponto fixo em que não são derivadas novas proposições atómicas
2. Considere-se o estado final com um modelo *m*, atribuindo verdadeiro/falso aos símbolos
3. Toda a cláusula em KB inicial é verdadeira em *m*
   Demo: Suponha-se que a cláusula $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ é falsa em *m*
   Logo $a_1 \wedge \ldots \wedge a_k$ é verdadeiro em *m* e *b* é falso em *m*
   Logo o algoritmo não atingiu um ponto fixo!
4. Portanto *m* é modelo de KB
5. Se KB $\models q$, *q* é verdadeiro em *todo* o modelo de KB, incluindo *m*

Encadeamento para trás

Ideia: andar ao contrário a partir da pergunta *q*:
para provar *q* por ET, verificar se *q* já é sabido, ou provar por ET todas as premissas de alguma regra concluindo *q*

Evitar ciclos: verificar sem um novo-objectivo já se encontra na pilha de objectivos

Evitar trabalho repetido: verificar se o novo sub-objectivo

1. Já foi demonstrado verdadeiro, ou
2. Já falhou

Encadeamento para a frente vs. Para trás

EF é guiado pelos dados, cf. Processamento automático, inconsistente, e.g., reconhecimento de objectos, decisões rotineiras

Pode efectuar muito trabalho desnecessário que é irrelevante para o objectivo

ET é guiado pelo objectivo, adequado para a resolução de problemas, e.g.,
Onde estão as minhas chaves? Como posso entrar para o 2º ciclo?

Complexidade de ET pode ser muito inferior do que linear no tamanho da KB

Resolução

Forma Normal Conjuntiva (FNC—universal)
conjunção de *disjunções* de *literais*
cláusulas

E.g., (A ¬B) (B ¬C ¬D)

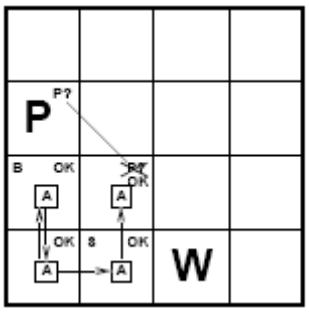
Regra de inferência **Resolução** (para FNC): completa para a lógica proposicional

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

em que ℓ_i e m_j são literais complementares. E.g.,

$$\frac{P_{1,3} \vee P_{2,2}, \qquad \neg P_{2,2}}{P_{1,3}}$$

Resolução é sólida e completa para a lógica proposicional



Conversão para FNC

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminar ⇔, substituindo $\alpha \Leftrightarrow \beta$ por $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
   $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
2. Eliminar ⇒, substituindo $\alpha \Rightarrow \beta$ por $\neg\alpha \vee \beta$.
   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
3. Deslocar ¬ para dentro recorrendo às leis de Morgan e dupla negação:
   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee \neg P_{2,1}) \vee B_{1,1})$
4. Aplicar distributividade (∨ sobre ∧):
   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{1,2} \vee B_{1,1})$

Algoritmo da Resolução

Prova por contradição, i.e., demonstrar que KB ∧ ¬α é insatisfazível

```
function PL-RESOLUTION(KB, α) returns true or false
    clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
    new ← { }
    loop do
        for each C_i, C_j in clauses do
            resolvents ← PL-RESOLVE(C_i, C_j)
            if resolvents contains the empty clause then return true
            new ← new ∪ resolvents
        if new ⊆ clauses then return false
        clauses ← clauses ∪ new
```

Um agente lógico no mundo do Wumpus

Formulação em lógica proposicional:

$$
\begin{aligned}
&\neg P_{1,1} \\
&\neg W_{1,1} \\
&B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y}) \\
&S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y}) \\
&W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,4} \\
&\neg W_{1,1} \vee \neg W_{1,2} \\
&\neg W_{1,1} \vee \neg W_{1,3} \\
&\vdots
\end{aligned}
$$

```
function PL-WUMPUS-AGENT(percept) returns an action
    inputs: percept, a list, [stench,breeze,glitter]
    static: KB, a knowledge base, initially containing the "physics" of the world
            x, y, orientation, the agent's position (initially [1,1]) and orientation (initially right)
            visited, an array indicating which squares have been visited, initially false
            action, the agent's most recent action, initially null
            plan, an action sequence, initially empty

    update x,y,orientation, visited based on action
    if stench then TELL(KB, S_x,y) else TELL(KB, ¬ S_x,y)
    if breeze then TELL(KB, B_x,y) else TELL(KB, ¬ B_x,y)
    if glitter then action ← grab
    else if plan is nonempty then action ← POP(plan)
    else if for some fringe square [i,j], ASK(KB, (¬ P_i,j ∧ ¬ W_i,j)) is true or
            for some fringe square [i,j], ASK(KB, (P_i,j ∨ W_i,j)) is false then do
                plan ← A*-GRAPH-SEARCH(ROUTE-PROBLEM([x,y], orientation, [i,j], visited))
            action ← POP(plan)
    else action ← a randomly chosen move
    return action
```

Limites de expressividade da lógica proposicional

- ✓ KB contém cláusulas capturando as leis da "física" para cada casa

  Para todo o tempo t e casa $[x, y]$,

  $$L^t_{x,y} \wedge FacingRight^t \wedge Forward^t \Rightarrow L^t_{x+1,y}$$

- ✓ Proliferação rápida do número de cláusulas

Resumo das classes de complexidade (P)

Um problema pertence à classe **P**, quando pode ser resolvido por uma máquina de Turing determinista em tempo polinomial.

Um problema de decisão é P-completo se pertence a P e *qualquer* problema na classe P pode ser reduzido a ele (em espaço logarítmico).

- ✓ Saber se um conjunto de cláusulas de Horn é satisfazível é um problema P-completo (HORNSAT)
- ✓ Saber qual o valor de um circuito booleano dados os seus inputs é P-completo (CIRCUIT VALUE)

Saber se um número inteiro é primo pertence a P

Resumo das classes de complex. (NP e coNP)

Um problema pertence à classe **NP**, quando pode ser resolvido por uma máquina de Turing não determinista em tempo polinomial

Um problema de decisão é **NP-completo** quando a solução pode ser verificada em tempo polinomial. Formalmente, um problema é NP-completo

1. Se pertence a NP
2. **Qualquer** problema na classe NP pode ser reduzido a ele por uma transformação polinomial

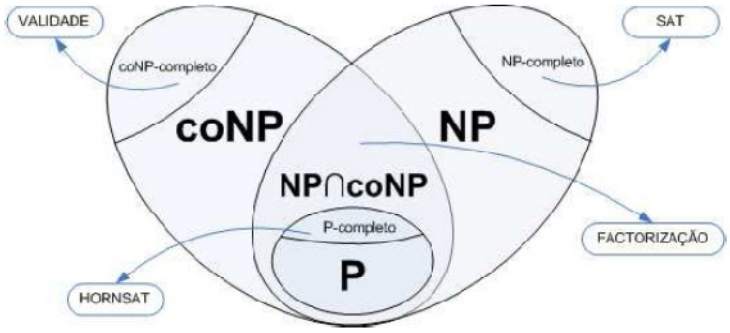
Caso um problema só obedeça ao critério (2) diz-se **NP-difícil**

Um problema pertence à classe **coNP** quando o seu complementar pertence à classe NP (pode-se verificar que **não** é solução em tempo polinomial).

Presume-se que P ≠ NP e que NP ≠ coNP

Complexidade e Lógica proposicional

Teorema de Cook-Levin (1971): O problema SAT é NP-completo.



NOTA: Testar a validade de um conjunto de fórmulas booleanas é um problema coNP-completo (VALIDITY)

Verificação eficiente de satisfatibilidade

- ✓ A existência de algoritmos eficientes de satisfatibilidade permite-nos lidar com problemas combinatórios complexos
- ✓ Todos os problemas NP-completos podem ser reduzidos ao problema da satisfatibilidade de cláusulas de lógica proposicional
- ✓ Duas famílias de algoritmos:
  - Baseados em retrocesso, e.g. Davis-Putnam-Longemann-Loveland
  - Por melhoramento iterativo (procura local), e.g. WalkSAT

Algoritmo de Davis-Putnam

Enumeração recursiva em profundidade primeiro de todos os modelos possíveis para proposições na FNC, com as seguintes melhorias:

- **Terminação com modelos parciais:** uma cláusula é verdadeira quando pelo menos um dos literais é verdadeiro. Logo, os restantes valores dos símbolos proposicionais são irrelevantes.
- **Heurística dos símbolos puros:** um símbolo é puro quando ocorre sempre com o mesmo sinal em todas as cláusulas. Nas proposições abaixo, A e ¬B são puros:
  (A ∨ ¬B) ∧ (¬B ∨ ¬C) ∧ (C ∨ A)
- **Heurística da cláusula unitária:** Quando todos os literais são falsos à excepção de um, o valor desse fica automaticamente definido. Pode originar propagações unitárias em cascata.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic
  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, [])

function DPLL(clauses, symbols, model) returns true or false
  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols–P, [P = value|model])
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols–P, [P = value|model])
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, [P = true|model]) or DPLL(clauses, rest,
[P = false|model])
```

Exemplo de avaliação:

$$a \vee \neg b \vee c$$
$$\neg b \vee \neg d$$
$$\neg a \vee \neg c$$
$$a \vee d$$
$$c \vee e$$
$$\neg e \vee f$$
$$\neg d \vee c \vee \neg f$$

Propriedades do algoritmo de Davis-Putnam

- ✓ Completo
- ✓ Eficaz na prática podendo resolver problemas de verificação de Hardware com 1 milhão de variáveis

WalkSAT

- ✓ Trepa-colinas no espaço de atribuições completas
- ✓ Em cada iteração o algoritmo escolhe uma cláusula não satisfeita e um símbolo dessa cláusula para trocar. A forma de escolha do símbolo a trocar de valor é ela própria aleatória, podendo ser:
  - Utilizando a heurística "min-conflitos" minimizando o número de cláusulas insatisfeitas no passo seguinte
  - Escolha aleatória do símbolo a trocar na cláusula ("passeio aleatório")

```
function WALKSAT(clauses, p, max-flips) returns a satisfying model or failure
   inputs: clauses, a set of clauses in propositional logic
           p, the probability of choosing to do a "random walk" move, typically
around 0.5
           max-flips, number of flips allowed before giving up

