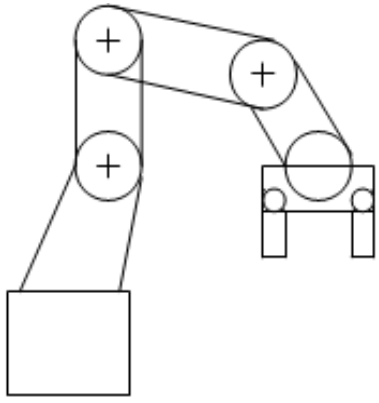


2	7	1
4	3	8
6	5	

Programação Lógica

Lógica de Primeira Ordem

Luiz Eduardo da Silva

Universidade Federal de Alfenas

Agenda

Agenda

Lógica

Na computação:

1. Lógica de proposições, composta por proposições e operadores lógicos, exemplo: $A \vee \neg B \rightarrow C$
2. Lógica de predicados (lógica de primeira ordem) composta por relações lógicas e qualificadores, exemplo:

$$\forall X \forall Y \exists P(pai(P, X) \wedge pai(P, Y) \wedge (X \neq Y) \rightarrow irmao(X, Y))$$

3. **Prolog** implementa um modelo lógico que está entre a Lógica de proposições e lógica de primeira ordem, **chamado lógica de cláusulas definidas**.

Lógica de proposições

Lógica booleana ou cálculo proposicional é um sistema matemático construído sobre o conjunto $\{verdadeiro, falso\}$.

- *verdadeiro* e *falso* são representados por 0 e 1 e são os valores booleanos (ou lógicos).
- As operações básicas lógicas são:
  - A **negação** (símbolo $\neg$): $\neg 0 = 1$ e $\neg 1 = 0$.
  - A **conjunção** (símbolo $\wedge$):
  $$\begin{aligned} 0 \wedge 0 &= 0 \\ 0 \wedge 1 &= 0 \\ 1 \wedge 0 &= 0 \\ 1 \wedge 1 &= 1 \end{aligned}$$
  - A **disjunção** (símbolo $\vee$):
  $$\begin{aligned} 0 \vee 0 &= 0 \\ 0 \vee 1 &= 1 \\ 1 \vee 0 &= 1 \\ 1 \vee 1 &= 1 \end{aligned}$$

Lógica de proposições

- Além dessas operações básicas ainda temos:
  - O **ou-exclusivo** (símbolo $\oplus$):

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

  - A **igualdade** (símbolo $\leftrightarrow$):

$$\begin{aligned} 0 \leftrightarrow 0 &= 1 \\ 0 \leftrightarrow 1 &= 0 \\ 1 \leftrightarrow 0 &= 0 \\ 1 \leftrightarrow 1 &= 1 \end{aligned}$$

  - A **implicação** (símbolo $\rightarrow$):

$$\begin{aligned} 0 \rightarrow 0 &= 1 \\ 0 \rightarrow 1 &= 1 \\ 1 \rightarrow 0 &= 0 \\ 1 \rightarrow 1 &= 1 \end{aligned}$$

Observações sobre a Implicação Lógica

- A implicação lógica é uma relação entre duas proposições, em que a primeira proposição (chamada de **antecedente**) é uma condição para que a segunda proposição (chamada de **consequente**) seja verdadeira.
- A implicação lógica é definida da seguinte forma: a proposição "p $\rightarrow$ q"é verdadeira quando a proposição "p"é falsa ou quando as proposições "p"e "q"são ambas verdadeiras. Em outras palavras, a implicação lógica é falsa somente quando a proposição "p"é verdadeira e a proposição "q"é falsa.
- Por exemplo, considere a proposição "Se está chovendo, então o chão está molhado", em que "está chovendo"é o antecedente e "o chão está molhado"é o consequente. Essa proposição é verdadeira quando está chovendo e o chão está molhado, ou quando não está chovendo. No entanto, se está chovendo e o chão não está molhado, a proposição é falsa.

Exemplo da Implicação Lógica

	Antecendente		**Consequente**	
se	está chovendo (V)	então	chão está molhado (V)	V
se	está chovendo (V)	então	o chão NÃO está molhado (F)	F
se	NÃO está chovendo (F)	então	o chão está molhado (V)	V
se	NÃO está chovendo (F)	então	o chão NÃO está molhado (F)	V

"NÃO está chovendo"implica que "o chão está molhado"é possível (verdadeira) porque o chão ficou molhado por outra condição (um copo de água que caiu, por exemplo).

Lógica de proposições

- Pode-se estabelecer relações entre essas operações. Por exemplo: todas as fórmulas da lógica proposicional podem ser reescritas usando somente os operadores $\wedge$ e $\neg$:
  - $$\begin{array}{lll} P \vee Q & & \neg(\neg P \wedge \neg Q) \\ P \rightarrow Q & \neg P \vee Q & \neg(P \wedge \neg Q) \\ P \leftrightarrow Q & (P \rightarrow Q) \wedge (Q \rightarrow P) & \neg(P \wedge \neg Q) \wedge \neg(Q \wedge \neg P) \\ P \oplus Q & \neg(P \leftrightarrow Q) & \neg(\neg(P \wedge \neg Q) \wedge \neg(Q \wedge \neg P)) \end{array}$$
- A **lei distributiva** para E e OU:
  - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
  - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
- Parece com a lei distributiva da **adição** e **multiplicação**, mas não é igual.

Cláusulas

- A forma normal conjuntiva (FNC) é uma **conjunção** de cláusulas, onde uma cláusula é uma **disjunção** de literais (predicados ou variáveis lógicas)
- Toda fórmula proposicional pode ser transformada como um conjunto de cláusulas na forma normal conjuntiva.
- Essa transformação está baseada em regras sobre equivalência lógica baseadas na lei da dupla negação, leis de De Morgan e a lei distribuitiva.

Cláusulas

$$
\begin{aligned}
((A \vee B) \wedge (\neg A \vee C)) \rightarrow D &= \neg((A \vee B) \wedge (\neg A \vee C)) \vee D \\
&= (\neg(A \vee B) \vee \neg(\neg A \vee C)) \vee D \\
&= ((\neg A \wedge \neg B) \vee (A \wedge \neg C)) \vee D \\
&= (\neg A \vee A \vee D) \wedge \\
&\quad (\neg A \vee \neg C \vee D) \wedge \\
&\quad (\neg B \vee A \vee D) \wedge \\
&\quad (\neg B \vee \neg C \vee D) \\
&= (A \vee D \vee \neg A) \wedge \\
&\quad (D \vee \neg A \vee \neg C) \wedge \\
&\quad (A \vee D \vee \neg B) \wedge \\
&\quad (D \vee \neg B \vee \neg C)
\end{aligned}
$$

Cláusulas

- O resultado é que a fórmula original é equivalente a quatro cláusulas ligadas por conjunções.
- Tudo que pode ser expresso na lógica de proposições também pode ser expresso na lógica de cláusulas.
- Uma notação abreviada para a lógica de cláusulas (na forma normal conjuntiva) foi proposta por **Kowalki**, como segue:
  - $$A_1 \vee ... \vee A_m \vee \neg B_1 \vee ... \vee \neg B_n$$

    é representada por:

    $$A_1, ..., A_m \leftarrow B_1, ...B_n$$
- Nessa notação simplificada remove-se todas as conjunções, negações e disjunções e troca-se por vírgula e o operador $\leftarrow$.
- Uma cláusula representa uma generalização de um argumento: conclusões $\leftarrow$ condições.

Cláusulas

- Uma cláusula representa uma generalização de um argumento, $\underline{\text{conclusões(P)} \leftarrow \text{condições(Q)}}$: $P \vee \neg Q = P \leftarrow Q$

$$
\begin{aligned}
& A_1 \vee A_2 \vee ...A_m \vee \neg B_1 \vee \neg B_2 \vee ...\neg B_n \\
= \quad & (A_1 \vee A_2 \vee ...A_m) \vee \neg(B_1 \wedge B_2 \wedge ...B_n) \\
= \quad & (A_1 \vee A_2 \vee ...A_m) \leftarrow (B_1 \wedge B_2 \wedge ...B_n) \\
= \quad & A_1, A_2, ...A_m \leftarrow B_1, B_2, ...B_n \qquad \textit{//cláusula Kowalski}
\end{aligned}
$$

- É possível passar termos de uma lado para o outro da condicional, negando-se o termo movimentado. É a regra de transposição de literais, por exemplo:

$$
\begin{aligned}
(A \vee B) \quad & \leftarrow \quad (C \wedge D) \\
= \qquad A \quad & \leftarrow \quad (\neg B \wedge C \wedge D) \\
= \qquad \textit{false} \quad & \leftarrow \quad (\neg A \wedge \neg B \wedge C \wedge D) \\
= \quad (A \vee B \vee \neg C \vee \neg D) \quad & \leftarrow \quad \textit{true}
\end{aligned}
$$

Cláusulas

- Algumas igualdades ajudam a entender melhor o significado de algumas condições:

$$\begin{aligned} & (A \leftarrow B) \\ = & (A \vee \mathit{false} \leftarrow B \wedge \mathit{true}) \\ = & (\mathit{false} \leftarrow \neg A \wedge B \wedge \mathit{true}) \\ = & (\mathit{false} \vee A \vee \neg B \leftarrow \mathit{true}) \end{aligned}$$

- outro exemplo ("cláusula nula"):

$$\begin{aligned} & \mathit{false} \vee \neg \mathit{true} \\ = & (\mathit{false} \leftarrow \mathit{true}) \\ = & (\leftarrow \neg \mathit{false} \wedge \mathit{true}) \\ = & (\mathit{false} \vee \neg \mathit{true} \leftarrow) \\ = & \leftarrow \end{aligned}$$

Cláusulas

Um exemplo concreto

$$
\begin{array}{rrcl}
 & \underline{\text{não}}\ \text{chove} & \leftarrow & \text{faz\_sol} \\
= & \text{falso} & \leftarrow & \text{chove}\ \underline{\text{e}}\ \text{faz\_sol} \\
= & \underline{\text{não}}\ \text{chove}\ \underline{\text{ou}}\ \underline{\text{não}}\ \text{faz\_sol} & \leftarrow & \text{verdadeiro} \\
= & \underline{\text{não}}\ \text{faz\_sol} & \leftarrow & \text{chove}
\end{array}
$$

Cláusulas

Dada a notação abreviada, temos as seguintes classificações para as cláusulas:

- Se $m > 1$, temos $A_1, A_2, ... \leftarrow ....$ Existem várias consequências ou conclusões. Essas são cláusulas indefinidas pois a conclusão é uma disjunção: $A_1 \vee A_2 ...$
- Se $m \leq 1$, temos $A \leftarrow ....$ Essas são as cláusulas de **Horn** (cláusulas definidas). Existem as seguintes subclasses:
  - Se $m = 1, n > 0$, temos $\boxed{A \leftarrow B_1, B_2, ...B_n}$ são cláusulas definidas (uma única conclusão), também chamadas de **regras**.
  - Se $m = 1, n = 0$, temos $\boxed{A \leftarrow}$, são cláusulas definidas incondicionais, também chamados de **fatos**.
  - Se $m = 0, n > 0$, temos $\boxed{\leftarrow B_1, ..., B_n}$, são negações puras, também chamadas de **perguntas**.
  - Se $m = 0, n = 0$, temos $\boxed{\leftarrow}$. É a cláusula vazia.

Resumindo

$$A_1 \vee ... \vee A_m \vee \neg B_1 \vee ... \vee \neg B_n$$

$$\underbrace{A_1, ..., A_m}_{\text{conclusões}} \underbrace{\leftarrow}_{se} \underbrace{B_1, ...B_n}_{\text{condições}}$$

$m > 1$ cláusulas indefinidas (mais de uma conclusão possível)

$$m \leq 1 \begin{cases} m = 1, n > 0 & \boxed{A \leftarrow B_1, B_2, ...B_n} \text{ (regra)} \\ m = 1, n = 0 & \boxed{A \leftarrow} \text{ (fato)} \\ m = 0, n > 0 & \boxed{\leftarrow B_1, ..., B_n} \text{ (consulta)} \\ m = 0, n = 0 & \boxed{\leftarrow} \text{ (cláusula nula)} \end{cases}$$

Cláusulas de Horn com Predicados

Exemplo

$$
\begin{aligned}
\text{pai(joao, jose)} &\leftarrow \\
\text{pai(jose, pedro)} &\leftarrow \\
\text{mae(maria, jose)} &\leftarrow \\
\text{fem(X)} &\leftarrow \text{mae(X,Y)} \\
\text{irmao(X,Y)} &\leftarrow \text{pai(P,X), pai(P,Y), X} \neq \text{Y}
\end{aligned}
$$

- Um programa lógico é uma conjunto de cláusulas.
- O significado de um programa é tudo o que pode ser deduzido desse conjunto de cláusulas.

Prova por refutação (para proposições)

- Regra de eliminação:

$$\frac{\begin{array}{lll} A & \leftarrow & B, C, \ldots \\ D & \leftarrow & \boxed{A}, E, \ldots \end{array}}{D \leftarrow B, C, \ldots, E, \ldots}$$

- Provar que uma proposição Q é derivada de P (ou seja, P então Q, ou $P \rightarrow Q$) é o mesmo que provar $P \wedge \neg Q \rightarrow \mathit{false}$. Esta é a prova por absurdo: junta-se a conclusão negada com as condições (premissas) para se chegar a uma falsidade.
- Exemplo: provar que Q é derivado do conjunto de cláusulas $(Q \leftarrow B, D | Q \leftarrow C, A | C \leftarrow | A \leftarrow D | D \leftarrow)$

Prova por refutação (para proposições)

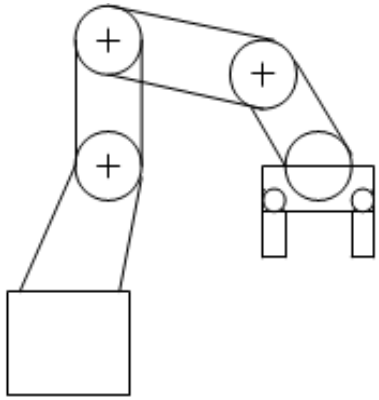
- $(Q \leftarrow B, D | Q \leftarrow C, A | C \leftarrow | A \leftarrow D | D \leftarrow)$
- Prova:

Q	$\leftarrow$	B, D	regra 1
Q	$\leftarrow$	C, A	regra 2
C	$\leftarrow$		fato 1
A	$\leftarrow$	D	regra 3
D	$\leftarrow$		fato 2
	$\leftarrow$	Q	Pergunta? (primeira tentativa)
	$\leftarrow$	B, D	Experimenta a regra 1
			Falha pois não tem como eliminar B
			B não aparece como conclusão em nenhuma fato ou regra
	$\leftarrow$	Q	Pergunta? (segunda tentativa)
	$\leftarrow$	C, A	C $\leftarrow$ é o fato 1
	$\leftarrow$	A	eliminando A aplicado a regra 3
	$\leftarrow$	D	D $\leftarrow$ é o fato 2
	$\leftarrow$		Chega-se a cláusula vazia.

Prova por refutação

- Chama-se **prova por refutação** porque a pergunta $\leftarrow$ Q equivale à negação $\neg Q$ e chegar na cláusula vazia é o mesmo que chegar ao falsidade. Ou seja, nega-se a pergunta e chega-se a uma conclusão falsa ($\neg Q \leftarrow \mathit{false}$).
- A **resolução** implementada em linguagens de programação lógica como Prolog generaliza esse processo de refutação da lógica de cláusulas permitindo predicados parametrizáveis.
- As cláusulas do programa são manipulados por um processo de **unificação** (associado à regra de **eliminação**) que gera de forma sistemática e controlada novas instâncias de cláusulas.
- Conforme já foi observado, o significado de um programa é tudo o que pode ser deduzido desse conjunto de cláusulas.

Agenda



2	7	1
4	3	8
6	5	

Programação Lógica

Programação Lógica

Luiz Eduardo da Silva

Universidade Federal de Alfenas

Programação Lógica

- Prolog = **Pro**gramming in **log**ic, é uma linguagem que implementa esse algoritmos de unificação e eliminação apresentados anteriormente, junto com outros mecanismos e estruturas pré-definidas
- Um programa prolog é um conjunto de predicados.
- Um predicado é um conjunto de cláusulas (regras ou fatos).
- Variável é um elemento não especificado em Prolog. Usa-se os nomes iniciados com letra maiúscula.
- Constante é um elemento especificado. Número ou texto (em minúscula ou entre aspas)

Programação Lógica

Formalmente:

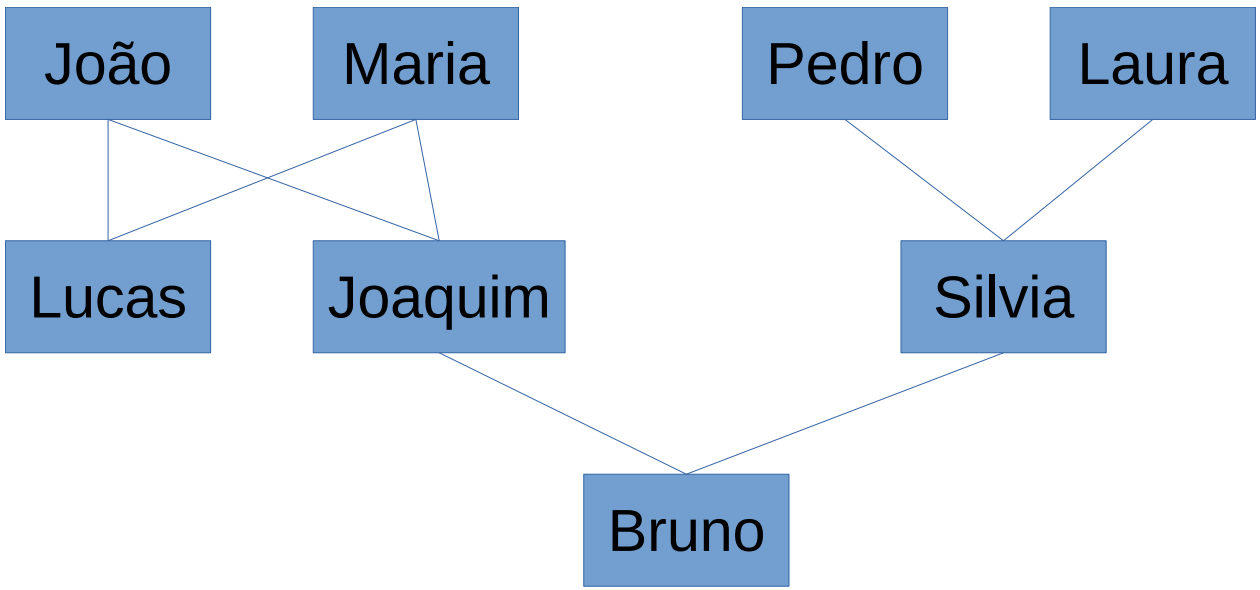
- Cláusulas são fórmulas lógicas construídas sobre fórmulas atômicas.
  - Se P é uma fórmula atômica então sua negação \+ P também é uma fórmula atômica
  - Se P e Q são fórmulas atômicas então a conjunção (P,Q), a disjunção (P;Q) e a condicional (P:-Q) são fórmulas não atômicas. O símbolo :- significa a condicional invertida P:-Q é o mesmo que Q $\rightarrow$ P.
- Termos são construídos a partir de variáveis e constantes. Representado por um símbolo (functor) seguido por argumentos. Ex: p(t1,t2,...tn) ou f(t1,t2,...tn).