   model ← a random assignment of true/false to the symbols in clauses
   for i = 1 to max-flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol
from clause
      else flip whichever symbol in clause maximizes the number of satisfied clauses
   return failure
```

Propriedades do WalkSAT

- ✓ Incompleto
- ✓ Se uma proposição é insatisfazível então o algoritmo não termina: limita-se *max_flips…*
- ✓ Logo, procura no local não serve em geral para resolver o problema da consequência lógica
- ✓ Algoritmos locais como o WalkSAT são mais eficazes quando se espera que uma solução exista
- ✓ Muito eficiente na prática…

Problemas de satisfatibilidade difíceis

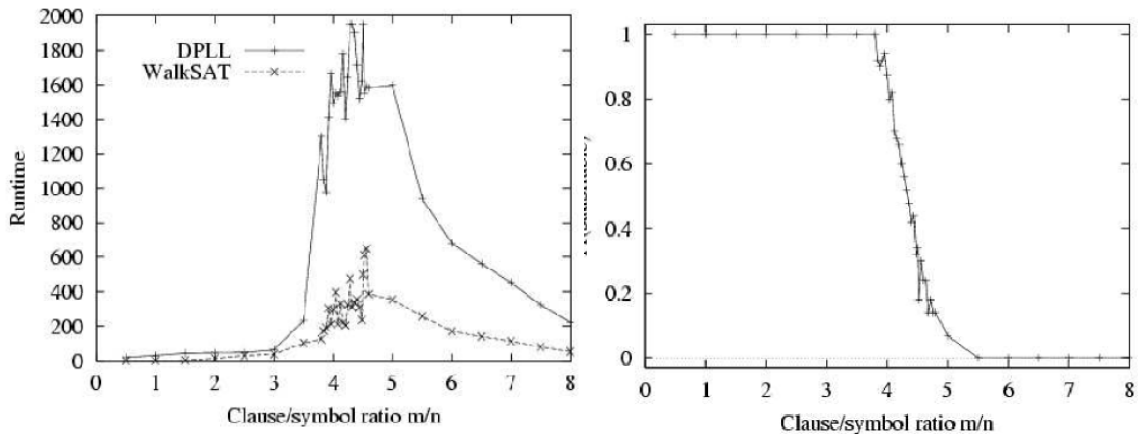
Considere cláusulas 3-CNF geradas aleatoriamente, e.g.:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$$

Seja

- ✓ *m* = numero de cláusulas
- ✓ *n* = numero de símbolos

Os problemas mais difíceis parecem concentrar-se perto do valor do rácio *m/n* = 4.3 (ponto crítico)



VIII - Lógicas não monótonas

Lógica monótona: Se algo é concluído a partir do que sei agora, então continuará a ser concluído no futuro.

Nas bases de dados usualmente adoptam-se as duas seguintes hipóteses simplificadoras:
Hipótese do Mundo Fechado (CWA - Closed World Assumption)
Hipótese do Domínio Fechado (assume-se apenas a existência dos objectos mencionados na teoria)
O que se conclui por hipótese do mundo fechado é obtido com:

$$Cn_{CWA}(\Sigma) = \{\phi \mid \Sigma \cup \Sigma_{CWA} \models \phi\}$$

onde ϕ é um literal positivo e

$$\Sigma_{CWA} = \{\neg\phi \mid \Sigma \not\models \phi\}$$

Uma teoria de lógica por omissão é um par {D, W} em que W é um conjunto de fórmulas de Linguagem de Primeira Ordem e D é um conjunto de regras por omissão com a forma

$$\frac{\varphi : \psi 1, \cdots, \psi n}{X}$$ *"Se φ for consistente assumir $\psi 1, \cdots, \psi n$ então X"*

Em que φ é o pré-requisito, $\psi 1, \cdots, \psi n$ é a justificação e X é a conclusão.

Cálculo de Extensões

E é uma extensão sse

$$E_0 = W$$
$$E_{i+1} = Cn(E_i) \cup \left\{\chi \mid \frac{\varphi : \psi_1, \ldots, \psi_n}{\chi} \text{ e } \varphi \in E_i \text{ e } \neg\psi_1 \notin E, \ldots, \neg\psi_n \notin E\right\}$$
$$E = \cup_{i=0}^{\infty} \; E_i$$

Em geral, o cálculo de extensões é indecidível para teorias de lógica de primeira ordem. Mas para o caso proposicional o problema é decidível.
Exemplo:

Normalmente, os cisnes são brancos

$$\frac{cisne(X) : branco(X)}{branco(X)}$$

Normalmente, os cisnes australianos são pretos

$$\frac{cisne(X) \wedge australiano(X) : preto(X)}{preto(X)}$$

Nada é preto e branco ao mesmo tempo

$$\forall_X \neg(branco(X) \wedge preto(X))$$

Para o exemplo dos cisnes temos duas extensões:
Uma extensão E_1 que contém

{*fcisne*(*bruce*); *australiano*(*bruce*); *branco*(*bruce*); *:preto*(*bruce*)}

Uma extensão E_2 que contém

{*fcisne*(*bruce*); *australiano*(*bruce*); *preto*(*bruce*);*:branco*(*bruce*)}

As regras por omissão anteriores são normais, pois têm a forma: $\frac{\varphi : X}{X}$

Se as regras numa teoria por omissão forem todas normais, então garante-se a existência de pelo menos uma extensão.

Lógica Autopistémica

A fórmula $\boldsymbol{L}\varphi$ significa "Eu acredito em φ" ou "Eu sei φ"

Expansão:

$$E = Cn\left(T \cup \{L\varphi \mid \varphi \in E\} \cup \{\neg L\varphi \mid \varphi \notin E\}\right)$$

Programação em Lógica

Programa em lógica definido ou positivo é um conjunto de regras da forma:

$A : -B_1,\ldots,B_m$

em que A, $B_1,\ldots,B_m$ são átomos da Lógica de Primeira Ordem. Se m = 0 temos um facto representado por A:

Uma regra lê-se "A se B_1 e . . . e B_m", correspondendo à implicação

$$\forall(B_1 \wedge \cdots \wedge B_m \Longrightarrow A)$$

ou seja, é um conjunto de clausulas de Horn.

Algumas variantes da programação em lógica admitem restrições de integridade com a forma:

$: -B_1,\ldots,B_m$

Os programas definidos datalog são aqueles que não utilizam símbolos de função nos seus termos, i.e. os termos ou são variáveis ou constantes.

Datalog (Database Logic)

Os programas datalog estão relacionados com as bases de dados, podendo ser utilizados para expressar regras e consultas a bases de dados dedutivas.
Os predicados num programa datalog normalmente classificam-se em:

1. Predicados extensionais: formados apenas por factos contendo a informação das relações da base de dados.
2. Predicados intencionais: definidos através de 1 ou mais regras datalog.
3. Predicados aritméticos: predicados cuja interpretação se encontra previamente fixada e inalterável.

Semântica de Programas em Lógica Definidos

Universo de Herbrand: conjunto de todos os termos formados por combinação das constantes e símbolos de função que ocorrem no programa.

Base de Herbrand: conjunto de todos os átomos básicos cujos argumentos são termos do Universo de Herbrand.

Programas em Lógica Normais

Um programa em lógica normal é um conjunto de regras:

A : -B_1,...,B_m, not C_1,...,not C_n

A semântica do *not* da linguagem PROLOG é definida por mecanismos operacionais com uma série de problemas práticos e teóricos

Dado um programa normal *P* completamente instanciado (ground), define-se a divisão de *P* por uma interpretação *I* como o programa definido:

$$\frac{P}{I} = \{A : -B_1, \ldots, B_m \mid A : -B_1, \ldots, B_m, not\, C_1, \ldots, not\, C_n \in P \text{ e } C_1, \ldots, C_n \notin I\}$$

Quais os modelos estáveis para:

preto : *-australiano; cisne; not branco:*
branco : *-cisne; not preto:*
cisne.
australiano.

Uma regra de um programa em lógica normal pode ser traduzida para o default:

$$\frac{B_1 \wedge \ldots \wedge B_m \quad : \neg C_1, \ldots, \neg C_n}{A}$$

As extensões estão em correspondência 1-1 com os modelos estáveis.

Complexidade datalog

Definem-se várias medidas de complexidade para o raciocínio com programas em lógica datalog:

- ✓ data complexity: fixa-se o programa, varia-se a base de dados (predicados intensionais) e a consulta.
- ✓ program complexity: fixa-se a base de dados, varia-se a base de dados extensional e a consulta.
- ✓ combined complexity: quer os predicados intencionais, quer os predicados extensionais e a consulta fazem parte do input.

	data complexity	(combined) complexity
definidos	P-complete	EXPTIME-complete
não recursivos estratificados	polinomial	PSPACE-complete
estratificados	P-complete	EXPTIME-complete
Bem-fundado	P-complete	EXPTIME-complete
Estável	coNP-complete	co-NEXPTIME

IX - Lógica de Primeira Ordem

A lógica proposicional é *declarativa*: a sintaxe permite expressar factos. Permite representar informação parcial, disjuntiva e negativa (contrariamente à maioria das estruturas de dados e bases de dados). É *composicional*: o significado de $B_{1,1}$ $P_{1,2}$ é obtido compondo o significado de $B_{1,1}$ e de $P_{1,2}$. O significado da lógica proposicional é *independente do contexto* (contrariamente à linguagem natural, em que o significado depende do contexto). Tem um poder expressivo muito limitado (contrariamente à linguagem natural).

Enquanto a lógica proposicional assume que o mundo contém *factos*, a lógica de primeira ordem (tal como a linguagem natural) assume que o mundo pode conter *objectos*, *relações* e *funções*.

Existem inúmeras lógicas, variando com o seu domínio de aplicação:

Lógicas terminológicas;
Lógicas de primeira ordem tipificadas;
Lógicas de segunda ordem;
Lógicas de ordem superior;
Lógicas de primeira ordem intuicionistas;

Lógicas modais.

O vocabulário da LPO é constituído pelos seguintes elementos:

1. Símbolos de constantes de predicados de aridade ≥ 1	$Brother,\ >,\ Irmao,\ Gato,\ \ldots$
2. Símbolos de constantes de objectos	$KingJohn,\ 2,\ UNL,\ Portugal,$ $Benfica, Reitor, \ldots$
3. Símbolos de constantes de funções	$Sqrt,\ LeftLegOf, \ldots$
4. Um número infinito de variáveis	$x,\ y,\ a,\ b, \ldots$
5. Conectivos lógicos	$\wedge\ \vee\ \neg\ \Rightarrow\ \Leftrightarrow$
6. Quantificadores	$\forall\ \exists$
7. Parêntesis esquerdo, direito e vírgula	$(\ )\ ,$
8. Igualdade	$=$

A lógica de primeira ordem não atribui qualquer interpretação pré-definida aos seus símbolos não lógicos: constantes de predicados, objectos e funções.

Regras de formação: Termos

O poder expressivo adicional da lógica de primeira ordem advém da sua possibilidade de referir objectos no domínio de discurso. Sintacticamente, os termos da lógica de primeira ordem denotam esses objectos: Um termo é definido recursivamente de acordo com as seguintes regras:

Uma constante de objecto é um termo (denotando um objecto concreto do domínio de discurso)

Uma variável é um termo (denotando um objecto "anónimo" do domínio de discurso)

Qualquer expressão $f(t_1,...,t_n)$ é um termo, em que os seus n 1 argumentos são termos e f é um símbolo de função com aridade n. Também é designado por expressão funcional.

Exemplo:

Objectos

- 1, 1.54, i, e, 12e40, pi, -3, MMVII, 0x20
- Portugal, UNL, Benfica
- Tweety, Diabo, Bem, Unicórnio
- Abc, Xpto123, Key123, 'Uma cadeia de caracteres muito longa'

Expressões funcionais

- Exp(1.0), Exp(Mult(I,Pi)), +(x,0,0.35), 0 * x (notação infixa), $Log(x,2)+Ln(E)$
- $Idade(Carlos,$ '17-04-2007 10:23:00 GMT')
- $Peso(Manuel)$
- 'C' & '++'
- Mãe(Árbitro(Jogo(Sporting,Benfica,Época200607))

Frases atómicas:

As frases ou fórmulas atómicas são construídas a partir dos símbolos de predicados e termos na linguagem:

Se *P* é um símbolo constante de predicado de aridade *n* e cada $termo_i$ ($1 \leq n$) são termos, então $P(a_1;...;a_n)$ é uma fórmula atómica.

Se a linguagem inclui igualdade, então $termo_1 = termo_2$ é uma fórmula atómica.

Exemplo:

$$Brother(KingJohn, RichardTheLionheart)$$

$$> (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))$$

$$Exp(I * Pi) + 1 = 0$$

$$Matriculado(s123, inf, ciclo1)$$

$$0 + x = x$$

$$Chove$$

$$Arco(a1, a2)$$

As fórmulas bem formadas (fbf) definem-se recursivamente através das seguintes regras:

- ♢ Qualquer frase atómica é uma fórmula bem formada (fbf)
- ♢ Se ϕ é uma fórmula bem formada então $\neg\phi$ é uma fbf.
- ♢ Se ϕ and ν são fbfs então $(\phi \wedge \nu)$, $(\phi \vee \nu)$, $(\phi \Rightarrow \nu)$ e $(\phi \Leftrightarrow \nu)$ também são fbfs.
- ♢ Se ϕ é uma fbf e x é uma variável então $\forall x \ \phi$ e $\exists x \ \phi$ é uma fbf.
- ♢ Nada mais é uma fbf.

Frases complexas são construídas a partir de frases atómicas utilizando os conectivos e os quantificadores.

Exemplos:

$$Sibling(KingJohn, Richard) \Rightarrow Sibling(Richard, KingJohn)$$

$$>(1,2) \vee \leq(1,2)$$

$$>(1,2) \wedge \neg>(1,2)$$

$$\forall x \ \forall y \ (x + y = y + x)$$

$$\forall x \ \forall y \ \exists z \ (x < y \Rightarrow (x < z \wedge z < y))$$

$$\forall y \ \exists x \ Progenitor(x, y)$$

$$\forall x \ (Humano(x) \Leftrightarrow (Mulher(x) \vee Homem(x)))$$

$$\exists x \ (Humano(x) \wedge \neg \exists y \ Progenitor(x, y))$$

Variáveis livres e ligadas:

♢ Se ϕ é uma fórmula atómica então x é livre sse x ocorre em ϕ.

♢ x é livre em $\neg\phi$ sse x é livre em ϕ.

♢ x é livre em $(\phi \wedge \nu)$, $(\phi \vee \nu)$, $(\phi \Rightarrow \nu)$, $(\phi \Leftrightarrow \nu)$ sse x é livre em ϕ ou ν.

♢ x é livre em $\forall y \ \phi$ ou $\exists y \ \phi$ sse $x \neq y$ e x é livre em ϕ.

De forma semelhante define-se a noção de variável ligada (aquelas que ocorrem no âmbito de algum quantificador). Uma variável pode estar livre e ligada na mesma fórmula:

$$(\forall x \ (R(x, y) \Rightarrow P(x)) \wedge \forall y \ (\neg R(x, y) \vee \forall x \ P(x)))$$

Nota: Qualquer fórmula pode ser reescrita numa fórmula equivalente em que as variáveis livres e ligadas são disjuntas.

Semântica da Lógica de Primeira Ordem

As fbfs são avaliadas em interpretações (ou estruturas) constituídas por um par *M* =< *D, I* > em que *D* é um conjunto não vazio (o domínio de discurso) e *I* uma função de interpretação. O domínio de discurso contém 1 ou mais objectos (elementos do domínio) e relações entre eles.

A função de interpretação especifica referentes para:

símbolos de constante → objectos $I(c) \in D$ ou
símbolos de predicado → relações $I(P) \subseteq D^n$ para predicado P/n
símbolos de função → relações funcionais $I(f) : D^n \rightarrow D$

Uma frase atómica *predicado(termo$_1$; ... ; termo$_n$)* é verdade sse os objectos referidos por *termo$_1$, ... ,termo$_n$* se encontram na relação referida por *predicado*. Quando temos fórmulas com variáveis livres é necessário considerar atribuições de variáveis que mapeiam variáveis em elementos do domínio de discurso.

A noção de verdade (relativa) em LPO é capturada através da relação de satisfação. Seja *M* uma interpretação e *s* uma atribuição de variáveis em *M*.

$M, s \models t_1 = t_2$ sse $(t_1)_{\mathrm{M}} = (t_2)_{\mathrm{M}}$
$M, s \models P(t_1, \ldots, t_n)$ sse $((t_1)_{\mathrm{M}}, \ldots, (t_n)_{\mathrm{M}}) \in I(P)$
$M, s \models \neg\phi$ sse não é o caso $M, s \models \phi$
$M, s \models (\phi \wedge \nu)$ sse $M, s \models \phi$ e $M, s \models \nu$
$M, s \models (\phi \vee \nu)$ sse $M, s \models \phi$ ou $M, s \models \nu$
$M, s \models (\phi \Rightarrow \nu)$ sse não é o caso $M, s \models \phi$ ou $M, s \models \nu$
$M, s \models \forall x \ \phi$ sse $M, s' \models \phi$, para toda a atribuição de variáveis s' idêntica a s excepto possivelmente na variável x
$M, s \models \exists x \ \phi$ sse $M, s' \models \phi$, para alguma de atribuição de variáveis s' idêntica a s excepto possivelmente na variável x

Consequência Lógica

♢ Uma fórmula ϕ é satisfazível se existir uma interpretação M e uma atribuição de variáveis s tal que $M, s \models \phi$.

♢ Um conjunto de fbfs Γ é satisfazível se existir uma intepretação M e uma atribuição de variáveis s tal que $M, s \models \psi$ para toda a fórmula ψ de Γ. Se Γ for um conjunto fechado de fórmulas diz-se que M é um modelo de Γ.

♢ Se Γ é um conjunto de sentenças então Uma fórmula ϕ é logicamente verdadeira ou válida se $M, s \models \phi$ para toda a interpretação M e atribuição de variáveis s (representado por $\models \phi$).

♢ Seja Γ um conjunto de fórmulas bem formadas e ϕ uma fbf. Diz-se que ϕ é uma consequência de Γ sse para toda a interpretação M e atribuição de variáveis s se $M, s \models \psi$ para toda a fórmula ψ de Γ então $M, s \models \phi$. Representa-se este facto através de $\Gamma \models \phi$.

Grupos Abelianos

Os grupos são definidos pelos seguintes axiomas, num vocabulário contendo uma constante *e*, um símbolo de função unário $^{-1}$ e um símbolo de função binário $\odot$:

$$\forall x \quad e \odot x = x \wedge x \odot e = x$$

$$\forall x \quad x^{-1} \odot x = e \wedge x \odot x^{-1} = e$$

$$\forall x \quad \forall y \quad \forall z \quad (x \odot y) \odot z = x \odot (y \odot z)$$

Se o grupo for comutativo diz-se que é abeliano em homenagem a Niels Henrik Abel.

$$\forall x \quad \forall y \quad x \odot y = y \odot x$$

Quantificação Universal

$\forall \langle variáveis \rangle \ \langle frase \rangle$

Toda a gente na UNL é inteligente:
$\forall x \ \ Em(x, UNL) \Rightarrow Inteligente(x)$

$\forall x \ \ P$ é verdade num dado modelo e dada interpretação nesse modelo sse P é verdade para todo o objecto x do modelo (i.e. x percorre todos os objectos possíveis do modelo)

Pode ser entendido como a conjunção das instanciações de P

$$\begin{aligned} & Em(ReiAfonsoI, UNL) \Rightarrow Inteligente(ReiAfonsoI) \\ \wedge \ & Em(Ana, UNL) \Rightarrow Inteligente(Ana) \\ \wedge \ & Em(UNL, UNL) \Rightarrow Inteligente(UNL) \\ \wedge \ & \ldots \end{aligned}$$

Erro comum a evitar:

Normalmente, $\Rightarrow$ é o o conectivo principal de $\forall$

Erro comum: utilizar $\wedge$ como conectivo principal de $\forall$:

$\forall x \ \ Em(x, UNL) \wedge Inteligente(x)$

significa "Toda a gente está na UNL e toda a gente é inteligente"

Quantificação Existencial

$\exists \langle variáveis \rangle \ \ \langle frase \rangle$

Alguém em Almada é inteligente:
$\exists x \ \ Em(x, Almada) \wedge Inteligente(x)$

$\exists x \ \ P$ é verdade num dado modelo e dada interpretação nesse modelo sse P é verdade para algum objecto x possível do modelo

Pode ser entendido como a disjunção das instanciações de P

$$\begin{aligned} & Em(ReiAfonsoI, Almada) \wedge Inteligente(ReiAfonsoI) \\ \vee \ & Em(Ana, Almada) \wedge Inteligente(Ana) \\ \vee \ & Em(Almada, Almada) \wedge Inteligente(Almada) \\ \vee \ & \ldots \end{aligned}$$

Outro erro comum a evitar:

Normalmente, $\wedge$ é o conectivo principal de $\exists$

Erro comum: utilizar $\Rightarrow$ como conectivo principal de $\exists$:

$\exists x \ \ Em(x, Almada) \Rightarrow Inteligente(x)$

é verdade se existir alguém que não está em Almada!

X - Inferência em Lógica de Primeira Ordem

Instanciação Universal

Qualquer instanciação de uma frase quantificada universalmente é consequência desta última:

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

para qualquer variável *v* e termo básico (concreto) *g*

E.g., $\forall x \;\; King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ origina

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$$
$$\vdots$$

Instanciação Existencial

Para qualquer frase α, variável v, e símbolo de constante k que não ocorre na base de conhecimento:

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

E.g., $\exists x \;\; Crown(x) \wedge OnHead(x, John)$ origina

$$Crown(C_1) \wedge OnHead(C_1, John)$$

desde que *C*1 seja um novo símbolo de constante, designado por constante de Skolem.

IU pode ser aplicado repetidamente para *adicionar* novas frases; a nova KB é logicamente equivalente à inicial. IE pode ser aplicada uma vez para *substituir* a frase existencial; a nova KB *não é equivalente* à inicial, mas é satisfazível sse a KB inicial era satisfazível!

Redução à Inferência Proposicional

Suponhamos que a KB contém apenas o seguinte:

Instanciando a frase universal de *todas as maneiras possíveis*, ficamos com

A nova KB foi proposicionada: os símbolos proposicionados são:
King(John), Greedy(John), Evil(John), King(Richard), etc

Redução

Resultado: uma frase básica é consequência da nova KB sse é consequência da KB original
Resultado: toda a KB em LPO pode ser proposicionalizada preservando a relação de consequência lógica
Ideia: proposicionalizar KB e pergunta, aplicar resolução, devolver resultado
Problema: com símbolos de função, existe um número infinito de termos básicos, e.g., *Father*(*Father*(*Father*(*John*)))
Teorema: Herbrand (1930). Se frase *α* é consequência de uma KB em LPO, então é consequência de um subconjunto *finito* da KB proposicional
Problema: funciona se *α* é consequência, pode não terminar se *α* não é consequência. Pode gerar inúmeras frases irrelevantes

Unificação

Podemos obter a conclusão imediatamente se conseguirmos encontrar uma substituição θ tal que *King*(*x*) e *Greedy*(*x*) concordem com *King*(*John*) e *Greedy*(*y*)
θ = {*x/John, y/John*} funciona
UNIFICAR(α,β) = θ se α θ = β θ

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	$\{x/OJ, y/John\}$
$Knows(John, x)$	$Knows(y, Mother(y))$	$\{y/John, x/Mother(John)\}$
$Knows(John, x)$	$Knows(x, OJ)$	$fail$

Standardização evita colisões de nomes de variáveis, e.g., *Knows*(z_{17};*OJ*)

Algoritmo de Encadeamento para a Frente

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Propriedades: Correcto para cláusulas definidas de primeira ordem (demonstração semelhante à do caso proposicional). Pode não terminar no caso geral se α não é consequência.

Inevitável: consequência com cláusulas definidas é semidecidível

Eficiência: Observação simples: não é necessário utilizar uma regra na iteração *k* se a premissa não foi adicionada na iteração *k*-1 $\Longrightarrow$ testar regras cuja premissa contém apenas literais adicionados recentemente. O mecanismo de concordância pode ser dispendioso
Indexação nas Bases de Dados permite a obtenção em tempo *O*(1) de factos conhecidos.
Concordância de premissas conjuntivas com factos conhecidos é NP-difícil. Encadeamento para a frente é amplamente utilizado em bases de dados dedutivas

Algoritmo de Encadeamento para a Frente

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $goals$, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution $\{\}$
  local variables: ans, a set of substitutions, initially empty

  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each r **in** KB where STANDARDIZE-APART(r) = $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    $ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n | \text{REST}(goals)]$, COMPOSE($\theta', \theta$)) $\cup$ ans
  return ans

Propriedades

Pesquisa da prova recursivamente em profundidade primeiro: espaço é linear no tamanho da prova.
Incompleto devido a ciclos infinitos

- ✓ Verificação do objectivo corrente com todos os outros na pilha

Ineficiente devido às subconsultas repetidas (de sucesso e de falha)

- ✓ Memorização dos resultados anteriores (espaço extra!)

Amplamente utilizado (sem melhoramentos!) na programação em lógica

Sistemas Prolog

Essência: encadeamento para trás com cláusulas de Horn
Programa = conjunto de cláusulas = head :- literal$_1$,..., literal$_n$.
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
Unificação eficiente sem teste de ocorrência
Obtenção eficiente de cláusulas
Encadeamento para trás em profundidade primeiro, da esquerda para a direita
Predicados de sistema para efectuar aritmética etc. (*X is Y*Z+3*)
Assunção do Mundo Fechado ("negação por falha") → dado *alive(X) :- not dead(X), alive(joe)* sucede se *dead(joe)* falha.

XI - Planeamento

Procura vs. Planeamento

Sistemas de Planeamento

1. Abrem a representação das acções e do objectivo para permitir selecção
2. Divisão e conquista por resolução de sub-objectivos
3. Não obriga à construção sequencial de soluções

	Procura	Planeamento
Estados	Estruturas de dados Java	Frases lógicas
Acções	Código Java	Pre-condições/resultados
Objectivos	Código Java	Frase Lógica (conjunção)
Plano	Sequência a partir de situação inicial	Restrições as acções

Planeamento Clássico

Consideram-se apenas ambientes:

- ✓ Totalmente Observáveis
- ✓ Deterministas
- ✓ Finitos
- ✓ Estáticos
- ✓ Discretos

Cálculo de Situações

O Formalismo em lógica de primeira ordem permite o raciocínio sobre o resultado das acções.

O cálculo de situações evita lidar explicitamente com o tempo, recorrendo em vez disso a *situações.*

Uma situação representa o estado resultante da aplicação de uma acção.

- ✓ **Acções** são denotadas por termos lógicos (Forward, Move, etc...)
- ✓ **Situações** são denotadas por termos lógicos, construídos a partir a situação inicial (S0) e por **aplicação de uma acção a uma situação**, sendo representado pela expressão funcional *Result(a;s).*
- ✓ **Fluentes** são predicados e funções que variam de uma situação para outra.

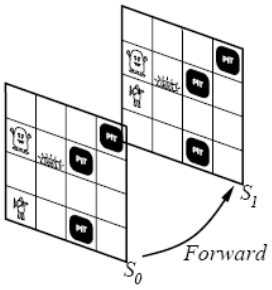
- ✓ Por convenção utiliza-se o último argumento do predicado ou função para identificar a situação a que se refere.
- ✓ Podem-se ainda utilizar **predicados ou funções eternas/intemporais**

Exemplo:

Factos verificam-se em **situações**, em vez de serem **eternos**.
Ex: *Holding(Gold;Now)* e não *Holding(Gold)*

Cálculo de Situações: uma das formas de representar a mudança em Lógica de Primeira Ordem.
Juntar um argumento de situação a cada predicado não eterno.
Ex: Now em *Holding(Gold;Now)* denota a **situação**

Situações são ligadas por intermédio da função *Result*. *Result(a;s)* é a situação resultante de se aplicar *a* em *s*.
Ex: *Result(Forward;S0)*



Descrevendo Acções

Modelo simplificado do mundo Wumpus em que é ignorada a orientação e em que o agente se pode deslocar para qualquer casa adjacente.

Assuma-se que o agente está na casa [1;1] e o outro na casa [1;2].

Vamos utilizar os **fluentes** *At(o;x;s)* e *Holding(o;s)* para representar que *o* se encontra na casa *x* na situação *s* e que o **agente** possui *o* na situação *s*, respectivamente.

Descrição da Situação Inicial

$$At(o, x, S_0) \equiv [(o = Agent \wedge x = [1,1]) \vee (o = G_1 \wedge x = [1,2])]$$
$$\neg Holding(o, S_0)$$
$$Gold(G_1)$$
$$Adjacent([1,1],[1,2]) \wedge Adjacent([1,2],[1,1])$$

Nota: O conhecimento *At(Agent; [1;1] ; S0) ∧ At(Gold; [1;2]; S0)* não captura a totalidade da situação, pois não especifica conhecimento negativo relativamente ao fluente At = 3 **(Princípio do Mundo Aberto).**

Tarefas que Pretendemos Efectuar

- ✓ **Projecção:** Deduzir o resultado da aplicação de uma sequência de acções.
  - Por exemplo, demonstrar que:
  - At(G1 ; [1;1] ; Result (Go([1;2] ; [1;1]) ; Result(Grab (G1) ; Result (Go ([1;1] ; [1;2]); S0)))
- ✓ **Planeamento:** Encontrar a sequência de acções que permitem atingir o objectivo
- ✓ $\exists_{s,g}\ Gold(g) \wedge Holding(g,s)$

Descrevendo acções no cálculo de situações

Uma acção pode ser descrita com dois axiomas: axioma da possibilidade e axioma de efeito.

- ✓ **Axioma de Possibilidade**: *Precondições => Poss(a;s)*

$$\begin{array}{lcl} At(Agent,x,s) \wedge Adjacent(x,y) & \Rightarrow & Poss(Go(x,y),s) \\ Gold(g) \wedge At(Agent,x,s) \wedge At(g,x,s) & \Rightarrow & Poss(Grab(g),s) \\ Holding(g,s) & \Rightarrow & Poss(Release(g),s) \end{array}$$

- ✓
- ✓ **Axioma de Efeito**: *Poss(a;s)* => mudanças devidas à acção

$$\begin{array}{lcl} Poss(Go(x,y),s) & \Rightarrow & At(Agent,y,Result(Go(x,y),s)) \\ Poss(Grab(g),s) & \Rightarrow & Holding(g,Result(Grab(g),s)) \\ Poss(Release(g),s) & \Rightarrow & \neg Holding(g,Result(Release(g),s)) \end{array}$$

- ✓

Interrogando a Teoria

Será que se pode concluir

At(Agent; [1; 2];Result(Go([1; 1]; [1; 2]); S0)) ? **SIM**

E

At(G1; [1; 2];Result(Go([1; 1]; [1; 2]); S0)) ? **NÃO**

Problema: Não basta dizer aquilo que se altera com a acção mas também se necessário dizer **aquilo que fica alterado**. Os axiomas de **quiescência** (frame axioms) especificam o que se **mantém inalterado com a acção.**

$$At(o,x,s) \wedge (o \neq Agent) \wedge \neg Holding(o,s) \Rightarrow At(o,x,Result(Go(x,y),s))$$

Se existirem *F* **fluentes** e *A* **acções** necessitaremos de, no pior caso, ***AxF* axiomas.**

Problemas a enfrentar

Problema da quiescência: encontrar uma forma elegante de lidar com o que se mantém inalterável.

a) *Representação* – Evitar axiomas de quiesciência
b) *Inferência* – evitar cópias repetidas para manter o estado

Problema da Qualificação: descrições das acções reais obrigam à consideração de uma série de casos excepcionas – o que acontece se o ouro estiver escorregadio ou pregado ao chão ou...

Problema da Ramificação: acções reais têm muitas consequências secundárias – o que acontece à poeira em cima do ouro...

Resolução do problema da quiescência

Axiomas de estado sucessor resolvem o problema da quiescência representacional.

Cada axioma é "acerca" de um **predicado** (não da acção individualmente):

Acção é possível $\Rightarrow$
(P verdade a seguir $\Leftrightarrow$ (uma acção tornou P verdadeiro
$\vee$ P já verdadeiro e nenhuma acção tornou P falso)]

Axiomas de estado sucessor para o Wumpus

Predicado $Holding/2$:

$$Poss(a, s) \Rightarrow$$
$$(Holding(g, Result(a, s)) \Leftrightarrow (a = Grab(g) \vee Holding(g, s) \wedge a \neq Release(g)))$$

Predicado $At/3$:

$$Poss(a, s) \Rightarrow$$
$$(At(Agent, y, Result(a, s)) \Leftrightarrow (a = Go(x, y) \vee At(Agent, y, s) \wedge a \neq Go(y, z)))$$

E o que acontece ao ouro?

Problema da ramificação para o Wumpus

É necessário dizer que aquilo que o agente transporta vai com ele, e se o agente não se mexer então tudo o que ele possuir também não se mexe.

$$\begin{aligned} & Poss(a,s) \Rightarrow \\ & (At(o,y,Result(a,s)) \Leftrightarrow (a = Go(x,y) \wedge (o = Agent \vee Holding(o,s)) \vee \\ & \qquad At(o,y,s) \wedge \neg(\exists_z\ y \neq z \wedge a = Go(y,z) \wedge \\ & \qquad (o = Agent \vee Holding(o,s))))) \end{aligned}$$

Axiomas de nomes únicos

Come se faz uso da igualdade na modelação em cálculo de situações é preciso indicar que constantes e símbolos de funções distintos se referem a objectos distintos. Para cada par de constantes temos que dizer que são distintos **(Axiomas de nomes únicos).**

Ex: $Agent \neq Gold \wedge Agent \neq 1 \wedge Agent \neq 2 \wedge Gold \neq 1 \wedge Gold \neq 2 \wedge 1 \neq 2$

Muitos sistemas assumem implicitamente estes axiomas (ex: Prolog). Nesse caso estamos na presença da **assumpção de nomes únicos.**

Axiomas de acções únicas

Temos ainda de dizer que todas as acções são distintas. Para cada par de nomes de acções A, B, é necessário indicar ***A(x_1 ; … ; x_n) ≠ B (y_1 ; … ; y_n)***:

$$\begin{aligned} & \forall_{x_1,y_1,y_2}\ Grab(x_1) \neq Go(y_1,y_2) \\ & \forall_{x_1,y_1}\ Grab(x_1) \neq Release(y_1) \\ & \forall_{x_1,y_1,y_2}\ Release(x_1) \neq Go(y_1,y_2) \end{aligned}$$

É ainda necessário juntar para cada termo de acção:

$$\begin{aligned} & \forall_{x_1,y_1}\ Grab(x_1) = Grab(y_1) \equiv x_1 = y_1 \\ & \forall_{x_1,y_1}\ Release(x_1) = Release(y_1) \equiv x_1 = y_1 \\ & \forall_{x_1,x_2,y_1,y_2}\ Go(x_1,x_2) = Go(y_1,y_2) \equiv x_1 = y_1 \wedge x_2 = y_2 \end{aligned}$$

Estes axiomas são designados por **axiomas de acções únicas**. Com estes axiomas já podemos resolver o problema de planeamento pretendido.

Linguagem STRIPS

A utilização do cálculo de situações para representar problemas de planeamento obriga a um conhecimento aprofundado das particularidades da lógica, não sendo fácil para o utilizador. A aproximação **STRIPS** (STanford Research Institute Problem Solver) combina a procura em espaços de estados com as representações lógicas.

- ✓ **Representação dos estados:** o mundo é representado logicamente por intermédio de conjunções de literais positivos concretos (sem variáveis e símbolos de função). Adopta-se a Hipótese do Mundo Fechado (CWA).
- ✓ **Objectivos:** são representados novamente por um conjunto de literais positivos.
- ✓ **Acções:** são representadas por **esquemas de acção**, constituídos por um **nome e lista** de parâmetros (**ex:** *Go(x;y)*), **pré-condições** (literais positivos), e os **efeitos**, que tanto podem ser positivos ou negativos.

Operadores STRIPS

Descrição organizada de acções, linguagem restrita

$$\begin{array}{l}\text{Acção: } Buy(x)\\ \text{Precondição: } At(p), Sells(p,x)\\ \text{Efeito: } Have(x)\end{array}$$

$$\begin{array}{l}\text{Acção: } Go(x)\\ \text{Precondição: } At(y)\\ \text{Efeito: } At(x), -At(y)\end{array}$$

Nota: abstrai de muitos detalhes importantes

Linguagem restrita => algoritmo "eficiente"

Pré-condição: conjunção de literais positivos

Efeito: conjunção de literais

Um operador STRIPS é uma acção esquemática que deve ser instanciada de maneira a ficar proposicionalizada.

Tradução para Cálculo de Situações

Um conjunto de operadores STRIPS pode ser traduzido para um conjunto de axiomas de estado-sucessor. Para o exemplo anterior temos:

$$\forall_{a,s,x,p}\ Have(x, Result(a,s)) \equiv [\ a = Buy(x) \wedge At(p,s) \wedge Sells(p,x,s) \vee Have(x,s)\ ]$$

$$\forall_{a,s,x,y}\ At(x, Result(a,s)) \equiv [\ a = Go(x) \vee At(x,s) \wedge a \neq Go(y)\ ]$$

Princípios da Linguagem STRIPS

- ✓ **Estados** são conjuntos de **literais positivos**
- ✓ Os **literais** que não ocorrem num **estado** assumem-se falsos (Princípio do Mundo Fechado)
- ✓ Um **efeito positivo** adiciona o literal ao estado. Um **efeito negativo** remove o literal complementar do estado. O resto mantém-se inalterado para o estado seguinte.

Exemplo

Considere-se o estado inicial *{em (fct) ; com (dinheiro) ; praia (caparica)}* e os operadores:

Acção: $ir(?X)$
Precondição: $em(?Y)$
Efeito: $em(?X), -em(?Y)$

Acção: $banhoSol$
Precondição: $em(?Y), praia(?Y)$
Efeito: $com(bronze), com(sede), -sem(sede)$

Acção: $beberBijeca$
Precondição: $com(sede), com(dinheiro)$
Efeito: $-com(sede), -com(dinheiro), sem(sede)$

Objectivo: ficar com bronze e sem sede.

Planeamento em espaços de estados

Podem-se utilizar algoritmos de procura em espaço de estados para efectuar planeamento em dois sentidos:

- ✓ **Planeamento Progressivo:** partir do estado inicial, aplicar os operadores STRIPS até atingir um estado que torna os objectivos verdadeiros

Utilização imediata dos algoritmos de procura analisados anteriormente mas pode gerar muitos estados por utilização de acções irrelevantes.

- ✓ **Planeamento regressivo:** partir dos objectivos e utilizar os "operadores" ao contrário. Trabalha com estados incompletos.

Seja *G* o objectivo corrente. O novo objectivo corrente é obtido considerando as acções *A* tal que *A* tem um efeito positivo em *G* e nenhum efeito negativo em *G*. O novo objectivo é obtido:

- ✓ Removendo todo o efeito positivo de *A* que apareça em *G*
- ✓ Toda a pré-condição de *A* é adicionada a *G*, desde que não ocorra lá.

Exemplo: Planeamento Progressivo

Plano STRIPS: *[ir (caparica) ; banhoSol ; beberBijeca]*

Execução do plano e mudanças de estado:

$$\{em(fct), com(dinheiro), praia(caparica)\}$$
$$\Downarrow \quad ir(caparica)$$
$$\{em(caparica), com(dinheiro), praia(caparica)\}$$
$$\Downarrow \quad banhoSol$$
$$\{em(caparica), com(dinheiro), praia(caparica), com(bronze), com(sede)\}$$
$$\Downarrow \quad beberBijeca$$
$$\{em(caparica), praia(caparica), com(bronze), sem(sede)\}$$

Exemplo: planeamento regressivo

Plano STRIPS: *[ir (caparica) ; banhoSol ; beberBijeca]*

Execução do plano e mudanças de estado:

$$\{sem(sede), com(bronze)\}$$
$$\Uparrow \quad beberBijeca$$
$$\{com(dinheiro), com(sede), com(bronze), \}$$
$$\Uparrow \quad banhoSol$$
$$\{em(caparica), com(dinheiro), praia(caparica)\}$$
$$\Uparrow \quad ir(caparica)$$
$$\{em(fct), com(dinheiro), praia(caparica)\}$$

Heurísticas para planeamento em espaços de estados

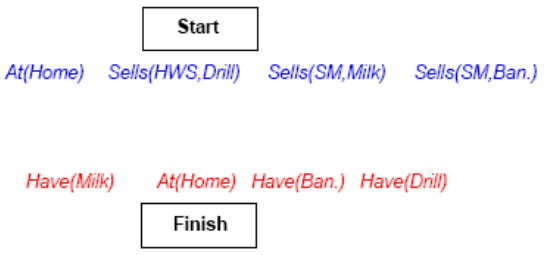
Quer o planeamento progressivo quer o planeamento regressivo requerem **heurísticas** apropriadas.

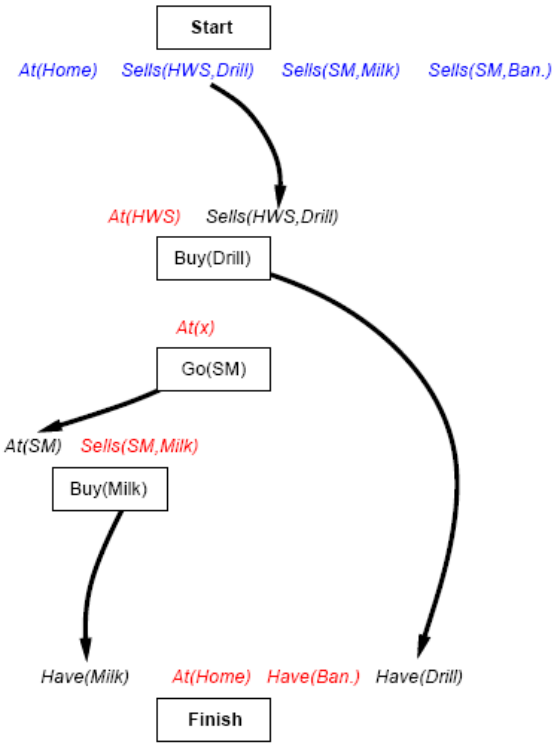
Utiliza-se a técnica de **relaxamento de problemas**. Uma heurística com bastante sucesso é a heurística da **lista de efeitos vazia.**

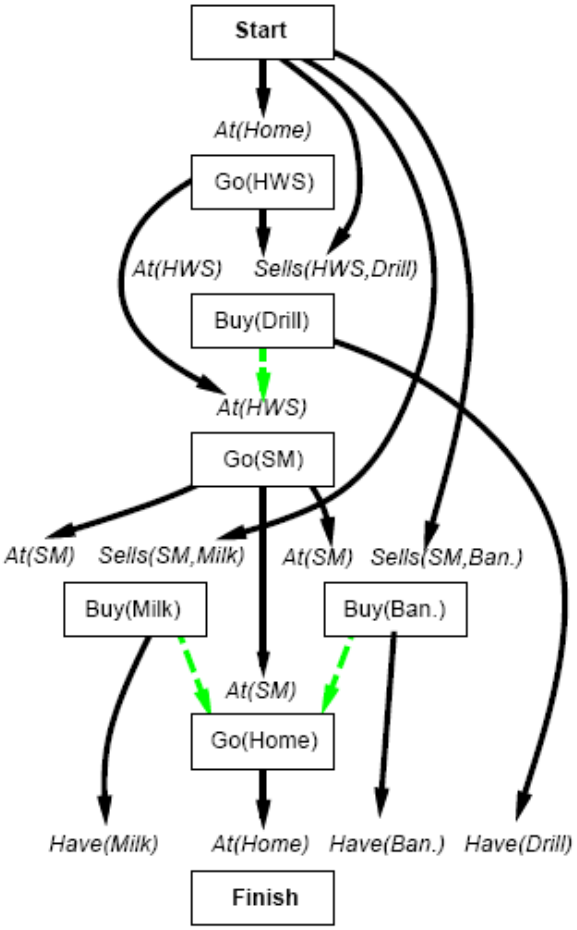
- ✓ Removem-se todos os efeitos negativos aos operadores e utiliza-se um algoritmo de planeamento (mais simples) para obter um número mínimo de acções necessárias

Exemplo

Considerando o exemplo do supermercado visto anteriormente:







Complexidade do planeamento

Se os operadores são fornecidos no INPUT, então:

- ✓ A existência de um plano descrito na linguagem STRIPS é um problema **EXSPACE-completo.**
- ✓ A existência de um plano de tamanho ≤ *n* descrito na linguagem STRIPS é um problema **NEXPTIME-completo** (mais fácil que o anterior)

Planos parcialmente ordenados

Colecção de passos **parcialmente ordenados** tal que:

passo Start descreve o estado inicial como os seus efeitos
passo Finish descreve o objectivo como as suas pré-condições
ligações causais do resultado de um passo para a pré-condição de outro
ordenação temporal entre pares de passos

Condição aberta é uma pré-condição de um passo que ainda não está ligada casualmente.

Um plano está completo se e só se toda a pré-condição foi alcançada.

Uma pré-condição encontra-se **alcançada** se e só se o é efeito de um passo anterior e nenhum **passo possivelmente interveniente** a contraria.

Processo de Planeamento

Operadores em planos parciais:

adiciona uma ligação de uma acção existente para uma condição aberta

adiciona um passo para garantir uma condição aberta

ordenar um passo relativamente a outro para eliminar possível conflitos

Estes planos evoluem gradualmente de planos incompletos/vagos para planos correctos e completos.

Retrocede se uma condição não é alcançável ou se um conflito não se puder resolver.

Algoritmo POP