Prolog

- É uma linguagem de programação centralizada em torno de um pequeno conjunto de mecanismos:
  - Pesquisa por padrão (pattern-matching).
  - Estrutura de dados baseada em árvore.
  - Backtracking automático.
- Este pequeno conjunto constitui uma estrutura poderosa e flexível de programação.
- Prolog é especialmente indicado para objetos - principalmente objetos estruturados - e as relações entre estes objetos.
- Muito usada como linguagem suporte em aplicações não numéricas e Inteligência Artificial (I.A.).
- Os desenvolvedores desta idéia foram: **Robert Kowalski**: responsável pelo lado teórico, **Maarten Van Emdem**: responsável pela demonstração experimental e **Alain Comerauer**: responsável pela Implementação

Prolog

- Prolog é uma linguagem para computação simbólica, não numérica. Ela é indicada para resolver problemas que envolvem objetos e relações entre estes objetos.
- A figura abaixo apresenta um exemplo de relações familiares:



Prolog

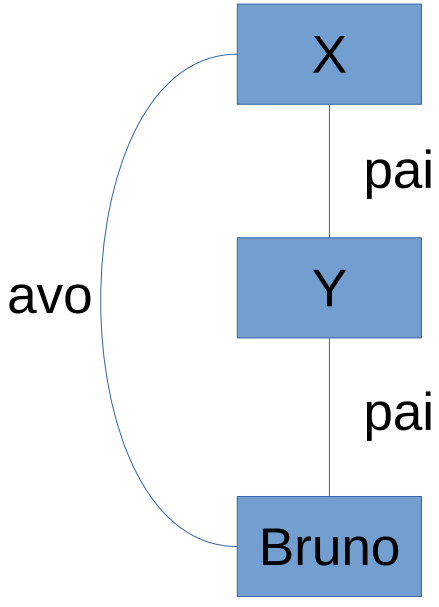
- Escolhemos pai como o nome da relação e joao e joaquim como os argumentos.
- A árvore genelógica anterior pode ser definida pelo seguinte programa prolog:

```
1  pai(joao,joaquim).  % joao e' pai de joaquim.
2  pai(joao,lucas).
3  pai(pedro,silvia).
4  pai(joaquim,bruno).
5  mae(maria,joaquim). % maria e' mae de joaquim.
6  mae(maria,lucas).
7  mae(laura,silvia).
8  mae(silvia,bruno).
```

Prolog

- Alguns exemplos de consulta a esse programa:
  - ?- pai(pedro,silvia).
  - ?- pai(pedro,claudio).
  - ?- pai(X,silvia).
  - ?- pai(joao,X).
  - ?- pai(X,Y).
- Uma questão composta, "quem é o avô de Bruno?"

```
?- pai(Y,bruno),pai(X,Y).
```



Prolog

- Como uma extensão para o programa prolog anterior podemos escrever uma **regra** para as relações familiares: avô paterno, avô materno, avó paterna e avó materna.
- Em prolog estas relações podem ser definidas a partir das relações pai e mãe. Assim:

```
avo_paterno(X,Y) :- pai(X,Z), pai(Z,Y).
```

- Esta regra pode ser lida como: X é avo de Y se X é pai de Z e Z é pai de Y.

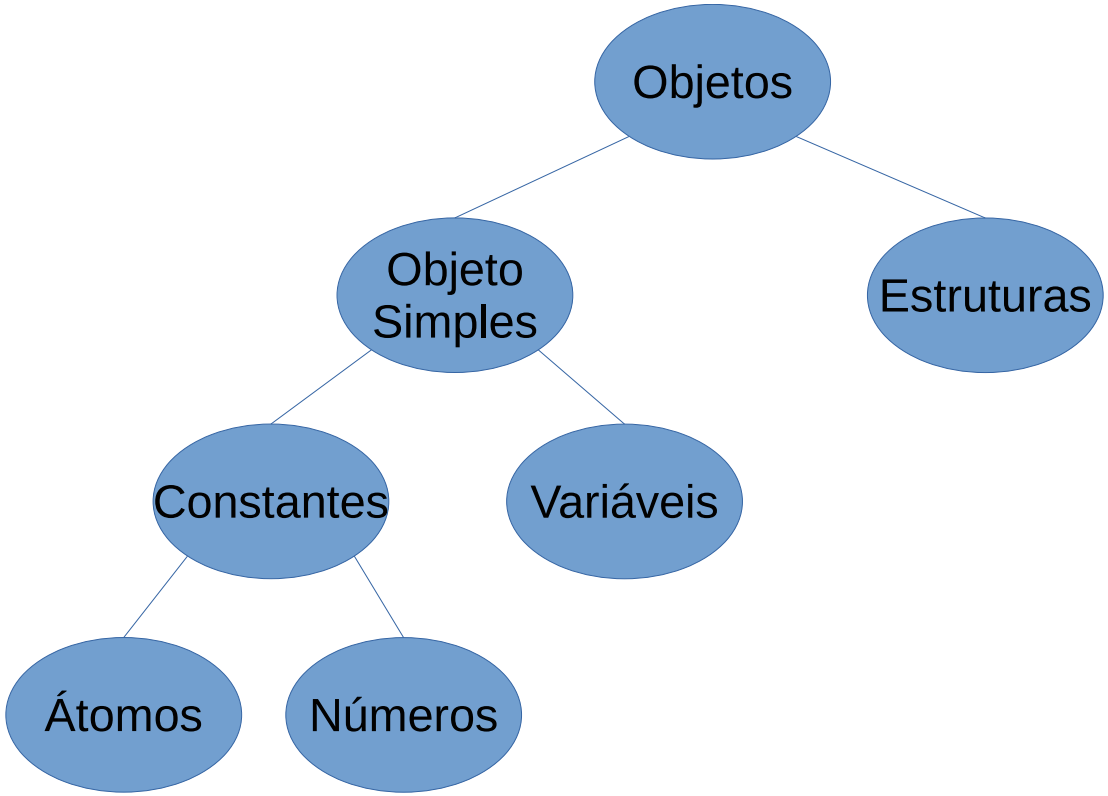
Prolog

- A relação antecessor no programa de relações familiares pode expressa com algumas regras:
  - A duas regras que definem o antecessor diretamente (pai e mãe).
  - e as duas últimas que definem o antecessor de forma indireta (recursivamente).

```
1 antecessor(X,Y) :- pai(X,Y).
2 antecessor(X,Y) :- mae(X,Y).
3 antecessor(X,Y) :- pai(X,Z), antecessor(Z,Y).
4 antecessor(X,Y) :- mae(X,Z), antecessor(Z,Y).
```

Objetos Prolog

- Classificação dos objetos Prolog:



Objetos Prolog

- **Objetos** - Prolog pode reconhecer os tipos dos objetos sem que seja necessário uma declaração explícita como em outras linguagens, porque cada objeto tem uma forma diferente. Por exemplo, toda variável prolog começa com letra maiúscula.
- **Átomos e Números** - Átomos podem ser escritos de três formas:
  1. Strings de letras, números e underscore começando com letra minúscula. Ex: ana, nil, x25, x_25, etc.
  2. Strings de caracteres especiais: Ex: <—>, ===>, etc.
  3. Strings de caracteres entre aspas simples. Ex: 'Tom', 'América do Sul', etc.
  4. Os números usados em prolog incluem os números inteiros e os números reais.

Objetos Prolog

- **Variáveis** - são formadas por sequência de letras, dígitos e underscore que começam com letra maiúscula ou underscore. Ex: X, Resultado, _x25, etc.
- Quando uma variável aparece numa regra prolog somente uma vez, nós não precisamos inventar um nome para ela, basta usar a variável anônima representada por um símbolo underscore '_'. Exemplo:

```
tem_filho(X) :- pai(X,Y).
```

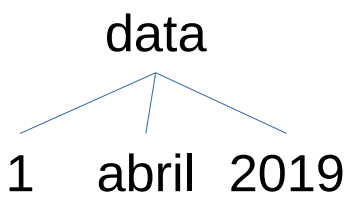
Pode ser escrita simplesmente como:

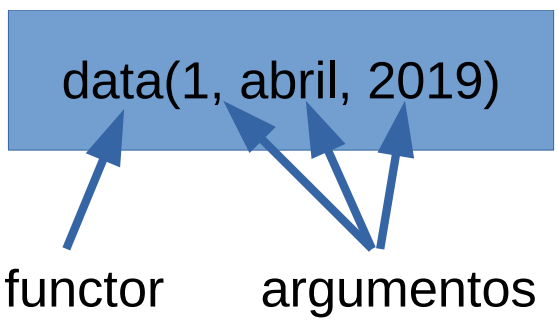
```
tem_filho(X) :- pai(X,_).
```

- O escopo de uma variável está na mesma regra, ou seja, toda variável prolog é local a regra em que ela aparece.

Objetos Prolog

- **Estruturas** - Objetos estruturados, objetos compostos ou simplesmente estruturas são objetos que contém vários componentes.
- A data pode ser vista como uma estrutura com 3 componentes: dia, mês e ano. Para combinar os componentes num único objeto temos que escolher um functor. Para este exemplo o functor é data.





- Todos os componentes neste exemplo são constantes (dois inteiros e um átomo). No entanto, podemos ter variáveis como componentes de uma estrutura, assim: data(D,abril,2019).

Objetos Prolog

- Todo objeto em prolog é representado por uma árvore. Por exemplo, vamos definir estruturas prolog para definir alguns objetos geométricos simples. São eles:
  - ponto no plano cartesiano - é definido por duas coordenadas.
  - segmento - é definido por dois pontos.
  - triângulo - é definido por três pontos.

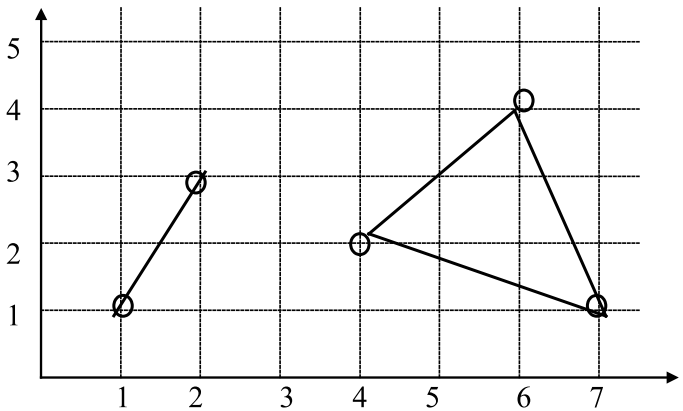
Em prolog

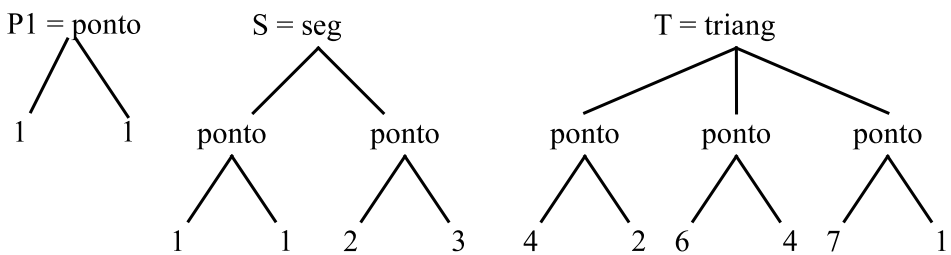
P1=ponto(1,1)
P2=ponto(2,3)
S=seg(P1,P2)=seg(ponto(1,1),ponto(2,3))
T=triang(ponto(4,2),ponto(6,4),ponto(7,1))

Objetos Prolog

Em prolog

P1=ponto(1,1)
P2=ponto(2,3)
S=seg(P1,P2)=seg(ponto(1,1),ponto(2,3))
T=triang(ponto(4,2),ponto(6,4),ponto(7,1))





Listas Prolog

- A lista é uma estrutura de dados simples largamente utilizada em programação não-numérica. Uma lista é uma sequência de itens como: ana, pedro, paulo, joao.

Em prolog

[ana, pedro, paulo, joao]

- A lista vazia é representada por [].
- A lista não vazia contém duas partes:
  - A cabeça da lista, o primeiro item da lista.
  - A cabeça da lista, o primeiro item da lista.
- Por exemplo, para a lista anterior:
  - *ana* = cabeça da lista.
  - [*pedro*, *paulo*, *joao*] = cauda da lista.

Listas Prolog

Alguns exemplos

Lista	Cabeça	Cauda
$[a, b, c]$	a	$[b, c]$
$[coisa]$	$coisa$	$[]$
$[]$	$indefinido$	$indefinido$
$[[1, 3], [2, 3, 4], []]$	$[1, 3]$	$[[2, 3, 4], []]$

Listas Prolog

- Variáveis são muitas vezes usadas para representar elementos desconhecidos e elementos genéricos de uma lista. Exemplo:

```
[A, B, pedro]
```

- O primeiro e segundo elementos desta lista são genéricos ou desconhecidos, representados pelas variáveis A e B. O terceiro elemento é conhecido e está representado pelo átomo pedro.
- Um número indeterminado de elementos genéricos na cauda de uma lista pode ser representado por uma barra vertical (|) seguida de uma variável. Exemplo:

```
[rosa, cravo | X]
```

- A barra vertical seguida da variável X indica que após o segundo elemento desta lista há uma quantidade ignorada de

Listas Prolog

- A barra vertical pode ser traduzida por "resto da lista". Isto faz com que $[5, 9, 7|X]$ seja considerada como a lista cujo primeiro elemento é 5, o segundo elemento é 9, o terceiro elemento é 7 e o resto dos elementos é X.
- Um padrão de lista muito importante é aquele cujo o primeiro elemento é uma variável e cujo resto da lista é também uma variável.

Exemplos

[X|Y]
[Primeiro|Resto]
[Topo|Cauda]

Listas Prolog

- Tais listas representam qualquer lista com pelo menos um elemento. Considere para fixar idéias, o caso de $[X|Y]$. Este padrão pode ser instanciado a lista $[ana]$ de um único elemento desde que X represente o valor 'ana' e Y represente a lista vazia. Num outro exemplo, se prolog tenta satisfazer a consulta:

```
?- placar([X|Y]).
```

e encontra o fato na sua base de conhecimento:

```
placar([0,1,2,3,4,5]).
```

X será instanciado ao valor 0 ($X = 0$) e Y será instanciado a lista $[1, 2, 3, 4, 5]$ ($Y = [1, 2, 3, 4, 5]$).

Listas Prolog

- Exemplos de casamentos de listas, onde as variável livres serão instanciadas aos valores da lista.

Lista 1	Lista 2	Variáveis instanciadas
$[X, Y, Z]$	$[joao, maria, jose]$	X=joao,Y=maria,Z=jose
$[7]$	$[X\|Y]$	X=7, Y=[]
$[1, 2, 3, 4]$	$[X, Y\|Z]$	X=1, Y=2, Z=[3, 4]
$[1, 2]$	$[3, X]$	Falha! As cabeças são diferentes.

Pertence

Predicado pertence

```
pertence(X,[X|Y]).
pertence(X,[Primeiro|Resto]) :- pertence(X,Resto).
```

Concatenação

- Para concatenar listas nos definiremos a relação:

```
ap(L1,L2,L3)
```

Onde L1 e L2 são duas listas, e L3 é o resultado da concatenação de L1 e L2. Por exemplo:

```
ap([a,b],[c,d],[a,b,c,d]).
```

é verdade, mas

```
ap([a,b],[c,d],[a,b,a,c,d]).
```

é falsa.