```
function POP(initial, goal, operators) returns plan
    plan ← MAKE-INITIAL-PLAN(initial, goal)
    loop do
        if SOLUTION?(plan) then return plan
        S_need, c ← SELECT-SUBGOAL(plan)
        CHOOSE-OPERATOR(plan, operators, S_need, c)
        RESOLVE-THREATS(plan)
    end
```

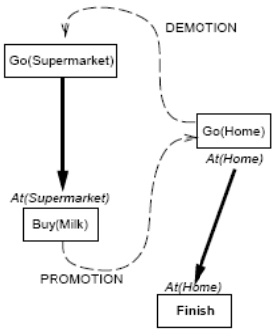
```
function SELECT-SUBGOAL(plan) returns S_need, c
    pick a plan step S_need from STEPS(plan)
        with a precondition c that has not been achieved
    return S_need, c
```

```
procedure CHOOSE-OPERATOR(plan, operators, S_need, c)
    choose a step S_add from operators or STEPS(plan) that has c as an effect
    if there is no such step then fail
    add the causal link S_add --c--> S_need to LINKS(plan)
    add the ordering constraint S_add ≺ S_need to ORDERINGS(plan)
    if S_add is a newly added step from operators then
        add S_add to STEPS(plan)
        add Start ≺ S_add ≺ Finish to ORDERINGS(plan)
```