Concatenação

- Na definição da relação para concatenação de lista ap, nós precisamos tratar dois casos, dependendo do primeiro argumento L1:
  1. Se o primeiro argumento é uma lista vazia então o segundo e o terceiro argumento são o mesmo valor, isto é, a concatenação da lista vazia com a lista L é a própria lista L:

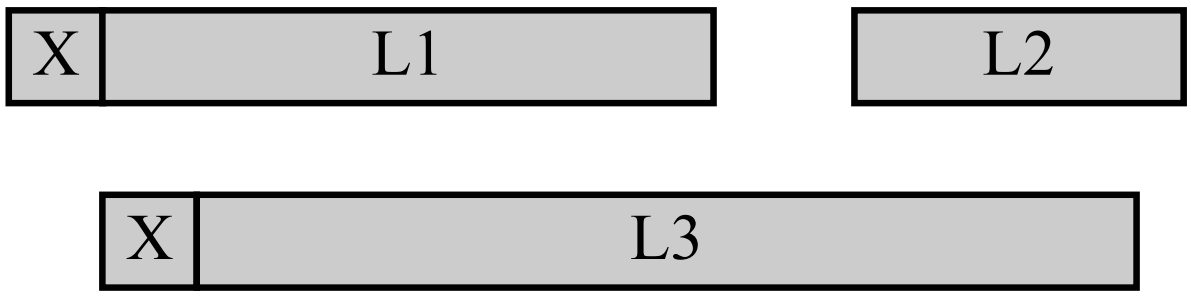
```
ap([],L,L).
```

  2. Se o primeiro elemento não é uma lista vazia, então ele tem cabeça e cauda. Seja o primeiro argumento a seguinte lista genérica [X|L1]. O resultado da concatenação desta lista com a lista L2 é uma lista [X|L3], onde L3 é a concatenação de L1 (cauda da primeira lista), com L2 (segunda lista). Em prolog, esta regra é escrita como:

```
ap([X|L1], L2, [X|L3]) :- ap(L1, L2, L3).
```

Concatenação

```
ap([],L,L).
ap([X|L1], L2, [X|L3]) :- ap(L1, L2, L3).
```



Remoção

```
apaga(X, [X|Y], Y).
apaga(X, [Y|Z], [Y|Z1]) :- apaga (X, Z, Z1).
```

A relação apaga é similar a relação membro da lista. De novo temos que pensar recursivamente em dois casos:

- Se X é a cabeça da lista então o resultado é a cauda da lista.
- Se X está na cauda da lista então ela será retirada de lá.

Operações Aritméticas e Relacionais

- Prolog é uma linguagem para computação simbólica, sendo assim a necessidade para cálculos numéricos é comparativamente modesta.
- Alguns dos operadores básicos para operações aritméticas em prolog são:
  - $+$, para adição
  - -, para subtração
  - *, para multiplicação
  - $/$, para divisão
  - mod, para módulo, resto da divisão inteira.
- Os operadores relacionais de prolog são:
  - $X > Y$, para X é maior do que Y
  - $X < Y$, para X é menor do que Y
  - $X >= Y$, para X é maior ou igual a Y
  - $X <= Y$, para X é menor ou igual a Y
  - $X \backslash= Y$, para X é diferente de Y

Aritmética e Relacionais

- Para fazer cálculos em prolog, lançamos mão de predicados com a declaração 'is'. Esta declaração é infixa e, portanto, deve ser colocada entre dois objetos. O primeiro objeto é uma variável e o segundo é uma expressão aritmética. O efeito do operador 'is' é realizar o cálculo indicado pela expressão e colocar o valor encontrado na variável.

Exemplos

```
mdc(X,X,X).
mdc(X,Y,D) :- X < Y, Y1 is Y - X, mdc(X,Y1,D).
mdc(X,Y,D) :- Y < X, mdc(Y,X,D).
nro_elementos([],0).
nro_elementos([_|Y], N) :- nro_elementos(Y,NY), N is NY + 1.
```

Programação com listas

Exercício 1

Escreva o predicado Prolog máximo/2 que retorna o maior valor de uma lista de inteiros. Exemplo de uso:

```
?-maximo(X,[4,3,7,9,1]).
X = 9
```

Programação com listas

Exercício 2

Escreva o predicado Prolog comprimento/2 que calcula o comprimento de uma lista. Exemplo de uso:

```
?-comprimento(X,[a,b,c]).
X = 3
```

Programação com listas

Exercício 3

Escreva o predicado Prolog nesimo/3 que encontra o n-ésimo valor de uma lista. Exemplo de uso:

```
?-nesimo(3,[4,3,7,9,1],X).
X = 7
```

Programação com listas

Exercício 4

Escreva o predicado Prolog total/2 que calcula a soma dos valores numa lista de inteiros. Exemplo de uso:

```
?-total([4,3,7,9,1],X).
X = 24
```

Depuração

- Prolog tem alguns predicados pré-definidos para auxiliar na depuração de programas.
  - **trace**: o predicado trace sem argumentos, liga o mecanismo de depuração de prolog. Cada consulta(meta) é expressa com os seguintes prefixos (eventos):

CALL	quando uma regra ou fato é aplicado para satisfação de uma consulta
EXIT	quando uma consulta é satifeita (pela aplicação de um fato ou regra)
REDO	quando o backtracking (retrocesso) é invocado para tentar outra regra ou fato na satisfação da meta
FAIL	quando a aplicação do fato ou regra para satisfação da regra falha.

  - **spy(P)**: faz com que os eventos relacionados ao predicado P sejam acompanhados.
  - Para cancelar o efeito desses predicados, usa-se **notrace**, **nospy(P)** e **nodebug**.

Depuração

- Quando a depuração está ligada, o interpretador Prolog interrompe a execução de cada consulta e mostra o evento e a meta que está sendo avaliada.
- Os controles para continuar o processamento são os seguintes:

Opção	**Significado**	**Descrição**
w	write	imprime a meta
c	creep	segue para o próximo evento
s	skip	salta até o próximo evento
l	leap	salta até o próximo evento acompanhado
r	retry	volta a primeira satisfação da meta
f	fail	causa a falha da meta
b	break	inicia uma nova sessão recursiva do interpretador (CTRL-D, interrompe)
a	abort	interrompe a depuração

Trabalhando com ordem

Permuta

- se a lista é vazia, a permutação também é um lista vazia;
- senão, seleciona-se um elemento da lista para ser o primeiro da lista resposta e chama-se recursivamente a permutação para lista restante (sem o elemento selecionado).

```
seleciona(X,[X|Y], Y).
seleciona(X,[Y|Z],[Y|W]) :- seleciona(X,Z,W).

permuta(A,[B|D]) :- seleciona(B,A,C), permuta(C,D).
permuta([], []).
```

Exercício

Lista reversa

Usando de concatenação de lista ap/3, defina reversa/2. Codifique as regras:

- (trivial) o reverso de uma lista vazia é também uma lista vazia.
- (geral) o reverso de uma lista é o reverso da cauda concatenada com a lista que contém somente a cabeça.

```
ap([],L,L).
ap([A|B], C, [A|D]) :- ap(B, C, D).
```

O corte !

- Em algumas situações é preciso impedir que o programa prolog volte atrás em uma decisão tomada (backtracking). Isto pode ser feito usando um predicado que se chama corte e é representado por um ponto de exclamação.

Representação

$< a >: - < b >, ..., < c >, !, < d >, < f >, ...$
$< a >: - < g >, < h >, < i >, ...$

- Este corte é usado para interromper backtracking em dois casos:
  - Nas condições anteriores a ele na mesma regra.
  - Nas regras abaixos daquela em que ele aparece.

Exemplo de uso do corte

Predicado add

```
add(X,L,L) :- in(X, L), !.
add(X,L,[X|L]).
```

Predicado in

```
in(X,[X|_]) :- !.
in(X,[_|L]) :- in(X,L).
```

Predicados Findall, Forall, Between e Maplist

- Existem no Prolog, também, predicados similares aos comandos de repetição de uma linguagem procedural: forall, findall, between e maplist.
  - **Findall** é um predicado pré-definido de prolog gera uma lista com todas as alternativas encontradas que satisfaça a uma consulta.
  - O **forall**(CONTROLE, DO) imita um comando for; o parâmetro CONTROLE indica quantas vezes ele deve repetir, o executando o segundo parâmetro.
  - O **between**(INIC,FIM,I) retorna por retrocesso todos os valores da faixa declarada. A combinação forall e between imita um comando for.
  - O predicado **maplist** mapeia um predicado, no exemplo seguinte, plus/3, para todos os elementos de uma lista, retornando uma nova lista.

Predicados Findall, Forall, Between e Maplist

Exemplos de uso

```
?- between(1,3,X).
X = 1 ; X = 2 ; X = 3 ; No
?- forall(member(X,[1,2,3]),write(X)).
123
?- forall(between(1,5,I),(write(I*I),write(' '))).
1*1 2*2 3*3 4*4 5*5
?- forall(between(10,20,I),write(I:' ')).
10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20:
?- findall(Y*Y,(between(1,5,Y)),L).
L = [1*1, 2*2, 3*3, 4*4, 5*5]
?- plus(4,5,X).
X = 9
?- maplist(plus(2),[1,2,3],L).
L = [3, 4, 5]
```

maplist

```
maplist(G, [X_11, ..., X_1n],
           [X_21, ..., X_2n],
           ...,
           [X_m1, ..., X_mn]) :-
call(G, X_11, ..., X_m1),
call(G, X_12, ..., X_m2),
...
call(G, X_1n, ..., X_mn).
```

```
vez(X,Y,Z) :- Z is X * Y.
% ?- maplist(vez,[1,3,4],[4,5,6],L).
% L = [4, 15, 24].
comp([],0).
comp([_|R],N1):-comp(R,N), N1 is N + 1.
% ?- maplist(comp,[[3,4,5],[1,2,6,5]],L).
% Gera solucoes:
% ?- length(L, 3), maplist(between(0, 1), L).
```

call

```
comp([],0).
comp([_|R],N1):-comp(R,N), N1 is N + 1.

soma([],0).
soma([C|R],S1):-soma(R,S), S1 is S + C.

teste(X,Y,Z) :-
    call(X,Y,Z).

/*
* exemplos:
* ?- teste(comp,[3,4],X).
* X = 2
* ?- teste(soma,[3,4],X).
* X = 7
*/
```

Outros predicados

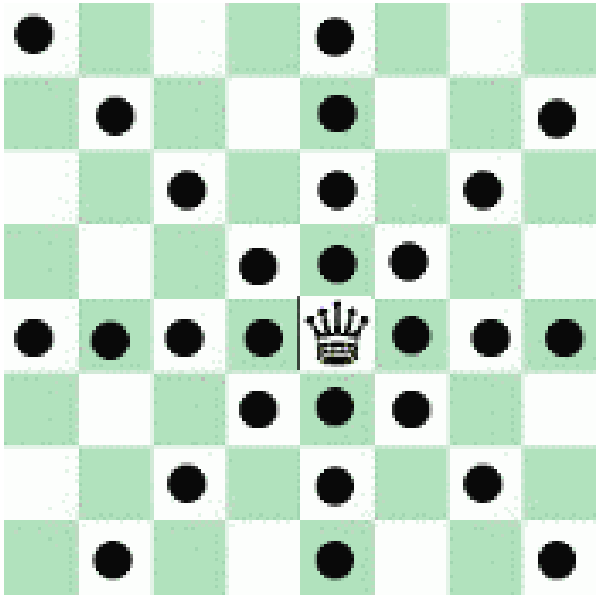
A partir do predicado ap podemos definir outros predicados como:

```
ap([],L,L).
ap([A|B], C, [A|D]) :- ap(B,C,D).
membro(X,L) :- ap(_,[X|_], L).
ultimo(U,L) :- ap(_, [U], L).
prefixo(P,L) :- ap(P,_,L).
sufixo(S,L) :- ap(_,S,L).
sublista(S,L) :- prefixo(P,L), sufixo(S,P).
apaga(A,[A|B],B).
apaga(A,[B|C],[B|D]) :- apaga(A,C,D).
permuta(B,[A|C]):-apaga(A,B,D), permuta(D,C).
permuta([],[]).
```

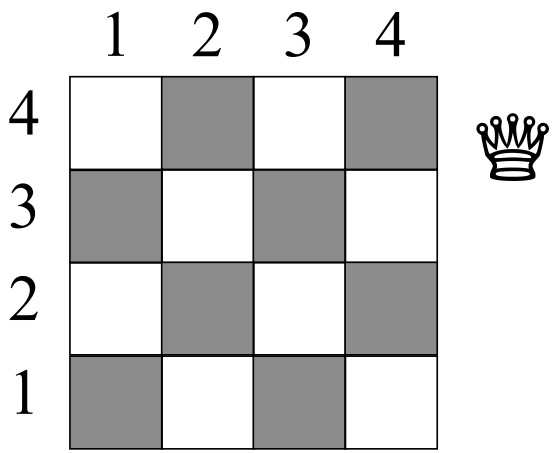
O problema das oito rainhas

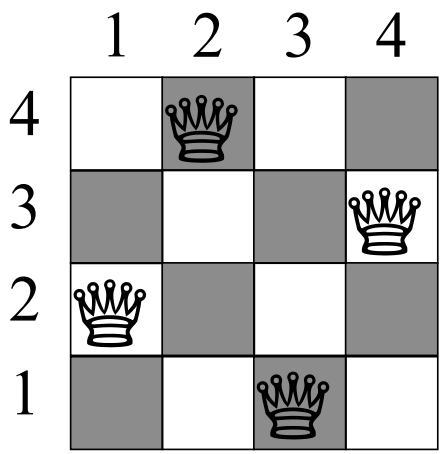
O problema

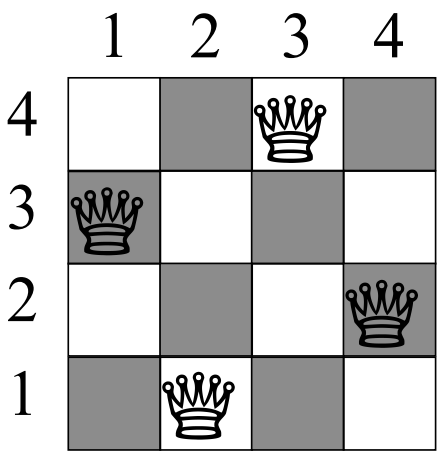
O problema clássico em inteligência artificial, chamado de problema das 8-rainhas, consiste em posicionar 8 rainhas de xadrez num tabuleiro 8x8 de modo que uma rainha não possa capturar outra. Lembrando que uma rainha captura outra se ambas estiverem numa mesma diagonal, linha ou coluna do tabuleiro.



Solução para 4-Rainhas

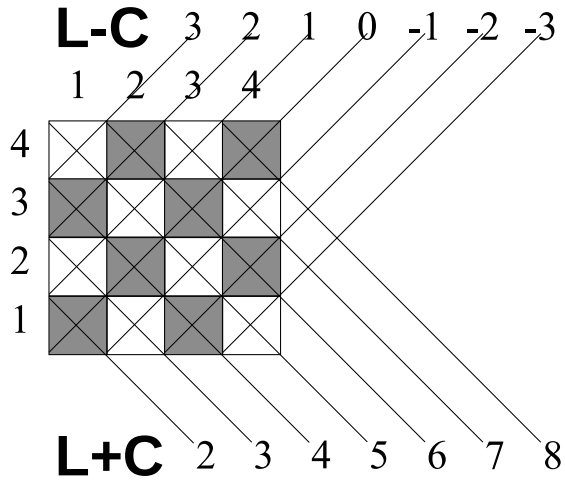




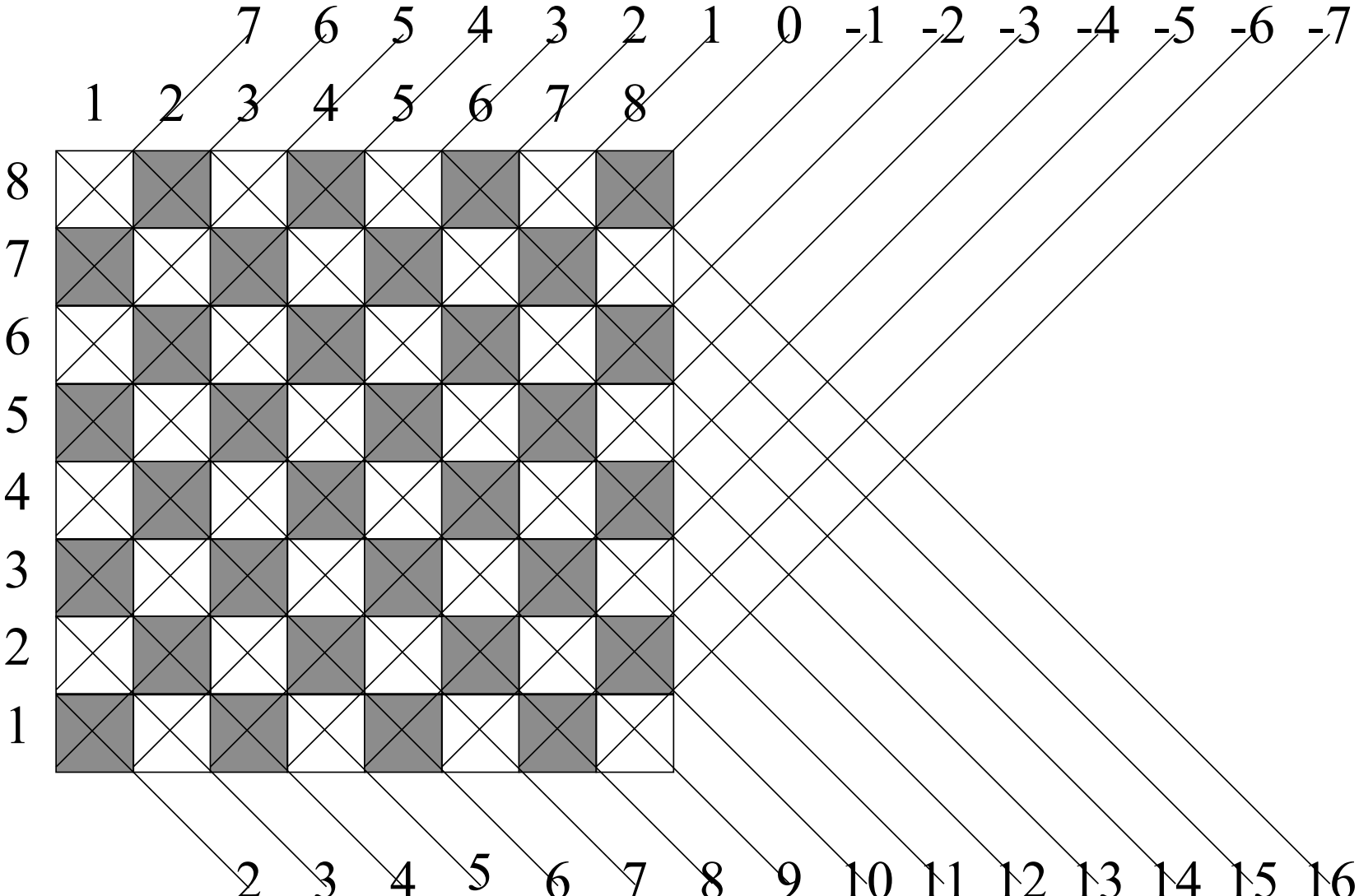


Solução para 4-rainhas

```
solucao(S) :-
  resolve(S, [1,2,3,4],[1,2,3,4],
             [3,2,1,0,-1,-2,-3],
             [2,3,4,5,6,7,8]).
resolve([], [], _, _, _).
resolve([C|LC],[L|LL],CO,DS,DI):-
  apaga(C,CO,CO1), NS is L - C,
  apaga(NS,DS,DS1),NI is L + C,
  apaga(NI,DI,DI1),
  resolve(LC,LL,CO1,DS1,DI1).
apaga(A,[A|B],B).
apaga(X,[Y|Z],[Y|Z1]) :-
  apaga(X,Z,Z1).
```