```
procedure RESOLVE-THREATS(plan)
    for each S_threat that threatens a link S_i --c--> S_j in LINKS(plan) do
        choose either
            Demotion: Add S_threat ≺ S_i to ORDERINGS(plan)
            Promotion: Add S_j ≺ S_threat to ORDERINGS(plan)
        if not CONSISTENT(plan) then fail
    end
```

Ameaças e promoção/despromoção

Um **passo** é uma **ameaça** (conflituante) quando potencialmente destrói uma condição já alcançada por uma ligação causal. **Ex:** *Go(Home)* é uma ameaça a *At(Supermarket)*:



Despromoção: colocar antes de $Go(Supermarket)$

Promoção: colocar depois de $Buy(Milk)$

O teste de consistência assegura que não foram introduzidos ciclos nas relações de ordem temporal.

Propriedades do POP

Algoritmo não determinista: retrocede no caso de falha para pontos escolha:

- Escolha de S_{add} para alcançar Sn_{edd}
- Escolha de promoção ou despromoção
- Selecção de S_{need} irrevogável

POP é **fidedigno**, **completo** e **sistemático** (sem repetições).

Extensões para **disjunção**, **universais**, **negação** e **condicionais**.

Consegue-se eficiência com boas **heurísticas** derivadas da descrição do problema.

Particularmente bom para problemas com muitos sub-objectivos fracamente interligados.

Ameaças e variáveis

Será que um passo com o efeito: *At(x)* ameaça uma ligação para *At(Home)*?

SIM se *x* vier a unificar com *Home*

NÃO se *x* vier a unificar com outra constante

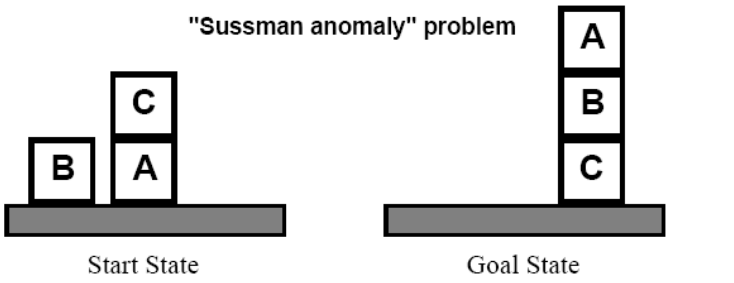
Como lidar com esta situação? Três situações possíveis:

1. Resolver logo com igualdade unificando *x* com algo diferente de *Home*, por exemplo *Supermercado*. Restrige demasiado

2. Resolver com desigualdade unificando *x* com ≠ *Home*. Mas o algoritmo de unificação Prolog não lida com desigualdades
3. Deixar para resolver mais tarde, caso *x* venha a unificar tarde com *Home*. Nessa situação é necessário
   - ✓ A representação dos planos precisa ser aumentada com lista de restrições da forma *z* ≠ X
   - ✓ Cada vez que efectuamos uma substituição temos que verificar que as desigualdades não contrariam a substituição

Exemplo: Mundo dos blocos

1.



Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

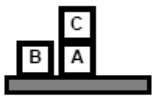
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

2.

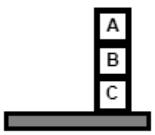
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

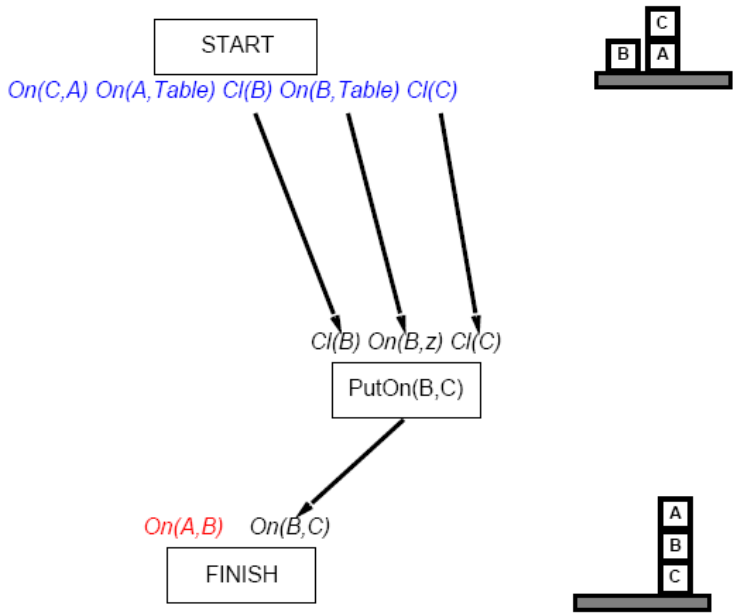


On(A,B) On(B,C)

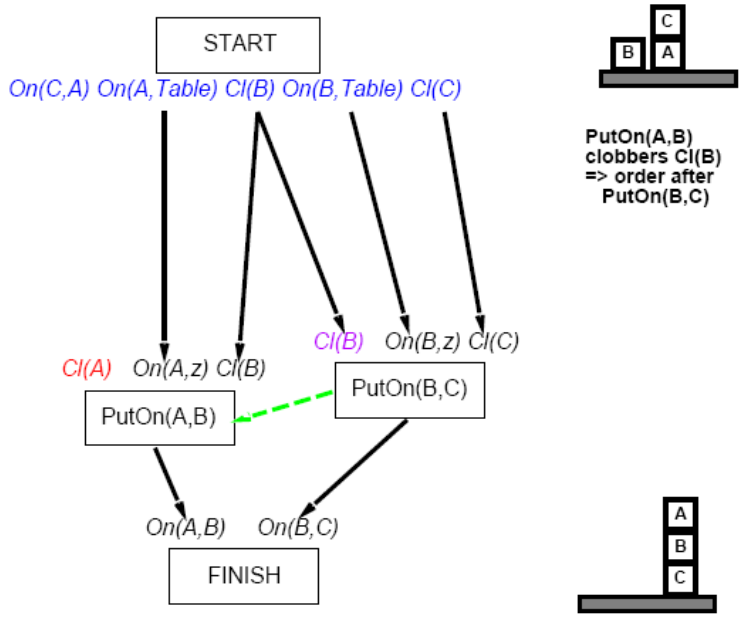
FINISH



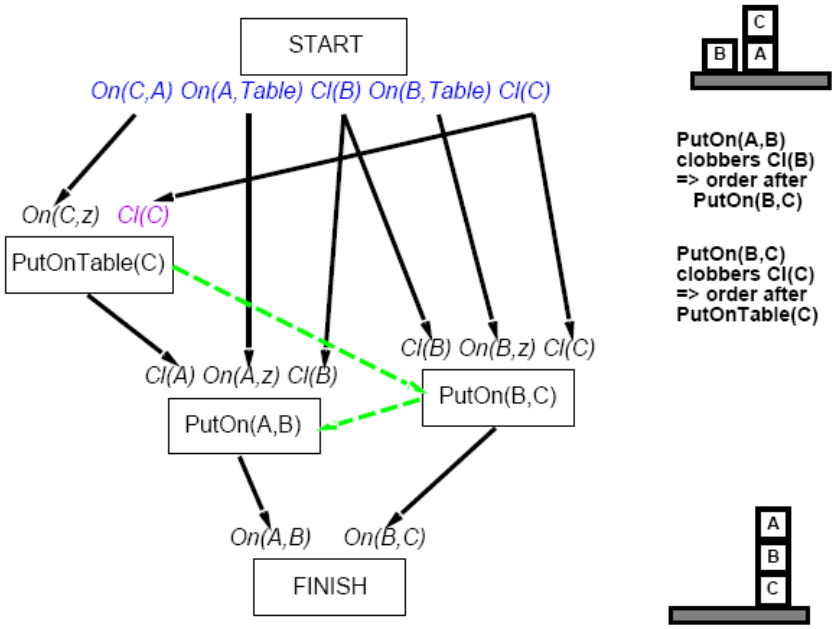
3.


START
On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)
Cl(B) On(B,z) Cl(C)
PutOn(B,C)
On(A,B) On(B,C)
FINISH
C
B
A
A
B
C

4.


START
On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)
Cl(A) On(A,z) Cl(B)
Cl(B) On(B,z) Cl(C)
PutOn(A,B)
PutOn(B,C)
On(A,B) On(B,C)
FINISH
C
B
A
PutOn(A,B)
clobbers Cl(B)
=> order after
PutOn(B,C)
A
B
C

5.



Casa & Descasa

A agência de matrimónios Casa&Descasa decidiu investir num sistema de Inteligência Artificial para melhorar o atendimento aos seus clientes. O sistema mantém informação sobre os homens e mulheres registados no sistema, os casais (heterossexuais) e indivíduos solteiros. O objectivo do sistema é aconselhar casamentos e divórcios sugerindo ainda "falecimento" de pessoas. A poligamia não é permitida.

As acções são: casar, divorciar, matar homem casado e matar mulher casada.

Por exemplo, o sistema deverá ser capaz de indicar os passos que levam o indivíduo Aníba, homem e solteiro, a casar com Belarmin que inicialmente está casada com César.

- Existe um número infinito de planos?
- Porque é que não é possível poligamia?

Acção: $Casar(H, M)$
Precondição: $solteiro(H), homem(H), solteiro(M), mulher(M)$
Efeito: $casados(H, M), -solteiro(H), -solteiro(M)$

Acção: $Divorciar(H, M)$
Precondição: $casados(H, M)$
Efeito: $solteiro(H), solteiro(M), -casados(H, M)$

Acção: $Matar\_Homem\_Casado(H)$
Precondição: $casados(H, M)$
Efeito: $solteiro(M), -casados(H, M), -homem(H)$

Acção: $Matar\_Mulher\_Casada(H)$
Precondição: $casados(H, M)$
Efeito: $solteiro(H), -casados(H, M), -mulher(M)$

O Zé Tuga, com tanto campeonato Europeu de futebol, não tem dedicado muito tempo à sua Maria. Para acalmar os ânimos decidiu organizar um jantar a dois, em que ele põe a mesa. Dada a sua manifesta ignorância na organização destes eventos, pediu conselhos a Maria que o instruiu:

- Podes pôr uma toalha ou não, mas só o podes fazer se não estiver nada em cima da mesa.
- Para a mesa ficar arranjadinha só deves colocar os talheres e os copos depois dos pratos.
- A ordem de colocação dos talheres e dos copos é indiferente, mas só depois dos pratos.

Podem existir planos com acções repetidas?

Acção: PôrToalha
Precondição: $Vazia(Mesa)$
Efeito: $Toalha, -Vazia(Mesa)$

Acção: PôrPratos
Precondição:
Efeito: $Pratos, -Vazia(Mesa)$

Acção: PôrCopos
Precondição: $Pratos$
Efeito: $Copos, -Vazia(Mesa)$

Acção: PôrTalheres
Precondição: $Pratos$
Efeito: $Talheres, -Vazia(Mesa)$

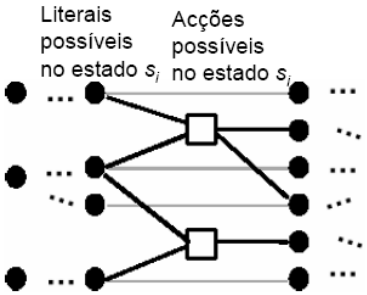
XII - Técnicas de Grafos

Procedure Graphplan:

- ✓ for k= 0, 1, 2,...
  - Expansão do grafo:
    - Criar um "**grafo de planeamento**" contendo k níveis
  - Verificar se o grafo de planeamento satisfaz uma condição necessária (mas insuficiente) para a existência de um plano

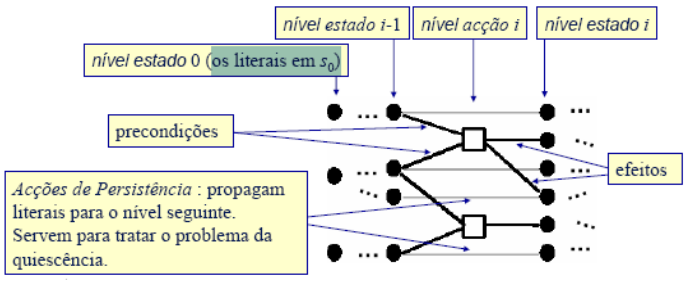
Problema Relaxado

  - Se satisfazer então:
    - Extrair Solução:
      - **Procura regressiva**, modificada para considerar apenas acções no grafo de planeamento
      - Caso se encontre uma solução, então **devolvê-la**



Grafo de Planeamento

- ✓ **Camadas alternadas** de literais básicos e acções básicas (sem vars)
- ✓ Todas as acções que possivelmente podem ocorrer em **cada instante de tempo**
- ✓ Todos os literais produzidos por essas **acções**



Exemplo (Dan Weld –U. of Washington):

Suponha-se que pretende preparar um jantar surpresa para o seu/sua cara-metade (que está a dormir)

S_0 = { garbage, cleanHands, quiet }

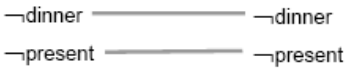
g = { dinner, present, ¬garbage }

Acção	Precondições	Efeitos
cook()	cleandHands	Dinner
wrap()	quiet	Present
carry()	nenhum	¬garbage, ¬cleanHands
dolly()	nenhum	¬garbage, ¬quiet

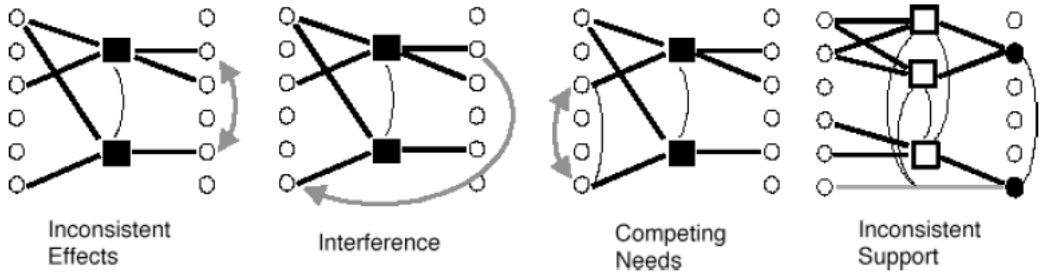
Também se adicionam **acções de persistência**: uma para cada literal *L*, com pré-condição *L* e efeito *L*.

- ✓ Nível estado 0: {todos os átomos em S_0} U {negação de todos os átomos não em S_0}
- ✓ Nível acção 1: { todas as acções cujas pré-condições são satisfeitas em S_0}
- ✓ Nível estado 1: {todos os efeitos para todas as acções do nível de acção 1}

Acções de Persistência:



Exclusão Mútua



Duas acções no mesmo o nível de acção são mutex se:

- ✓ **Efeitos inconsistentes:** o efeito de uma nega o efeito da outra
- ✓ **Interferência:** uma remove a pré-condição da outra
- ✓ **Necessidades em competição**: têm pré-condições mutuamente exclusivas

Caso contrário, não interferem uma com a outra:

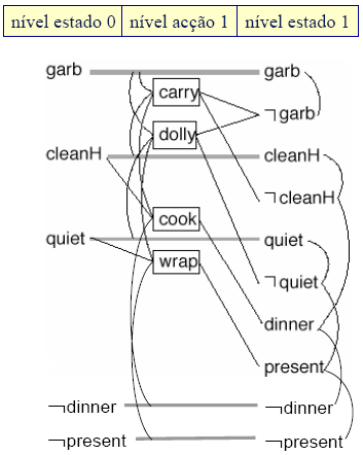
- ✓ Ambas podem aparecer no plano solução

Dois literais no mesmo nível estado são mutex se:

- ✓ **Suporte inconsistente:** um é a negação do outro, ou todas as maneiras de os alcançar são mutex

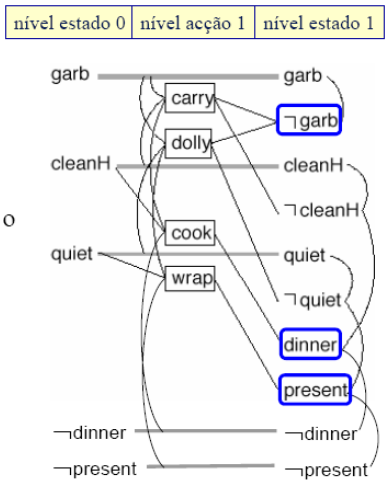
Exemplo (continuação):

- ✓ Aumentar o grafo para indicar **mutexes**
- ✓ **Carry** é o mutex com acção de persistência para *garbage* (efeitos inconsistentes)
- ✓ **Dolly** é um mutex com *wrap* (interferência)
- ✓ **Quiet** é mutex com *present* (suporte inconsistente)
- ✓ Quer *cook* quer *wrap* é mutex com uma acção de persistência



- ✓ Verificar se existe um plano possível
- ✓ O objectivo é {¬garbage, dinner, present}
- ✓ Repare-se que:
  - Todos são possíveis em S_1
  - Nenhum é mutex com qualquer outro
- ✓ Logo há hipóteses de um plano existir
- ✓ Tentar encontrá-lo
  - **Extracção da solução**

o



Extracção da Solução

procedure Solution-extraction(g^1,j^2)

se j = 0 então devolver solução

para cada literal *l* em *g*

escolher não deterministicamente uma acção[3] a usar no estado S_{j-1} para alcançar *l*.

se qualquer par de acções é mutex

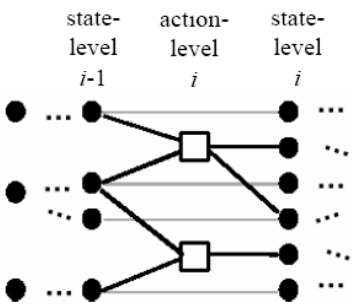
então retroceder

g' := {as pré-condições das acções escolhidas}

Solution-extraction(*g'*,*j* – 1)

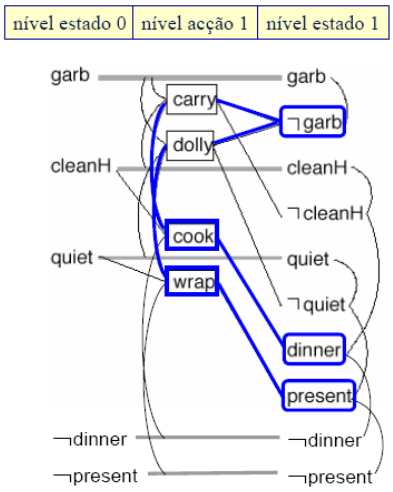
End Solution-extraction

[1] Os objectivos que estamos a tentar atingir
[2] O nível do estado s_j
[3] Uma acção real ou de persistência

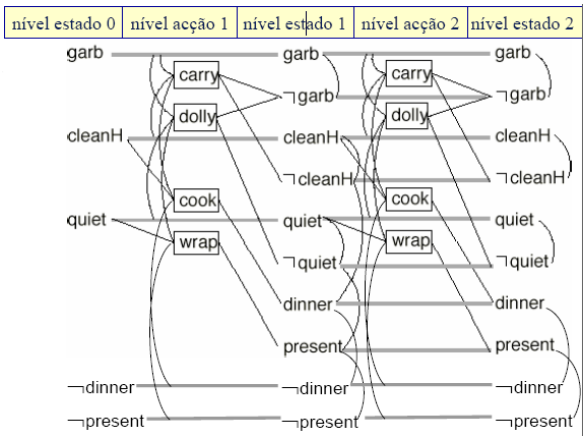


Exemplo (continuação)

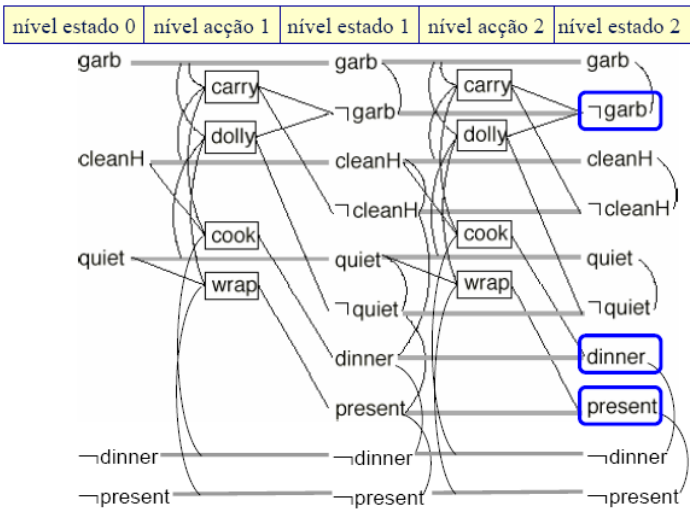
- ✓ Dois conjuntos de acções para os objectivos no nível estado 1
- ✓ Nenhum serve: ambos os conjuntos contêm acções que são mutex



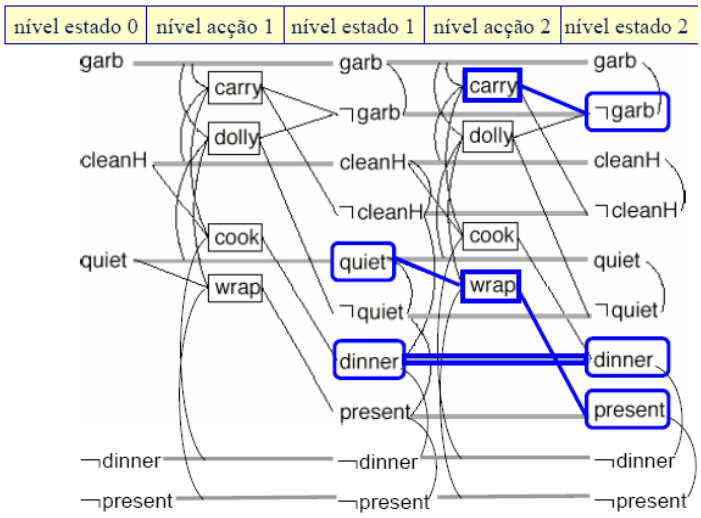
- ✓ Iterar e efectuar uma expansão do grafo adicional
- ✓ Gerar outro nível de acção e outro nível de estado



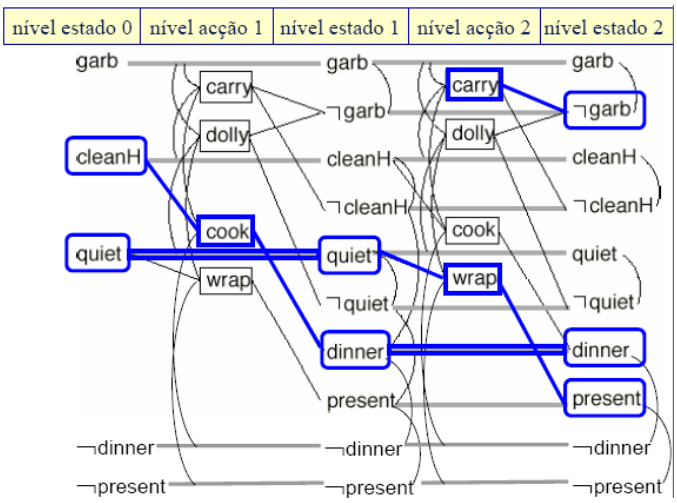
- ✓ Extracção da solução
- ✓ 12 combinações:
  - Três maneiras para ¬*garb*
  - Duas maneiras para *dinner*
  - Duas maneiras para *present*



- ✓ Algumas combinações parecem bem no nível 2
- ✓ Assinala-se uma delas:



- ✓ Chamar Solution-Extraction recursivamente no nível 2
- ✓ Tem sucesso
- ✓ Solução cujo **comprimento paralelo** é 2



Comparação com POP

Vantagem: a procura regressiva do GraphPlan – que é a parte difícil – só olha para as acções no grafo de planeamento. Espaço de procura mais pequeno do que o do POP, logo mais rápido.

Desvantagem: para gerar o grafo de planeamento, GraphPlan cria um número enorme de literais básicos (muitos deles podem ser irrelevantes)

Pode-se minimizar este problema (mas não eliminar) atribuindo tipos de dados a variáveis e constantes. Só se instanciam variáveis com termos do mesmo tipo de dados.

Heurísticas para o POP

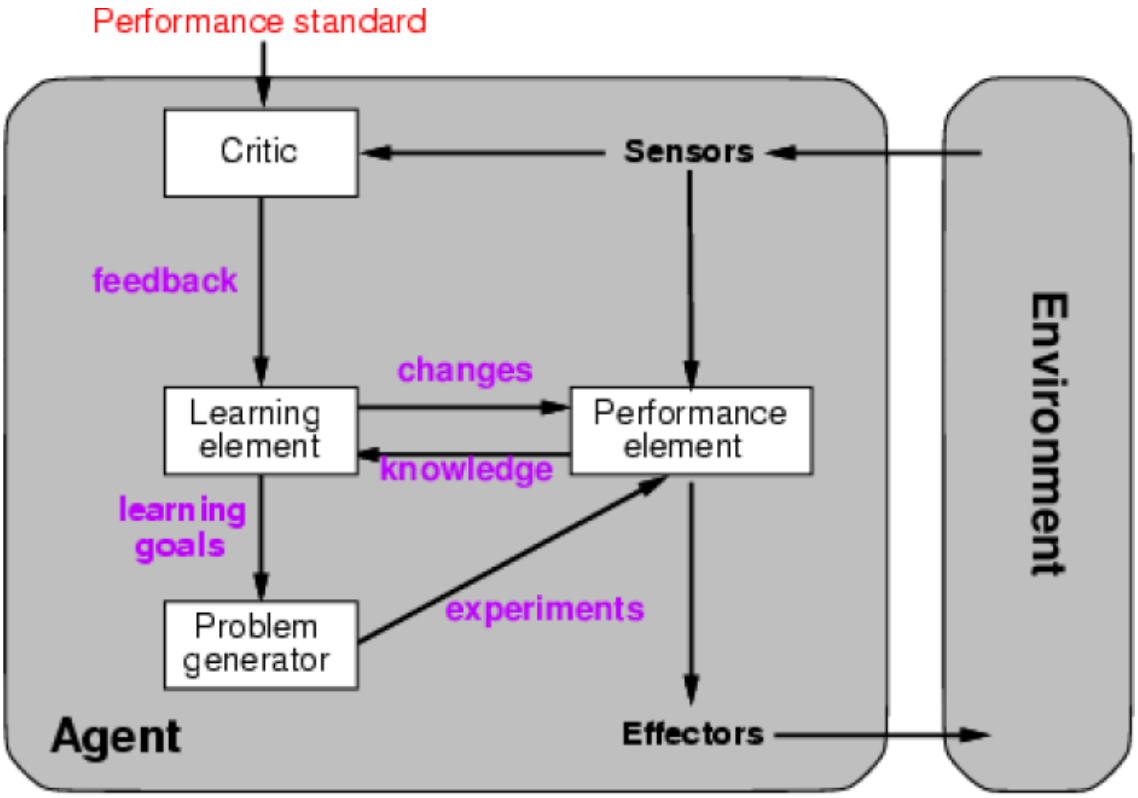
O grafo de planeamento pode ser gerado em tempo polinomial e termina sempre após um número finito de passos. Critério de paragem: quando dois níveis são idênticos.

Pode-se utilizar o grafo de planeamento para obter heurísticas para o POP.

Constrói-se o grafo de planeamento até obter um nível estado em que todos os objectivos sejam alcançados e não sejam mutex dois a dois.

A heurística assim obtida é admissível e pode ser utilizada conjuntamente com o POP.

XIII - Aprender a partir de exemplos



- ✓ O desenho do elemento de aprendizagem é ditado por
  - Quais as componentes do elemento de desempenho devem ser prendidas
  - Qual o feedback disponível para aprender esses componentes
  - Qual é a representação utilizada pelos componentes
- ✓ Tipo de feedback (Aprendizagem):
  - Supervisionada: respostas correctas para cada exemplo
  - Não Supervisionada: respostas correctas não são dadas
  - Por reforço: recompensas ocasionais

Paradigmas de Aprendizagem

- ✓ Computacional
  - – Empírica
  - – Casos
  - – Analítica
- ✓ Conexionista
- ✓ Biológica

Aprendizagem Indutiva

Aprender uma função a partir de exemplos dados (supervisionada)

- ✓ *f* é uma função alvo
- ✓ Um exemplo é um par (*x*, *f(x)*)

Problema da Indução: encontrar uma hipótese *h no espaço de hipóteses* tal que $h \approx f$ dado um conjunto de treino de exemplos (encontrar uma hipótese h que generalize bem, ou seja, que prediga correctamente exemplos desconhecidos); *h* diz-se consistente quando concorda com *f* em todos os exemplos.

Se a função alvo que estivermos a aprender for discreta estamos na presença de um problema de classificação.
Em particular, se estivermos interessados em funções booleanas dizemos que estamos na presença de um problema de aprendizagem conceptual.

O caso de aprendizagem de funções contínuas é designado por regressão (interpolação quando se impõe consistência)

Assumem a Hipótese da Aprendizagem Indutiva: "*Uma hipótese h que aproxime bem a função alvo num conjunto de treino suficientemente grande, também aproximará bem a função alvo num conjunto de exemplos não observados previamente*"

Navalha de Ockham: preferir a hipótese mais simples consistente com os dados

Aprendizagem Conceptual

Dados

- ✓ Instâncias X descritas por atributos
- ✓ Espaço de hipóteses H
- ✓ Conceito alvo c: X -> {0,1}
- ✓ Exemplos de treino D positivos e negativos

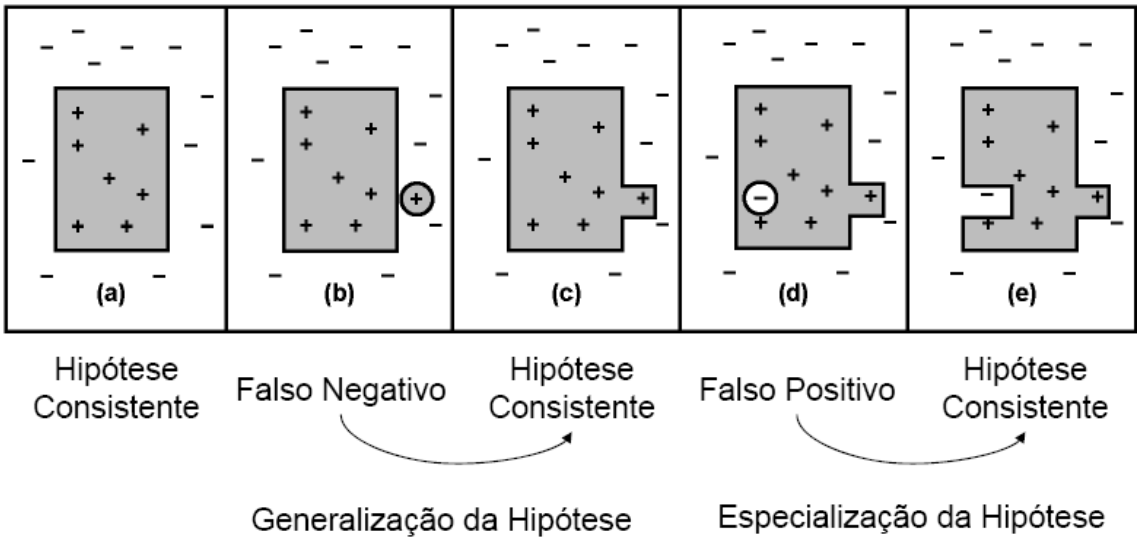
Determinar

- ✓ Uma hipótese *h* no espaço de hipóteses H tal que *h(x) = c(x)* para todo o *x*.

Exemplo:

Sky	AirTemp	Humidity	Wind	Water	Forecast	Sport?
Sunny	Warm	Normal	Strong	Warm	Same	1
Sunny	Warm	High	Strong	Warm	Same	1
Rainy	Cold	High	Strong	Warm	Change	0
Sunny	Warm	High	Strong	Cool	Change	1

Tratar Falsos Positivos e Negativos



Espaço de Hipóteses

Comecemos por considerar hipóteses conjuntivas com restrições simples nos valores de atributos:

'?' qualquer valor possível para o atributo

'*valor*' só *valor* é possível para o atributo

'_' nenhum valor é aceitável

Exemplos: h1=<?,Cold,High,?,?,?>; h2=<?,?,?,?,?,?>; h3=<_,_,_,_,_,_>

Satisfação de Hipóteses

Uma instância x satisfaz uma hipótese h sse h(x) = 1

Exemplo

A hipótese h=<Sunny,?,High,?,?,?> é satisfeita pela instância

<Sunny, Warm, High, Strong, Cool, Same>

e não é satisfeita pela instância

<Sunny, Warm, Normal, Strong, Cool, Same>

Ordenação parcial entre hipóteses

Uma hipótese hi é mais geral do que uma hipótese hj sse todas as instâncias que satisfazem hj também satisfazem hi (hi ≥hj)

Esta ordem parcial é induzida pela ordem parcial entre restrições simples nos valores do atributo.

Exemplo: h1=<Sunny,?,?,?,?,?>; h2=<Sunny,?,High,?,?,?>; h3=<Rainy,?,High,?,?,?>

Logo h1 ≥h2 (h1 é mais geral que h2). As hipóteses h1 e h3 são incomparáveis.

Espaço de hipóteses

Dado um conjunto de hipóteses qualquer, dizemos que h é maximamente específica, se não existir outra hipótese h' tal que h ≥h' e é maximamente geral, se não existir outra hipótese h'' tal que h''≥h

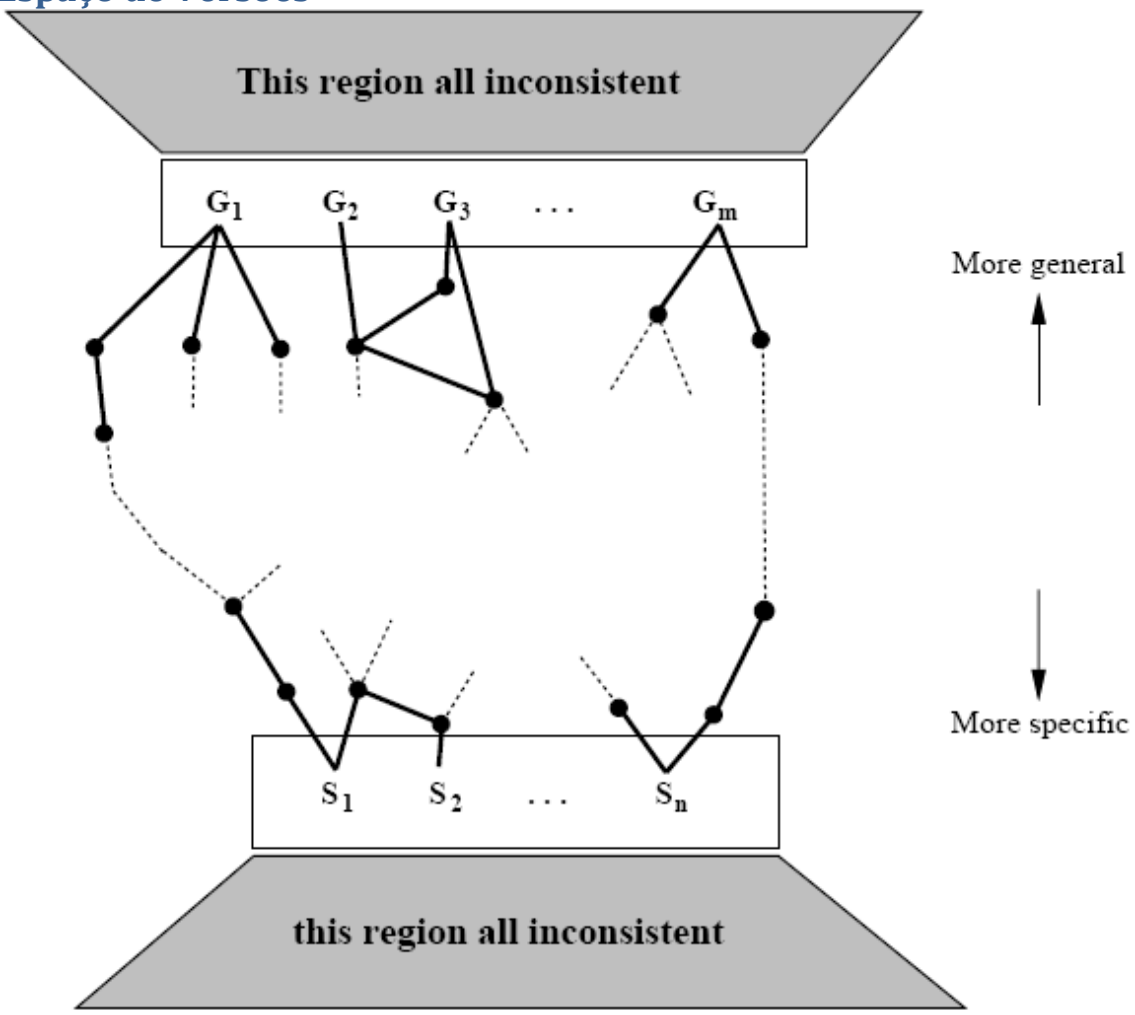
Eliminação de Candidatos

Algoritmo proposto por Mitchell que efectua a procura no espaço de versões (Version Space -VS), o conjunto de todas as hipóteses consistentes com os exemplos apresentados.
Uma hipótese *h* é consistente com um conjunto de treino *D* sse *h(x)=c(x)*para todo o exemplo de treino *<x,c(x)>* em *D*.
O algoritmo funciona mantendo duas fronteiras

G: a fronteira mais geral das hipóteses consistentes com os exemplos D

S: a fronteira mais específica das hipóteses consistentes com os exemplos D

Espaço de Versões



Evolução das Fronteiras

Seja *Si* uma hipótese em *S* e um novo exemplo

falso positivo para *Si*: eliminar *Si* de *S* pois não se pode especializar *Si*.

falso negativo para *Si: su*bstituir *Si* pelas suas generalizações imediatas, desde que mais específicas do que algum elemento de *G*

Seja *Gi* uma hipótese em *G* e um novo exemplo

falso negativo para *Gi:* eliminar *Gi* de *G* pois não se pode generalizar *Gi*.

falso positivo para *Gi*: *su*bstituir *Si* pelas suas especializações imediatas, desde que mais gerais do que algum elemento de *S*

Algoritmo de eliminação de candidatos (Mitchell)

S contém as hipóteses mais específicas de *H*
G contém as hipóteses mais gerais de *H*
Para cada exemplo de treino *d*, fazer

- Se *d* é um exemplo positivo
  - Remover de *G* qualquer hipótese inconsistente com *d*
  - Para cada hipótese *s* em S que não é consistente com *d*
    1. Retirar *s* de *S*
    2. Adicionar a *S* todas as generalizações minimais *h* de *s* tal que *h* é consistente com *d*, e algum membro de *G* é mais geral que *h*
    3. Retirar de *S* qualquer hipótese que seja mais geral que outra hipótese em *S*
- Se *d* é um exemplo negativo
  - Remover de *S* qualquer hipótese inconsistente com *d*
  - Para cada hipótese *g* em *G* que não é consistente com *d*
    1. Retirar *g* de *G*
    2. Adicionar a *G* todas as especializações minimais *h* de *g* tal que *h* é consistente com *d* e algum membro de *S* é mais específico que *h*
    3. Retirar de *G* qualquer hipótese que seja menos geral que outra hipótese em *G*

Prós

- ✓ É incremental
- ✓ Efectua o menor compromisso (tal como no POP)
- ✓ Converge para o conceito alvo pretendido se forem dados exemplos de treino suficientes (pelo menos log2|VS|).

Contras

- ✓ Se existir ruído ou o domínio não contiver atributos suficientes para a classificação exacta, o espaço de versões colapsa (um dos conjuntos S ou G fica vazio)
- ✓ O algoritmo com o espaço de hipóteses apresentado não permite aprender conceitos disjuntivos.
- ✓ Caso seja permitida disjunção ilimitada no espaço de hipóteses, então o algoritmo só conseguirá classificar os exemplos dados (não generaliza).
- ✓ Para alguns espaços de hipóteses o número de elementos de S e de G pode crescer exponencialmente.

Aprender árvores de decisão

Exemplos descritos por valores de atributos (Booleanos, discretos, contínuos)
Classificação dos exemplos é positiva (T) ou negativa (F)

Exemplo: situações em que espero/não esperam por uma mesa:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

Árvores de decisão podem expressar qualquer função dos seus atributos. Para as funções Booleanas, linha na tabela de verdade → caminho até à folha. Obviamente, existe uma árvore consistente para qualquer conjunto de treino tendo um caminho para a folha para cada exemplo (a não ser que *f* seja não determinista em *x*) mas provavelmente não é generalizável para novos exemplos

Preferir encontrar árvores de decisão mais compactas!

Utilizações típicas de Árvores de decisão

Exemplos são representados por pares atributos-valor.

A função alvo tem valores de saída discretos (existem extensões para lidar funções contínuas, mas são menos utilizadas).

Descrições disjuntivas podem ser necessárias: O conjunto de treino pode conter erros ou alguns atributos sem valor

Indução de Árvores de Decisão

Objectivo: encontrar uma árvore pequena consistente com os exemplos de treino
Ideia: escolher (recursivamente) o atributo "mais significativo" como raiz da sub(árvore)
Casos a tratar: Se existirem exemplos positivos e negativos, escolher o atributo mais significativo.

Se todos os exemplos remanescentes são positivos ou negativos, devolvemos Yes/No, respectivamente.

Se não existirem mais valores, devolve-se um valor por omissão (o que tiver mais ocorrências no nó pai – maioria).

Se não existirem mais atributos, temos um problema…Utiliza-se o valor maioritário no conjunto dos exemplos.

Algoritmo DTL (ou ID3)

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value v_i of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← DTL(examples_i, attributes − best, MODE(examples))
            add a branch to tree with label v_i and subtree subtree
        return tree
```