Solução para 8-Rainhas



Exercício

Exercício 1

Generalizar o problema para n-Rainhas, onde n é um parâmetro adicional do predicado solução.

```
?-solucao(4,S)
S = [3,1,4,2];
S = [2,4,1,3]
```

Exercício 2

Modifique o código para apresentar as soluções em modo texto, conforme o exemplo abaixo.

```
?-solucao(4,S)
S = [3,1,4,2];
. R . .
. . . R
R . . .
. . R .
S = [2,4,1,3]
. . R .
R . . .
. . . R
. R . .
```

Estrutura Sequencial

É possível, usando os recursos de Prolog, organizar o código em estrutura similares a de outras linguagens de programação.

```
1  prog :-
2    ledado(D),
3    calcula(D,R),
4    escreve(R).
5
6  ledado(X) :-
7    write('Digite um valor: '),
8    read(X).
9
10 calcula(D,R) :- R is D * D.
11
12 escreve(X) :-
13   write("O quadrado eh "),
14   write(X), nl.
```

Estrutura de Seleção

- Uma regra, por si só, já é uma estrutura de seleção na forma (<então> if <condições>). Mas é possivel usar uma estrutura de seleção (if-then-else) dentro de uma regra Prolog, usando a seguinte estutura:

```
1      ( condição -> então ; senão )
```

```
1 menor(X, Y, M) :-
2       ( X < Y  ->  M = X  ;   M = Y ).
3
4 % Exemplo de uso:
5 % ?- menor(3,20,X).
6 % X = 3
```

Estrutura de Repetição

- O predicado '**repeat**' sempre tem sucesso e reinicia a consulta.
- O predicado '**fail**' é usado para gerar uma falha, forçando o retrocesso e a tentativa de uma outra solução para a consulta.
- Pode-se simular um repetição tradicional usando esses predicados, conforme o seguinte exemplo:

```
1 repetir :-
2     repeat,
3         % colocar aqui o que se deseja repetir
4     write('Continuar? [sim/nao] '),
5     read(Resposta),
6     ( Resposta == nao -> ! ; fail ).
```

Agenda

Fatos e regras dinâmicas

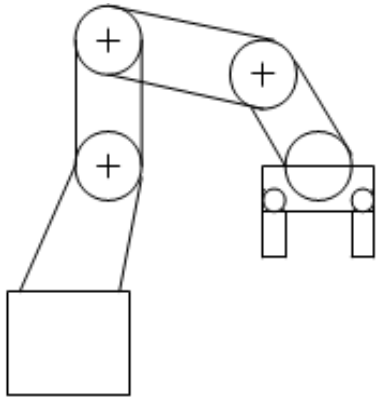
Em Prolog, podemos incluir e excluir fatos e regras, dinamicamente. Para isso, usamos os seguintes predicados:

- :-**dynamic(Functor/Aridade)** - declara que o predicado Functor/Aridade pode ser modificado durante a execução do programa.
- **assert(Clausula)** - armazena a cláusula passada no fim da lista de cláusulas associada ao predicado.
- **asserta(Clausula)** - armazena no início da lista de cláusulas.
- **retract(Clausula)** - remove uma cláusula da base de fatos.
- **abolish(Functor/Aridade)** - remove todas as cláusulas definidos para o predicado Functor/Aridade.

Exemplo de fatos dinâmicos

```
1  :-dynamic(fato/1).
2  cadastra([]).
3  cadastra([A|B]) :- assert(fato(A)), cadastra(B).
4
5  apaga(A) :- retract(fato(A)).
6
7  listafatos :-
8  fato(X),
9  writeln(X),
10 fail.
11 listafatos.
12
13 limpa :- abolish(fato/1).
```

Agenda



2	7	1
4	3	8
6	5	

Programação Lógica

Estruturas de Dados

Luiz Eduardo da Silva

Universidade Federal de Alfenas

Pilha

```
empilha(A,B,[A|B]) :- !.
desempilha(A,[A|B],B) :- !.

empLista([A|B],X,Z) :- empilha(A,X,Y), !, empLista(B,Y,Z).
empLista([],A,A) :- !.
```

Fila

```
insere(A,B,F) :- append(B,[A],F),!.
retira(A,[A|B],B) :- !.

insLista([A|B],X,Z) :- insere(A,X,Y), !, insLista(B,Y,Z).
insLista([],A,A) :- !.
```

Árvore Binária

```
% Uma representação de árvore em Prolog
% no(raiz, esq, dir)
no(a,b,c).
no(b,d,[]).
no(d,[],[]).
no(c,e,f).
no(e,[],g).
no(f,[],[]).
no(g,[],[]).

preordem([]).
preordem(X) :-
  write(X), write(' '), no(X,E,D),
  preordem(E),
  preordem(D).
```

Árvore Binária

```
%Outra representação para árvore
insere(X,[],no(X,[],[])) :- !.
insere(X,no(X,E,D),no(X,E,D)) :- !.
insere(X,no(I,E,D),no(I,E1,D)) :- X < I, !, insere(X,E,E1).
insere(X,no(I,E,D),no(I,E,D1)) :- X > I, !, insere(X,D,D1).

insLista([A|B],X-Z) :- !, insere(A,X,Y), insLista(B,Y-Z).
insLista([],A-A) :- !.

emordem([]).
emordem(no(I,E,D)) :- emordem(E), write(I), nl, emordem(D).
```

Exercícios

Exercício 1

Faça os caminhamentos inOrdem2/2 e posOrdem2/2 para retornarem uma lista com os nós da árvore.

Exercício 2

Modifique o caminhamento inOrdem/1 para escrever somente os nós-folha da árvore, os que não têm filhos.

Exercício 3

Usando base de fatos dinâmica e as estrutura de seleção e repetição apresentadas anteriormente, desenvolva um programa prolog para inserir fatos que representam os nós da árvore (no(Info,Esq,Dir)) (conforme o primeiro exemplo de árvore binária). O programa deverá ainda ler uma opção (1-inserir, 2-apagar, 3-pre-ordem, 4-em-ordem, 5-pos-ordem e 6-fim) e se for necessário um valor, para fazer a manutenção de uma árvore binária de busca.

Grafo

```
arco_(d,c).
arco_(d,e).
arco_(c,b).
arco_(f,e).
arco_(e,b).
arco_(b,a).
arco(X,Y):-arco_(X,Y);arco_(Y,X).
%%
%setof/3, coleta todos os valores de X para o predicado nodo
nodo(X):-arco_(X,_);arco(_,X).
nodos(L):-setof(X,nodo(X),L).
%%
cam0(X,Y,_):-arco(X,Y).
cam0(X,Z,C):-arco(X,Y), \+ member(Y,C), cam0(Y,Z,[Y|C]).
% ?- nodos(L).
% L = [a,b,c,d,e,f]
% ?- cam0(f,a,[]).
% yes
```

Grafo

```
(...)
caminho(X,Y,Co):-caminho(X,Y,[X],Co).
caminho(X,Y,C,[Y|C]):-arco(X,Y).
caminho(X,Z,C,Co):-
   arco(X,Y),
   \+ member(Y,C),
   caminho(Y,Z,[Y|C],Co).

%?- caminho(f,a,L).
%L = [a, b, e, f] ;
%L = [a, b, c, d, e, f] ;
%false.
```

Definição

Definição

Classificar ou ordenar um conjunto de dados consiste em receber como entrada uma sequência de n valores $< a_1, a_2, ..., a_n >$ e produzir como resultado uma permutação (reordenamento) dos valores de entrada $< a'_1, a'_2, ..., a'_n >$ de tal forma que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Métodos

- São três as classes gerais para classificação interna de dados: ordenação por **troca**, por **seleção** e por **inserção**.
- Para simulação suponha o problema de ordenação das cartas do baralho.