Escolha de um atributo: um bom atributo divide os exemplos em subconjuntos que (idealmente) são “todos positivos" ou “todos negativos"

Implementar *Choose-Attribute* no algoritmo DTL?
Conteúdo de Informação (Entropia): I(P(v1), …, P(vn)) = Σi=1-P(vi) log2P(vi)
Para um conjunto de treino contendo *p* exemplos positivos e *n* exemplos negativos:

$$I\left(\frac{p}{p+n},\frac{n}{p+n}\right)=-\frac{p}{p+n}\log_2\frac{p}{p+n}-\frac{n}{p+n}\log_2\frac{n}{p+n}$$

Ganho de informação:
Um atributo *A* escolhido divide o conjunto de treino *E* em subconjuntos *E1*, …, *Ev* de acordo com os valores que *A* toma, em que *A* tem *v* valores distintos

$$remainder(A)=\sum_{i=1}^{v}\frac{p_i+n_i}{p+n}I\left(\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}\right)$$

Ganho de Informação (IG) ou redução de entropia no atributo de teste:

$$IG(A)=I\left(\frac{p}{p+n},\frac{n}{p+n}\right)-remainder(A)$$

Escolher o atributo com maior IG

Medida de desempenho

Como é que sabemos que $h \approx f$?
Usar teoremas da teoria da aprendizagem computacional/estatística
Aplicar *h* num novo conjunto de exemplos de teste
Curva de aprendizagem = % de testes correctos em função da dimensão do conjunto de treino

Aspectos Práticos de Utilização

Se existirem muitos atributos irrelevantes ou ruído é mais fácil encontrar uma hipótese exacta. Essa hipótese é totalmente espúria. Neste caso estamos na presença de sobreajustamento na árvore de decisão

Avaliar a Performance

1. Coleccionar um grande conjunto de dados
2. Dividir esses dados arbitrariamente em dois conjuntos disjuntos: conjunto de treino e conjunto de teste
3. Aplicar o algoritmo de aprendizagem ao conjunto de treino, gerando uma hipótese *h*.
4. Medir a percentagem de exemplos no conjunto de teste classificados correctamente por *h*.
5. Repetir os passos 1 a 4 para diferentes tamanhos dos conjuntos de treino e conjuntos de treino seleccionados aleatoriamente para cada tamanho.

Técnicas de Avaliação

Holdout validation: Escolhem-se aleatoriamente exemplos do conjunto de dados para formar o conjunto de teste (habitualmente menos de 1/3 dos valores são utilizados como conjunto de teste)

K-fold cross-validation: Particionam-se os dados em K conjuntos de dimensão idêntica. Escolhe-se um desses como conjunto de teste e os restantes como conjuntos de treino. Repete-se o processo para cada subconjunto K e efectua-se a média dos resultados.

Leave-one-out cross-validation: Deixa-se um exemplo de fora e treina-se com todos os restantes exemplos. Repete-se para cada exemplo (*K-fold cross-validation* com K igual ao número de exemplos…)

Conclusões:

- ✓ Aprendizagem essencial para lidar com ambientes desconhecidos
- ✓ Agente aprendiz = elemento de desempenho + elemento de aprendizagem.
- ✓ No caso da aprendizagem indutiva, o objectivo consiste em encontrar uma hipótese simples que é aproximadamente consistente com os exemplos de treino.
- ✓ Aprendizagem conceptual é um caso particular de aprendizagem indutiva onde se pretende aprender uma função booleana a partir de exemplos dados.
- ✓ O algoritmo de eliminação de candidatos mantém as fronteiras de hipóteses maximamente específicas e maximamente gerais.
- ✓ Aprendizagem de árvores de decisão utiliza o ganho de informação
- ✓ Desempenho do algoritmo de aprendizagem = precisão no(s) conjunto(s) de teste.

XIV - Incerteza

Comece-se com um conjunto Ω—o *espaço amostra*
$\omega \in \Omega$ é um ponto amostra/mundo possível/acontecimento atómico

Um acontecimento atómico é uma especificação completa do estado do mundo acerca do qual o agente tem incerteza. Os acontecimentos atómicos são mutuamente exclusivos e exaustivos (e.g. 6 lançamentos de um dado).

Um *espaço de probabilidade* ou *modelo de probabilidade* é um espaço amostra com uma atribuição $P(\omega)$ para todo o $\omega \in \Omega$ tal que
$0 \leq P(\omega) \leq 1$
$\Sigma_\omega P(\omega) = 1$
e.g., $P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6$.

Um *acontecimento* A é qualquer subconjunto de Ω

$$P(A) = \Sigma_{\{\omega \in A\}} P(\omega)$$

Independência Condicional

Sejam X, Y e Z três conjuntos de variáveis aleatórias.

Diz-se que X é condicionalmente independente de Y dado Z, quando para todo o acontecimento $Z = z$ com $P(Z = z) > 0$ e qualquer par de acontecimentos $X = x$ e $Y = y$ e z se tem:

$$P(X = x, Y = y, Z = z) = \frac{P(X = x, Z = z) \times P(Y = y, Z = z)}{P(Z = z)}$$

Regra de Bayes

Regra do produto $P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$

$$\Rightarrow \text{ Regra de Bayes } P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

ou na forma de distribuição

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

Útil para obter a probabilidade do diagnóstico dada a probabilidade causal:

$$P(Causa|Efeito) = \frac{P(Efeito|Causa)P(Causa)}{P(Efeito)}$$

A teoria das probabilidades é um formalismo rigoroso para lidar com conhecimento incerto. Distribuição de probabilidade conjunta especifica a probabilidade de todo o acontecimento atómico. As interrogações podem ser respondidas somando-se acontecimentos atómicos.

Para domínios não triviais, é essencial reduzir o tamanho da distribuição conjunta. A Independência e independência condicional fornecem as ferramentas.

XV - Redes Bayesianas

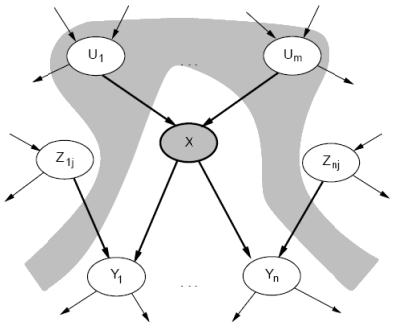
Notação gráfica simples para asseções de independência condicional e portanto uma especificação compacta para distribuições conjuntas totais

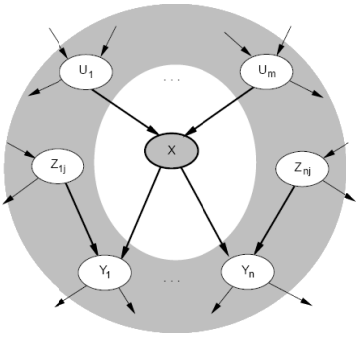
Sintaxe:

- ✓ um conjunto de nós, um por variável
- ✓ um grafo acíclico dirigido (arco influencia directamente")
- ✓ uma distribuição condicional para cada nó dados os seus pais: **$P(X_i | Parents(X_i))$**

No caso mais simples, distribuição condicional representada como uma tabela de probabilidade condicionada (CPT) especificando a distribuição de *Xi* para cada combinação dos valores dos pais.

Semântica Local: cada nó é condicionalmente independente dos seus não descendentes dado os seus pais.





Markov Blanket: Cada nó é condicionalmente independente de todos os outros dado o seu

Markov blanket: **pais** + **filhos** + **pais dos filho**

É necessário um método tal que uma série de asserções testadas localmente de independência condicional

1. Escolher uma ordenação das variáveis $X_1, \ldots, X_n$
2. For $i = 1$ to n
   adicionar X_i à rede
   seleccionar pais de $X_1, \ldots, X_{i-1}$ tal que
   $$\mathbf{P}(X_i | Parents(X_i)) = \mathbf{P}(X_i | X_1, \ldots, X_{i-1})$$

Esta escolha de pais garante a semântica global:

$$\begin{aligned} \mathbf{P}(X_1, \ldots, X_n) &= \prod_{i=1}^{n} \mathbf{P}(X_i | X_1, \ldots, X_{i-1}) \quad \text{(regra da cadeia)} \\ &= \prod_{i=1}^{n} \mathbf{P}(X_i | Parents(X_i)) \quad \text{(por construção)} \end{aligned}$$

garantam a semântica global pretendida.

XVI - Inferência em redes Bayesianas

Tarefas de inferência

Interrogações simples: calcular distribuição marginal a posteriori $P(X_i|E=e)$

Ex. P(NoGas|Gauge=empty, Lights=on, Starts=false)

Interrogações conjuntivas: $P(X_i,X_j|E=e) = P(X_i|E=e)P(X_j|X_i,E=e)$

Decisões óptimas: redes de decisão incluem informação de utilidade; inferência probabilística necessária para P(outcome|action, evidence).

Valor da informação: que evidência procurar a seguir?

Análise de sensibilidade: quais os valores de probabilidade mais críticos?

Explicação: por que é que preciso de um novo motor de arranque?

Algoritmo de enumeração

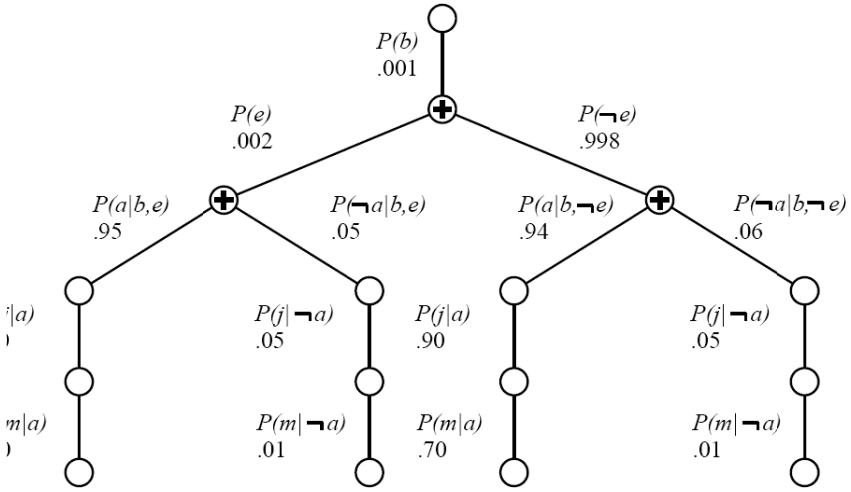
function ENUMERATION-ASK(X, **e**, bn) **returns** a distribution over X
  inputs: X, the query variable
    e, observed values for variables **E**
    bn, a Bayesian network with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$
  $\mathbf{Q}(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
    extend **e** with value x_i for X
    $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL(VARS[bn], **e**)
  return NORMALIZE($\mathbf{Q}(X)$)

---

function ENUMERATE-ALL($vars$, **e**) **returns** a real number
  if EMPTY?($vars$) **then return** 1.0
  $Y \leftarrow$ FIRST($vars$)
  if Y has value y in **e**
    then return $P(y \mid Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), **e**)
    else return $\Sigma_y\ P(y \mid Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_y$)
      where $\mathbf{e}_y$ is **e** extended with $Y = y$

Árvore de avaliação

Enumeração é ineficiente: cálculos repetidos



Inferência por eliminação de variáveis

Eliminação de variáveis: efectuar somatórios da direita para a esquerda, armazenando resultados intermédios (factores) para evitar recomputação.

$$
\begin{aligned}
&\mathbf{P}(B|j,m) \\
&\quad = \alpha \underbrace{\mathbf{P}(B)}_{B} \Sigma_e \underbrace{P(e)}_{E} \Sigma_a \underbrace{\mathbf{P}(a|B,e)}_{A} \underbrace{P(j|a)}_{J} \underbrace{P(m|a)}_{M} \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a \mathbf{P}(a|B,e) P(j|a) f_M(a) \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a \mathbf{P}(a|B,e) f_J(a) f_M(a) \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a f_A(a,b,e) f_J(a) f_M(a) \\
&\quad = \alpha \mathbf{P}(B) \Sigma_e P(e) f_{\bar{A}JM}(b,e) \text{ (soma-se } A) \\
&\quad = \alpha \mathbf{P}(B) f_{\bar{E}\bar{A}JM}(b) \text{ (soma-se } E) \\
&\quad = \alpha f_B(b) \times f_{\bar{E}\bar{A}JM}(b)
\end{aligned}
$$

Algoritmo da eliminação de variáveis

```
function ELIMINATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, evidence specified as an event
          bn, a belief network specifying joint distribution P(X_1, ..., X_n)
  factors ← [ ]; vars ← REVERSE(VARS[bn])
  for each var in vars do
      factors ← [MAKE-FACTOR(var, e)|factors]
      if var is a hidden variable then factors ← SUM-OUT(var, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Variáveis irrelevantes: *Y* é irrelevante a não ser que *Y* ∈ *Ancestor s*({*X*} ∪ *E*)

Fazendo X =JohnCalls, E={Burglary}, e Ancestors({X}UE) = {Alarm,Earthquake} conclui-se que M é irrelevante.

Inferência por simulação estocástica

Ideia básica:

- ✓ Efectuar *N* amostras de uma distribuição de amostragem *S*
- ✓ Calcular uma probabilidade a posteriori aproximada *P^*
- ✓ Mostrar que processo converge para a probabilidade real *P*

Métodos:

Amostragem a partir de uma rede vazia;
Amostragem por rejeição: rejeitar amostras que não estão de acordo com evidência;
Pesagem por Verosimilhança: utilizar evidência para pesar amostras;
Markov chain Monte Carlo (MCMC): amostrar a partir de um processo estocástico cuja distribuição estacionária é a probabilidade à posteriori real;

Amostragem por rejeição

(X|e) estimado das amostras que concordam com **e**

function REJECTION-SAMPLING(X, e, bn, N) **returns** an estimate of $P(X|\mathbf{e})$
  local variables: N, a vector of counts over X, initially zero

  for $j = 1$ to N **do**
    $\mathbf{x} \leftarrow$ PRIOR-SAMPLE(bn)
    if x is consistent with e **then**
      $\mathbf{N}[x] \leftarrow \mathbf{N}[x]+1$ where x is the value of X in x
  return NORMALIZE($\mathbf{N}[X]$)

Ex. estimar P(Rain|Sprinkler =true) utilizando 100 amostras
27 amostras têm *Sprinkler =true*
Destas, 8 têm *Rain=true* e 19 têm *Rain=false*.
(Rain|Sprinkler =true) = NORMALIZE((8; 19)) = (0:296; 0:704)

Semelhante aos procedimentos empíricos de estimativa.

Análise da amostragem por rejeição:

$$\begin{aligned}\hat{\mathbf{P}}(X|\mathbf{e}) &= \alpha \mathbf{N}_{PS}(X,\mathbf{e}) && \text{(algoritmo defn.)}\\ &= \mathbf{N}_{PS}(X,\mathbf{e})/N_{PS}(\mathbf{e}) && \text{(normalizado por } N_{PS}(\mathbf{e})\text{)}\\ &\approx \mathbf{P}(X,\mathbf{e})/P(\mathbf{e}) && \text{(propriedade de PRIORSAMPLE)}\\ &= \mathbf{P}(X|\mathbf{e}) && \text{(defn. de probabilidade condicional)}\end{aligned}$$

Logo amostragem por rejeição devolve estimativas consistentes da probabilidade à posteriori.

Markov Chain Monte Carlo (MCMC)

"Estado" da rede = atribuição corrente a todas as variáveis. Gerar o estado seguinte amostrando uma variável dado o seu Markov blanket- Altera-se uma variável de cada vez, por amostragem, mantendo a evidência.

function MCMC-ASK($X, \mathbf{e}, bn, N$) **returns** an estimate of $P(X|\mathbf{e})$
  local variables: $\mathbf{N}[X]$, a vector of counts over X, initially zero
    $\mathbf{Z}$, the nonevidence variables in bn
    $\mathbf{x}$, the current state of the network, initially copied from $\mathbf{e}$
  initialize $\mathbf{x}$ with random values for the variables in $\mathbf{Z}$
  for $j = 1$ to N **do**
    $\mathbf{N}[x] \leftarrow \mathbf{N}[x] + 1$ where x is the value of X in $\mathbf{x}$
    for each Z_i in $\mathbf{Z}$ **do**
      sample the value of Z_i in $\mathbf{x}$ from $\mathbf{P}(Z_i|MB(Z_i))$
      given the values of $MB(Z_i)$ in $\mathbf{x}$
  return NORMALIZE($\mathbf{N}[X]$)

Pode-se também escolher aleatoriamente a variável a amostrar.

Estimar P(Rain|Sprinkler = true,WetGrass = true)

Amostrar Cloudy ou Rain dado o seu Markov blanket, repetir. Contar p número de vezes que Rain é verdadeiro e falso nas amostras.

Ex. visita 100 estados
31 tem *Rain=true*, 69 tem *Rain=false*

$\hat{\mathbf{P}}$ (Rain|Sprinkler = true, WetGrass = true) = NORMALIZE((31, 69)) = (0,31; 0,69)

Teorema: cadeia aproxima-se da distribuição estacionária: a fracção de tempo gasto em cada espaço é exactamente proporcional à sua probabilidade à posteriori.

Inferência exacta por eliminação de variáveis:

tempo polinomial em polytrees, NP-difícil em grafos arbitrários
espaço = tempo, muito sensível à topologia

Inferência aproximada por PV, MCMC:

PV comporta-se mal quando existe muita evidência
PV, MCMC geralmente insensíveis à topologia;
Convergência pode ser muito lenta com probabilidades perto de 0 ou 1
Pode tratar combinações arbitrárias de variáveis discretas e contínuas

XVII - Procura com adversários

Adversário "imprevisível" - solução é uma **estratégia**, especificando uma jogada para cada resposta possível do oponente.

Limites temporais: pouco provável encontrar estratégia óptima, tem de se aproximar.

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Jogos Deterministas

Algoritmo Minimax: Estratégia perfeita para jogos *deterministas* e com informação perfeita. A ideia é escolher a jogada para posição com maior *valor minimax* (melhor recompensa alcançável com estratégia óptima do adversário.

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v
---
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v
---
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
    return v
```

Propriedades do minimax:

Completo Apenas se a árvore é finita (o Xadrez tem regras específicas para o efeito). NB uma estratégia finita pode existir mesmo em árvores infinitas!

Óptima? Sim, contra um adversário perfeito. Caso contrário?
Complexidade Temporal: *O*(*bm*)
Complexidade Espacial: *O*(*bm*) (exploração pelo melhor primeiro).

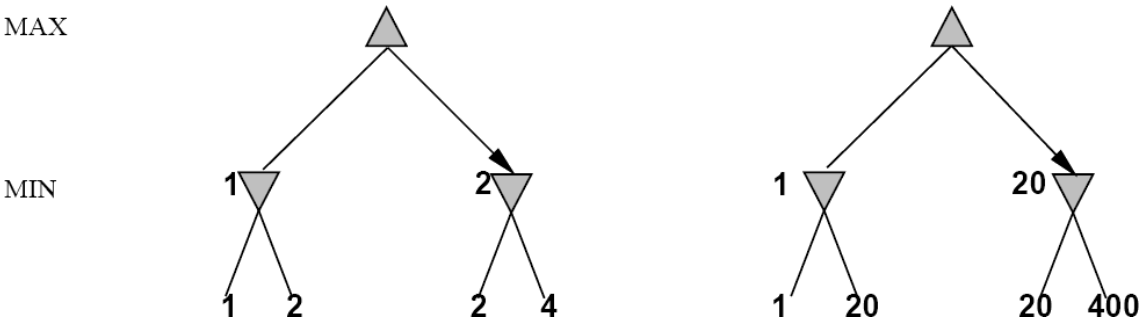
Suponha-se que temos 100 segundos, explorando 10^4 nós/segundo -> 10^6 nós por jogada

Aproximação Standard:

Teste de corte: limitar profundidade (possivelmente com procura quiescente)

Função de avaliação: estimativa da razoabilidade da posição A

Valores exactos não são importantes: O comportamento é preservado com qualquer transformação *monótona* de *EVAL*. Apenas a ordem interessa, recompensa em jogos deterministas comporta-se como uma função de *utilidade ordinal.*



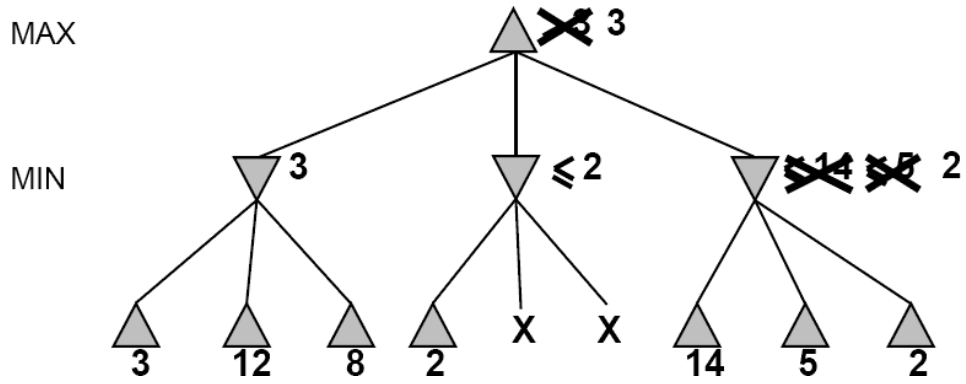
Como cortar um nó:

Limitar a profundidade a um número fixo (naïve). Efectuar aprofundamento progressivo enquanto houver tempo.

Efectuar procura quiescente. Só cortar nós em que o valor da função de avaliação estabilizou (sem grandes alterações previsíveis nas próximas jogadas)

Mais difícil de resolver é o problema da morte adiada (ou efeito horizonte). Uma jogada terrível pode ter os seus efeitos protelados pelo outro jogador, de maneira a ficarem invisíveis.

Exemplo de cote α-β



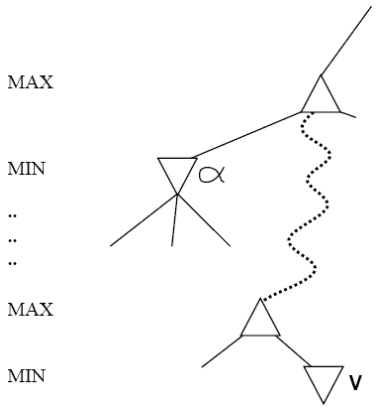
Propriedades do α-β:

Corte *não* afecta o resultado final. Uma boa ordenação das jogadas melhora o efeito do corte

Com "ordenação perfeita" complexidade temporal = *O*(*bm* = 2)

Duplica profundidade da procura
Pode facilmente atingir profundidade 8 e jogar bem Xadrez

Um exemplo simples do valor do raciocínio sobre quais as computações relevantes (uma forma de *meta-raciocínio*).



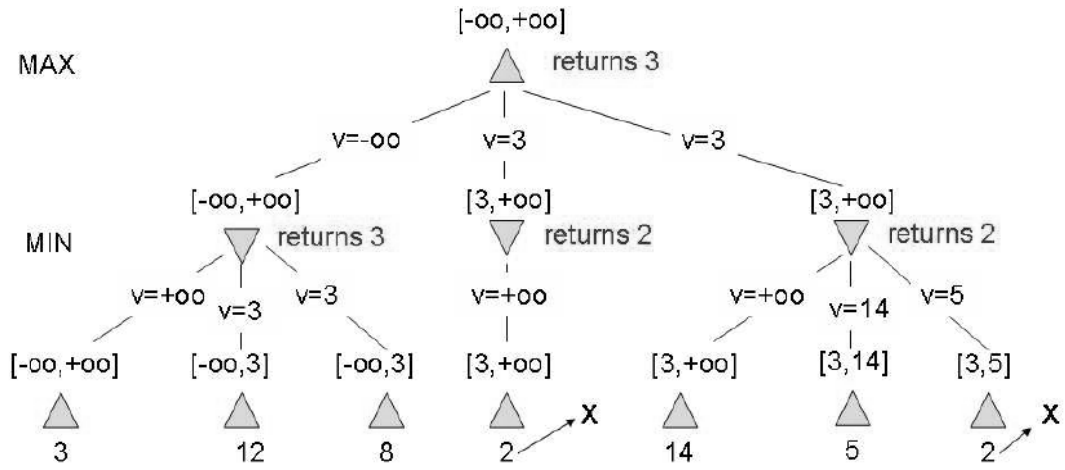
α *é* o melhor valor (para o jogador MAX) encontrado até ao momento. Se *v* é pior do que α, MAX evita-o; corta esse ramo. Define-se β similarmente para MIN.

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
    if CUTOFF-TEST(state) then return EVAL(state)
    v ← −∞
    for each a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
    if CUTOFF-TEST(state) then return EVAL(state)
    v ← +∞
    for each a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Execução do Algoritmo α-β:



Legenda:

$[\alpha, \beta]$	Valor dos parâmetros na invocação
v = u	Valor da variável v antes da chamada recursiva
X	Corte (após retorno da chamada recursiva)
returns u	valor MINIMAX de saída da chamada

Exemplos de jogos deterministas:

Damas: Chinook terminou com o reinado de 40 anos do campeão mundial Marion Tinsley em 1994. Utilizou uma base de dados de final de jogo definindo a estratégia perfeita para todas as posições com 8 ou menos peças no tabuleiro, num total de 443,748,401,247 posições.

Xadrez: Deep Blue derrotou o campeão mundial humano Gary Kasparov num encontro a 6 partidas em 1997. Deep Blue procura 200 milhões de posições por segundo, utiliza avaliação muito sofisticada, e recorres a métodos para estender algumas linhas de procura até 40 jogadas.

Othello: campeões humanos recusam-se a competir contra computadores, que são demasiado bons.

Go: campeões humanos recusam-se a competir contra computadores, que são péssimos jogadores. No go, $b > 300$, portanto a maioria dos programas recorrem a bases de conhecimento de padrões para sugerir jogadas plausíveis.

Jogos não deterministas

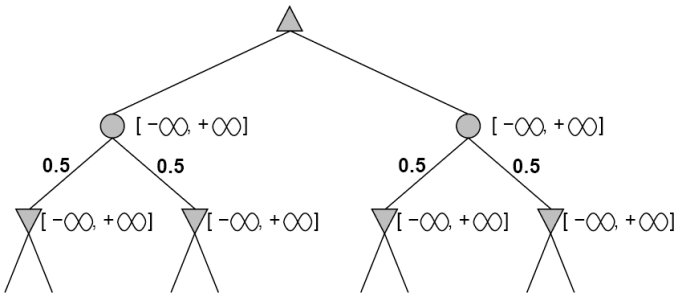
Algoritmo para jogos não deterministas:

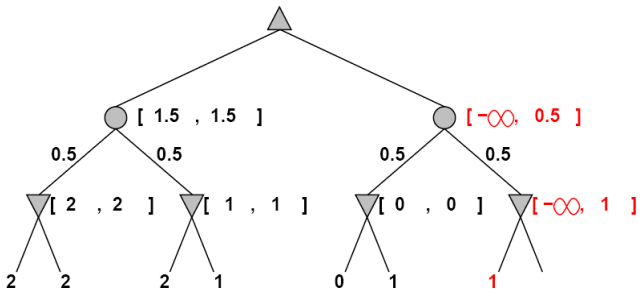
EXPECTIMINIMAX fornece estratégia óptima como o Minimax, excepto que se tem de ter em conta nós de sorte:

$$\text{EXPECTIMINIMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{se } n \text{ é um estado terminal} \\ \max_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{se } n \text{ é um nó MAX} \\ \min_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{se } n \text{ é um nó MIN} \\ \Sigma_{s \in Successors(n)} \text{P(s)} \times \text{EXPECTIMINIMAX}(s) & \text{se } n \text{ é um nó CHANCE} \end{cases}$$

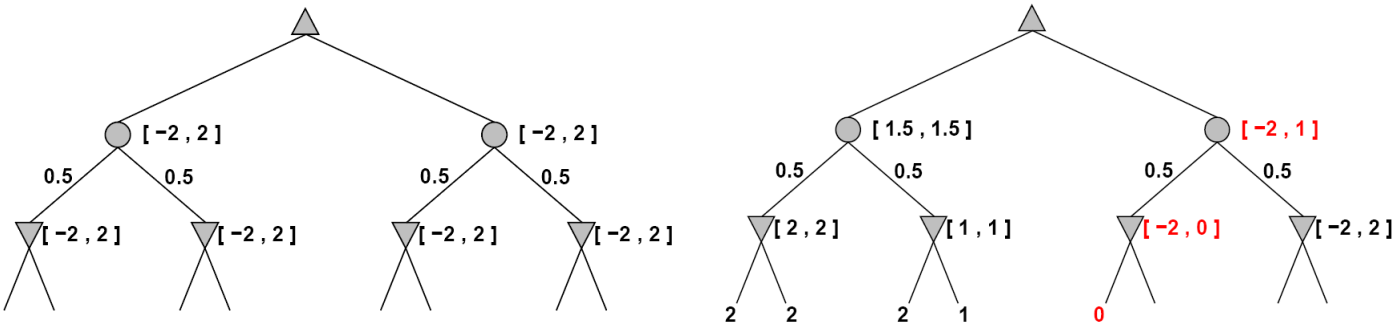
Corte em árvores de jogos não deterministas:

Uma adaptação do corte α-β é possível:

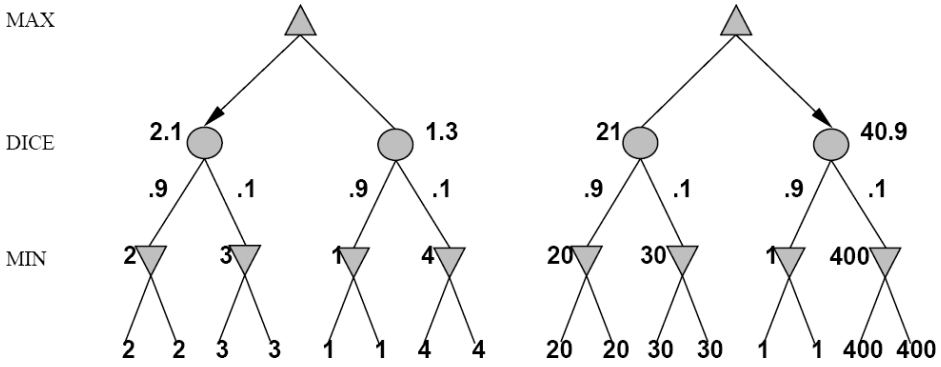




Pode-se cortar mais se os valores das folhas forem limitados:



Valores exactos são importantes: O comportamento é preservado apenas com transformações *lineares positivas* de EVAL. Logo EVAL deve ser proporcional à recompensa esperada.



Exemplo de jogos não deterministas: jogos de cartas, em que as cartas iniciais do adversário são desconhecidas. Tipicamente pode-se calcular a probabilidade de cada distribuição de cartas pelos jogadores. Aparentemente semelhante a um lançamento de dados no início do jogo.

Ideia: calcular o valor minimax de cada acção em cada mão, então escolher a acção com maior valor esperado de entre todas as mãos.

Caso especial: Se uma acção é óptima para todas as mãos então é óptima. O GIB, melhor programa de bridge actualmente, aproxima esta ideia:

- ✓ Gerando 100 mãos consistentes com a informação de apostas actuais
- ✓ Escolhe a acção que ganha mais vazas em média.

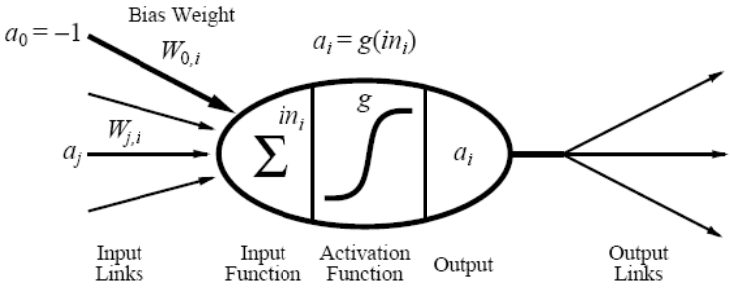
Os jogos ilustram vários aspectos importantes da IA, a perfeição é inatingível, tem de se aproximar; boa ideia pensar sobre o que se vai passar; a incerteza limita a atribuição de valores a estados.

XVIII - Redes Neuronais

Unidade de McCulloch-Pitts

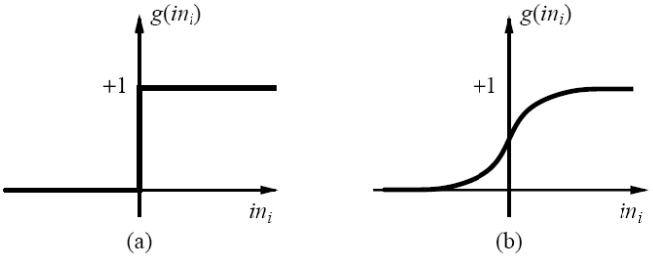
Saída é uma função linear "esmagada" das entradas:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$



Uma simplificação rude dos neurónios reais, mas os seus objectivos é obter conhecimento sobre o que conseguem fazer as redes de unidades simples.

Funções de activação



(a) é a função **degrau** ou função **limiar**

(b) é a função **sigmóide** $1/(1 + e^{-x})$

Alteração da **polarização** (bias weight), $W_{0,i}$ desloca a localização do **limiar**.

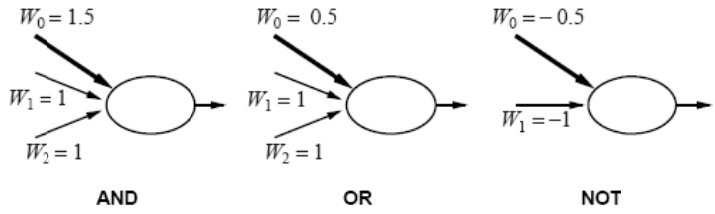
É importante que a função seja não-linear, caso contrário a saída reduz-se a uma combinação linear das entradas.

Os algoritmos de aprendizagem que estudaremos assumem que a função é diferenciável.

Outros exemplos:

- ✓ Tangente Hiperbólica (contra domínio [-1,1])
- ✓ Seno (contra-domínio [-1,1]
- ✓ Logarítmica (contra-domínio [0, +∞])

Implementação das funções lógicas



McCulloch e Pitts: qualquer função booleana pode ser implementada.

Topologia de redes

Redes alimentadas para a frente:

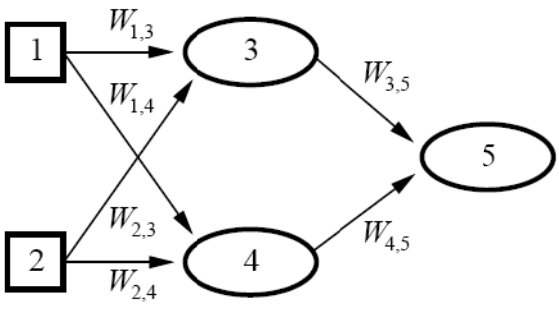
- ✓ Rede **monocamada** de perceptrões
- ✓ Rede **multicamada** de perceptrões

Redes alimentadas para a frente implementam funções, não tem estado interno.

- ✓ **Redes recorrentes:**
- ✓ Redes de **Hopfield** tem pesos simétricos ($W_{i,j} = W_{j,i}$)
  - g(x) = sign(x), $a_i = \pm 1$; **memórias holográficas associativas.**
- ✓ Máquinas de **Boltzmann** utilizam funções de activação estocásticas.

Nota: As redes recorrentes têm ciclos dirigidos com atrasos (delays). Tem estado interno interno (como flip-flops), podem oscilar etc.

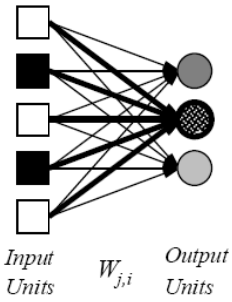
Exemplo de rede alimentada para a frente

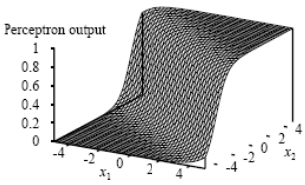


Rede alimentada para a frente = família parametrizada de funções não lineares:

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Ajustando os pesos altera-se a função: efectuar aprendizagem desta maneira.





As unidades operam separadamente – não existem **pesos partilhados**.

Ajustando os pesos altera-se a localização, orientação e inclinação do declive.

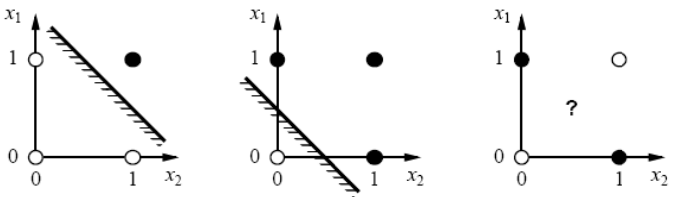
Expressividade dos perceptrões

Considere-se o perceptrão com a função de activação *g* = **degrau** (Rosenblatt, 1957, 1960).

Pode representar **AND**, **OR**, **NOT**, **maioria**, etc., mas não o XOR.

Representa um **separador linear** no espaço de entradas:

$\sum_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Regra Delta - Widrow e Hoff

Aprender por ajustamento dos pesos de forma a reduzir o **erro** no conjunto de exemplos de treino. Tratamos primeiro o caso de função de activação linear:

O erro quadrático para um exemplo com entrada *x* e o valor correcto *y* é:

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

Recorrer ao método do gradiente descendente para minimizar *E*:

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}(y - \Sigma_{j=0}^{n} W_j x_j)$$
$$= -Err \times x_j$$

Regra simples para actualização dos pesos:

$$W_j \leftarrow W_j + \eta \times Err \times x_j \quad (\eta \text{ é ritmo de aprendizagem})$$

Algoritmo de Aprendizagem do Perceptrão

function PERCEPTRON-LEARNING(*network*, *examples*, η) **returns** a perceptron hypothesis

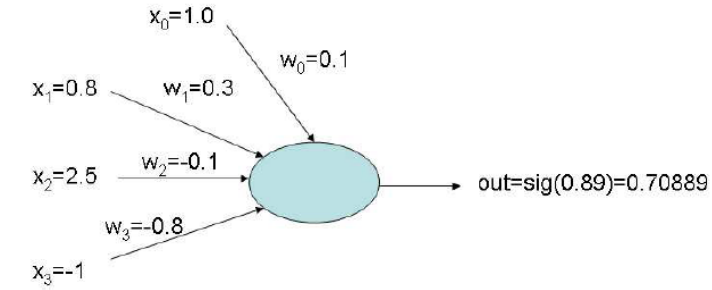
inputs: *network*, um perceptrão com pesos W_j, $J = 0, \ldots, n$ e função de activação g
examples, um conjunto de exemplos, com entrada $\mathbf{x} = x_1, \ldots, x_n$ e saída y
η, o ritmo de aprendizagem

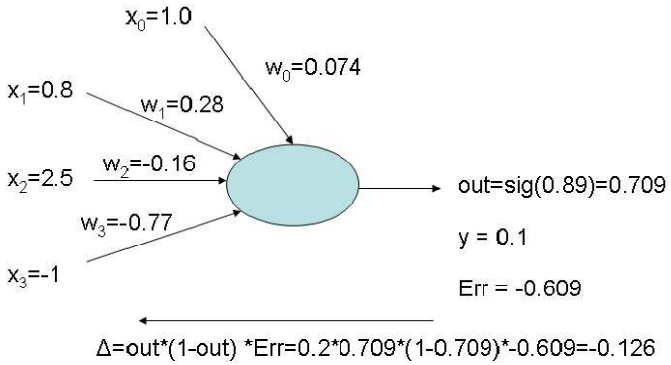
repeat
  for each e **in** *examples* **do**
    /* *Calcular o valor de saída para este exemplo* */
    $in \leftarrow \sum_{j=0}^{n} W_j\, x_j[e]$
    $out \leftarrow g(in)$
    /* *Calcular o erro* */
    $Err \leftarrow y[e] - out$
    /* *Actualizar os pesos das entradas* */
    for each entrada j do perceptrão **do**
      $W_j \leftarrow W_j + \eta \times Err \times g'(in) \times x_j[e]$
    end
  end
until se tenha atingindo um critério de paragem

Nota: com função de activação sigmóide a expressão de actualização de pesos é dada por $W_j \leftarrow W_j + \eta \times Err \times out \times (1 - out) \times x_j[e]$.

Com função limiar a expressão de actualização dos pesos é $W_j \leftarrow W_j + \eta \times Err \times x_j[e]$.

Exemplo de aplicação





Peso	Inicial	Entrada	Actualização(ΔW_i)	Peso Final
W_0	0.1	$x_0 = 1.0$	$\eta * \Delta * x_0 = 0.2 * -0.126 * 1.0 = -0.025$	0.074
W_1	0.3	$x_1 = 0.8$	$\eta * \Delta * x_1 = 0.2 * -0.126 * 0.8 = -0.02$	0.28
W_2	−0.1	$x_2 = 2.5$	$\eta * \Delta * x_2 = 0.2 * -0.126 * 2.5 = -0.06$	−0.16
W_3	−0.8	$x_3 = -1.0$	$\eta * \Delta * x_2 = 0.2 * -0.126 * -1.0 = 0.03$	−0.77

Perceptrão (regra delta generalizada)

Aprender por ajustamento dos pesos de forma a reduzir o **erro** no conjunto de exemplos de treino. Caso de funções de activação diferenciáveis:

Recorrer novamente ao método do gradiente descendente para minimizar *E*:

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right)$$
$$= -Err \times g'(in) \times x_j$$

Regras simples para actualização dos pesos:

$$W_j \leftarrow W_j + \eta \times Err \times g'(in) \times x_j$$

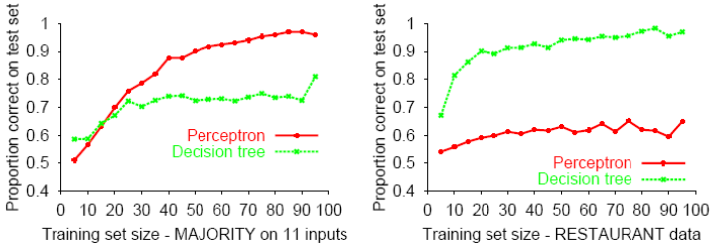
Ex: erro + ve => aumenta saída da rede

- ✓ Aumento de pesos com entradas com +ve, diminuir em entradas –ve

Nota: $sig'(x) = sig(x) \times (1 - sig(x))$

Aprendizagem do perceptrão (cont.)

A regra de aprendizagem do perceptrão converge para uma função consistente **para qualquer conjunto de dados linearmente separável.**

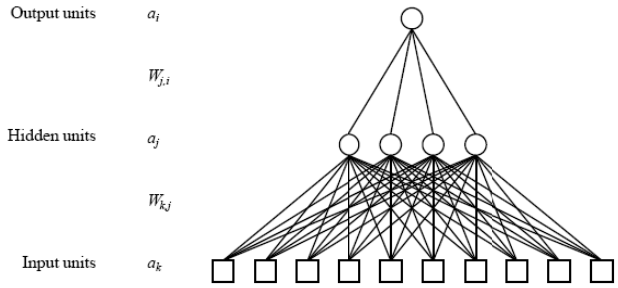


Perceptrão aprende função de maioria facilmente, indução de árvore de decisão é inútil.

Indução de árvore de decisão aprende função do restaurante, perceptrão não pode representá-la.

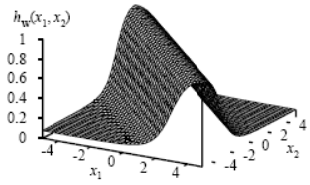
Perceptrões multicamada

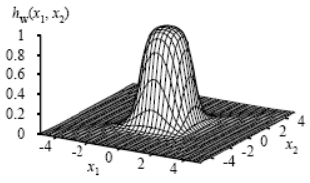
Normalmente as camadas são totalmente ligadas; número de **unidades escondidas** tipicamente escolhido manualmente.



Expressividade de redes multicamada

Todas as funções contínuas com 1 camada escondida, todas as funções com 2 camadas escondidas.





Combinar duas funções limiar opostas para construir uma crista;

Combinar duas cristas perpendiculares para construir um morro;

Juntar morros de vários tamanhos e localizações para obter qualquer superfície. A prova requer um número exponencial de unidades escolhidas.

Aprendizagem por retropropagação

Camada de saída: o mesmo para o perceptrão monocamada,

$$W_{j,i} \leftarrow W_{j,i} + \eta \times a_j \times \Delta_i$$

em que $\Delta_i = Err_i \times g'(in_i)$

Camada escondida: **retropropaga** o erro da camada de saída:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$

Regra de actualização para os pesos da camada escondida:

$$W_{k,j} \leftarrow W_{k,j} + \eta \times a_k \times \Delta_j \ .$$

Derivação da regra de retropropagação

O erro quadrático num único exemplo é definido como:

$$E = \frac{1}{2} \sum_i (y_i - a_i)^2 \ ,$$

em que a soma inclui todos os nós da camada de saída

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i) \frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i) \frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i) g'(in_i) \frac{\partial}{\partial W_{j,i}} \left(\sum_j W_{j,i} a_j \right) \\
&= -(y_i - a_i) g'(in_i) a_j = -a_j \Delta_i
\end{aligned}$$

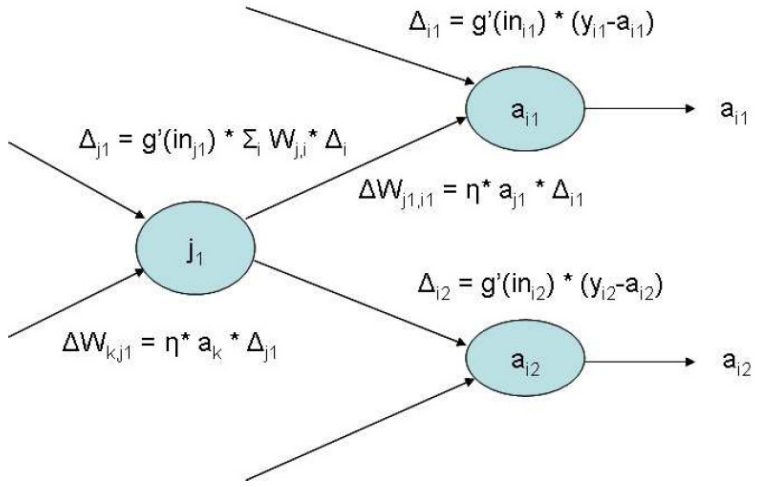
Derivação retropropagação outras camadas

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i) \frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i) \frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}} \left(\sum_j W_{j,i} a_j \right) \\
&= -\sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}} \left(\sum_k W_{k,j} a_k \right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}$$

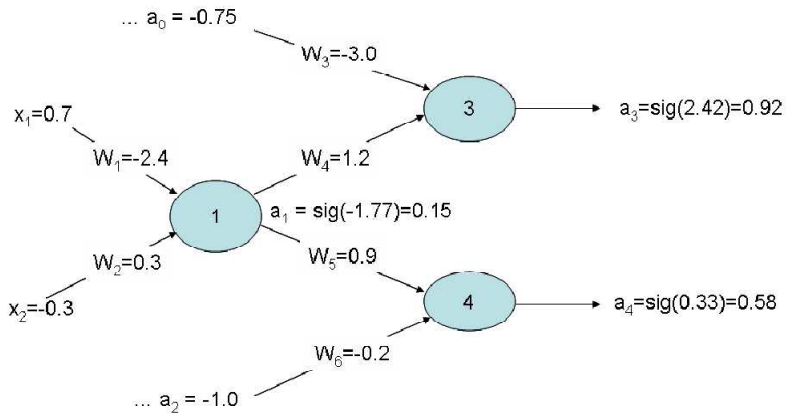
O algoritmo de retropropagação

```
function BACK-PROP-UPDATE(network, examples, α) returns a
network with modified weights
  inputs: network, a multilayer network
          examples, a set of input/output pairs
          η, the learning rate

  repeat
    for each e in examples do
      /* Compute the output for this example */
        O ← RUN-NETWORK(network, I^e)
      /* Compute the error and Δ for units in the output layer */
        Err^e ← T^e − O
      /* Update the weights leading to the output layer */
        W_{j,i} ← W_{j,i} + η × a_j × Err_i^e × g'(in_i)
      for each subsequent layer in network do
        /* Compute the error at each node */
          Δ_j ← g'(in_j) Σ_i W_{j,i} Δ_i
        /* Update the weights leading into the layer */
          W_{k,j} ← W_{k,j} + η × a_k × Δ_j
      end
    end
  until network has converged
```

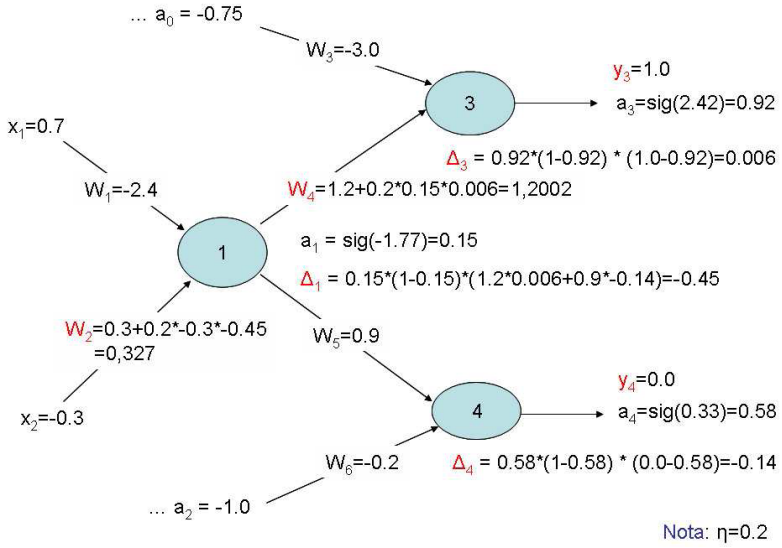
Esquema de funcionamento



Exemplo de aplicação (propagação)



Exemplo de aplicação (retropropagação)

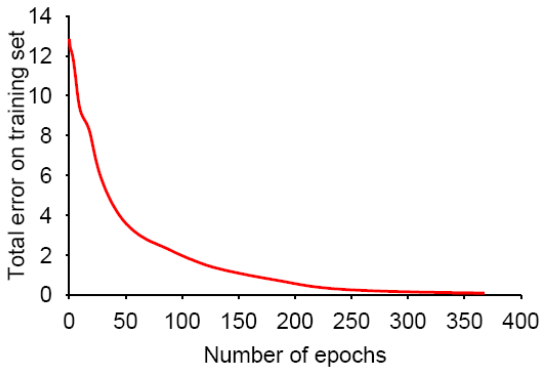


Só mostrada actualização dos pesos W_2 e W_4 a partir dos termos de erro Δ_1, Δ_3 e Δ_4. Os outros pesos são alterados de maneira semelhante.

Aprendizagem com Retropropagação (cont)

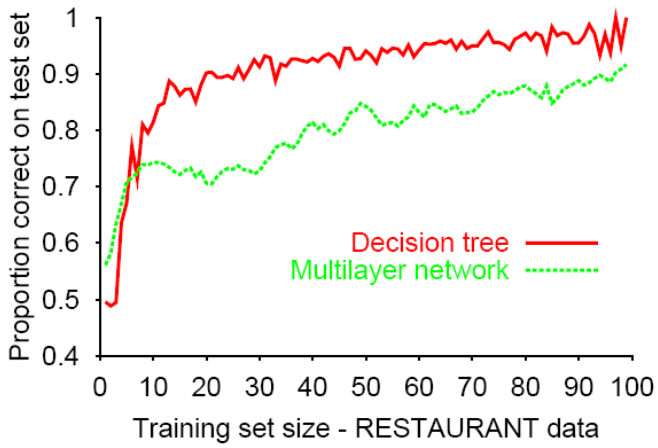
Em cada **época**, somar actualizações de gradiente para todos os exemplos e aplicar.

Curva de treino para 100 exemplos do restaurante: encontra ajustamento perfeito.



Problemas típicos: convergência lenta, mínimos locais.

Curva de aprendizagem para rede multi-camada com 4 unidades escondidas:



Redes multi-camada são adequadas para tarefas complexas de reconhecimento de padrões, mas o resultado não pode ser facilmente compreendido.

Utilização prática do Algoritmo

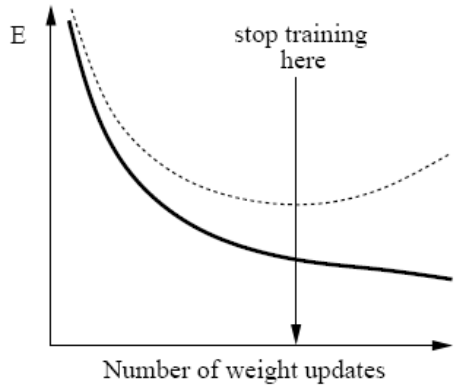
Para o caso da função sigmóide é habitual inicializarem-se os pesos com valores aleatórios no intervalo [-0,5;0,5] ou [-1,1]

O prolongamento da aprendizagem pode levar a problemas de sobreajustamento, ou seja, a rede classifica bem o conjunto de treino mas mais o conjunto de validação.

O número de exemplos de treino também é difícil de determinar, mas pode-se seguir uma das seguintes regras práticas:

- ✓ O número de exemplos deve ser 5 a 10 vezes mais que o número de pesos.
- ✓ Treino da rede para classificar com precisão 1-(ϵ/2) exemplos de treino.
- ✓ Para se obter precisão 1-ϵ no conjunto de validação serão necessários tantos exemplos de treino quanto o número de pesos na rede divididos por ϵ.
- ✓ Unidades escondidas a menos, resulta na impossibilidade de aprendizagem da função; unidades escondidas a mais, pode redundar em sobreajustamento.

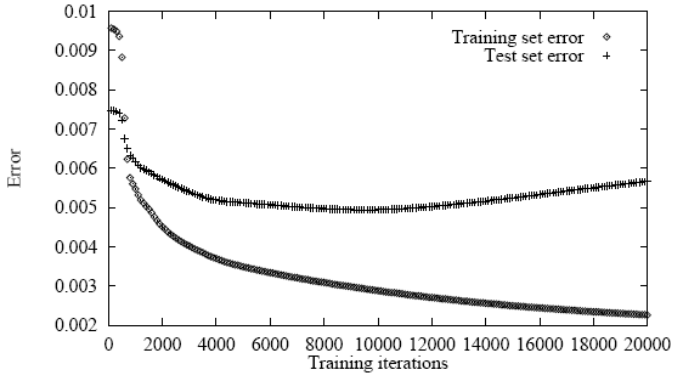
Sobreajustamento

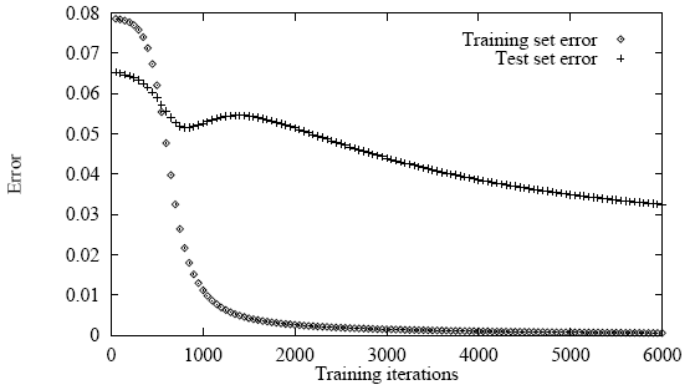


Os exemplos devem ser divididos em conjuntos de treino e em conjunto de validação. Deve haver um conjunto de teste que não é utilizado na aprendizagem.

Utiliza-se o conjunto de treino no algoritmo de aprendizagem, parando-se quando se minimiza o erro no conjunto de validação.

Sobreajustamento (quando parar?)





Problemas de convergência

Pode haver grandes oscilações no erro. Uma das formas consiste em alterar a regra para utilizar **momento - μ**.

As variações dos pesos da iteração *t* para *t + 1* são dadas por:

$$\Delta W_{j,i}(t+1) = \eta \times a_j \times \Delta_i + \mu \Delta W_{j,i}(t)$$
$$\Delta W_{k,j}(t+1) = \eta \times a_k \times \Delta_j + \mu \Delta W_{k,j}(t).$$

Paralelo com a bola a descer superfície:

- ✓ Ritmo de aprendizagem: velocidade (valores típicos entre 0.1 e 0.9).
- ✓ Momento: inércia – mantém direcção do movimento anterior.
- ✓ Acelerar convergência através de treino em lote. As alterações nos pesos de todos os casos de treino são acumuladas e só no final propagadas após o processamento integral do

conjunto de treino. Esta versão normalmente converge mais rapidamente, mas pode ficar preso mais facilmente em mínimos locais.

A maioria dos cérebros tem muitos neurónios, cada neurónio ≈ unidade de limiar

Perceptrões (redes monocamada) são pouco expressivos

Redes multi-camada são suficientemente expressivas; podem ser treinadas pelo método de descida do gradiente, i.e. retropropagação do erro

Sobreajustamento é um problema sério a evitar, devendo-se utilizar técnicas de validação cruzada.

Muitas aplicações: fala, condução, reconhecimento escrita, detecção fraudes, etc.