Por Inserção Direta

Por inserção: Segure todas as cartas na sua mão. Ponha uma carta por vez na mesa, sempre inserindo na posição correta. O maço estará ordenado quando não restarem mais cartas em sua mão.

```
insereOrd(X,[Y|L], [X,Y|L]) :-
   X =< Y, !.
insereOrd(X,[Y|L], [Y|Io]) :-
   X > Y, !, insereOrd(X,L,Io).
insereOrd(X,[],[X]).

insercao([C|Ls],So) :- insercao(Ls,Si), insereOrd(C,Si,So).
insercao([],[]).
```

Por Seleção

Por seleção: Espalhe as cartas na mesa, selecione a carta de menor valor, retire-a do baralho e segure-a na sua mão. Este processo continua até que todas as cartas estejam na sua mão.

```
selecao(L,[M|S]) :-
   removeMin(M,L,Lo), selecao(Lo,S), !.
selecao([],[]).

min([X],X).
min([X|Xs],M) :-
   min(Xs,M1),(X < M1 -> M = X ; M = M1), !.

removeMin(M,L,Lo) :- min(L,M), select(M,L,Lo).
```

Por Troca

Por troca: Espalhe as cartas numa mesa voltadas para cima e então troque as cartas de ordem até que todo o baralho esteja ordenado.

```
troca(L,S) :-
    append(Ord,[A,B|Ls],L), B < A, !,
    append(Ord,[B,A|Ls],Li),
    troca(Li,S).
    troca(L,L).
```

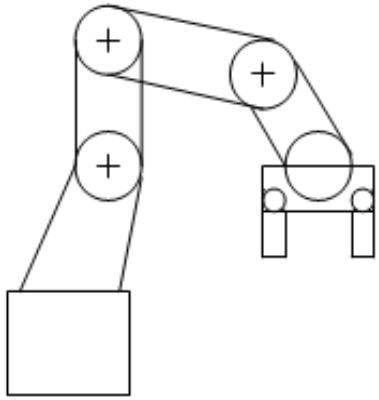
QuickSort

```
% quickSort = particao e troca

particao([X|L], Pivo, [X|Menores], Maiores) :-
    X < Pivo, !, particao(L,Pivo, Menores, Maiores).
particao([X|L], Pivo, Menores,[X|Maiores]) :-
    X >= Pivo, !, particao(L,Pivo, Menores, Maiores).
particao([],_,[],[]).

quickSort([X|Xs], S) :-
    particao(Xs,X,Me,Ma),
    quickSort(Me, SMe),
    quickSort(Ma, SMa),
    append(SMe, [X|SMa], S).
quickSort([],[]).
```

Agenda



2	7	1
4	3	8
6	5	

Programação Lógica

Busca

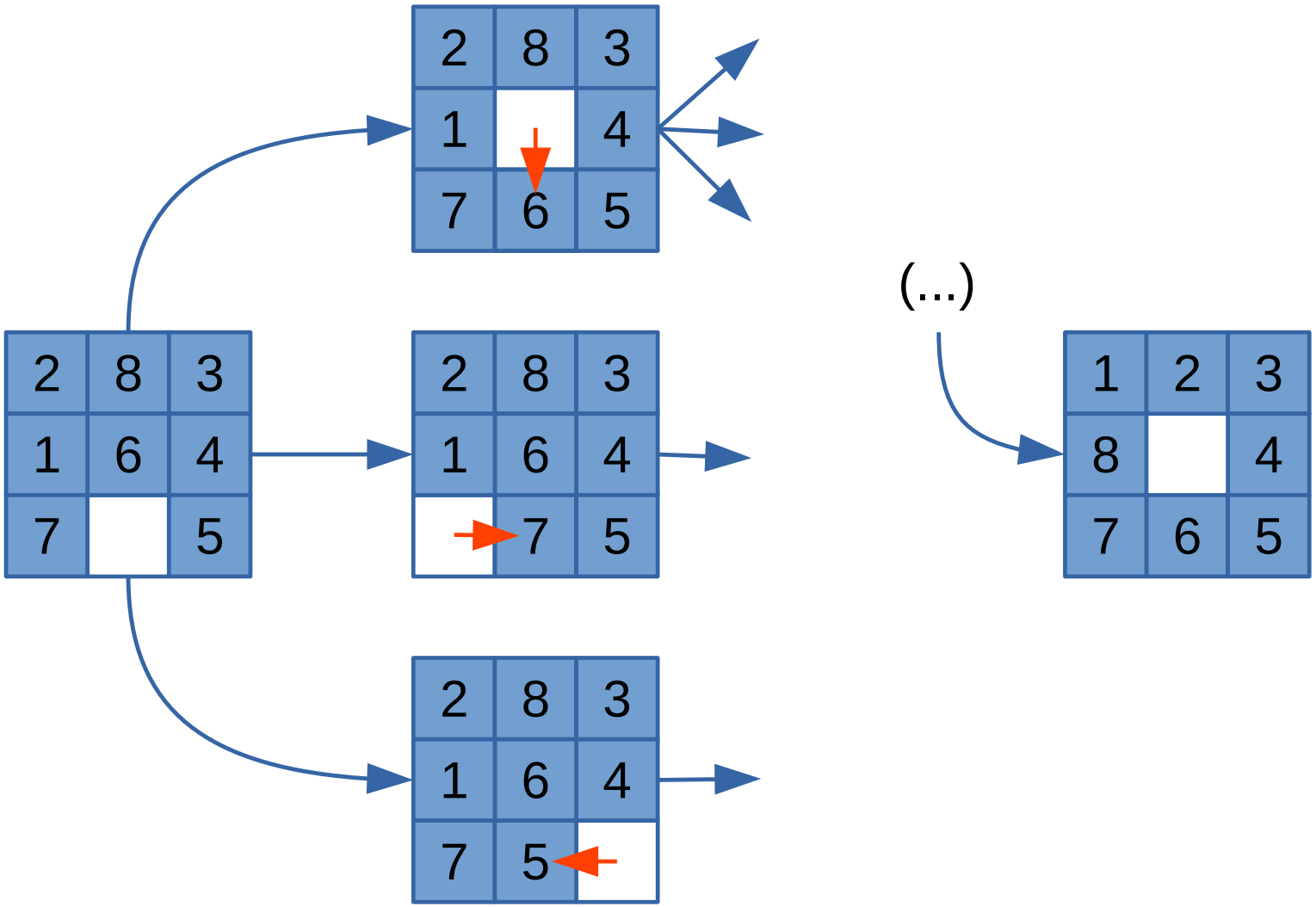
Luiz Eduardo da Silva

Universidade Federal de Alfenas

Introdução

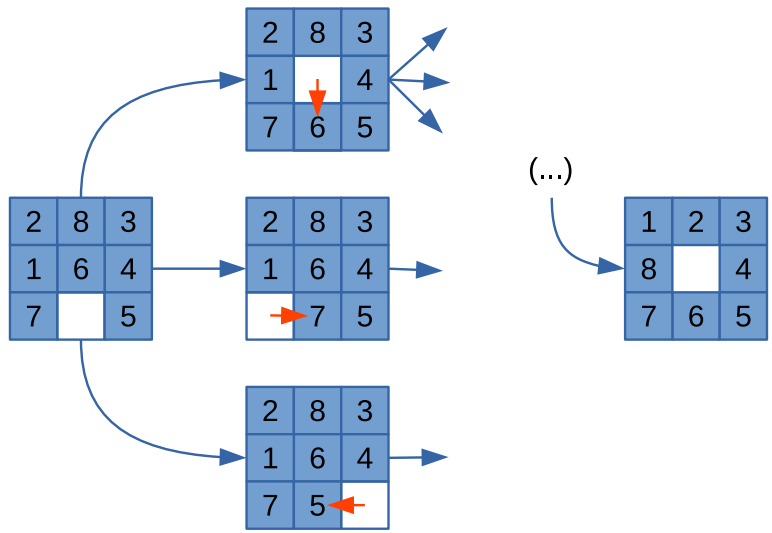
- Uma das características de ser inteligente é a capacidade de resolver problemas.
- Para dotar os computadores de inteligência, precisamos escrever programas que tenham a habilidade de procurar por soluções de problemas.
- *Busca no espaço de estados* é uma das técnicas para resolução de problemas em inteligência artificial.
- Seja um agente inteligente uma entidade capaz de executar ações(transformações) que modificam o estado corrente de seu mundo
- A partir de um estado inicial, a sequência de transformações que são realizadas para atingir um estado meta, determinam a solução do problema.

Espaço de estados

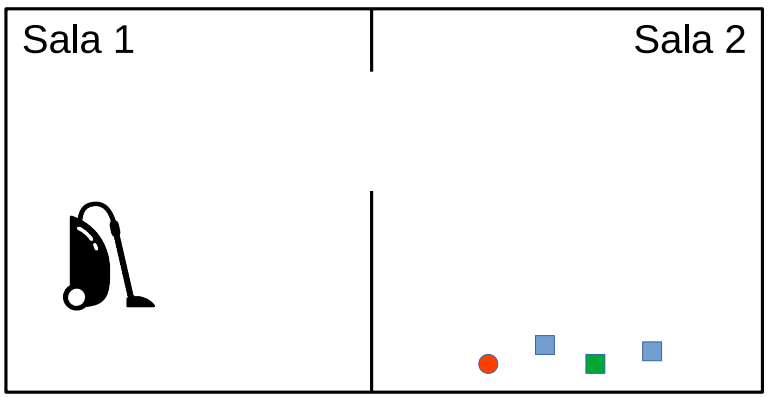


Espaço de estados

- **Estado**: situação em que os objetos se encontram em dado instante.
- **Estado Inicial**: situação dos objetos no início do problema.
- **Estado Final (meta)**: situação na qual se deseja por os objetos. Como devem estar os objetos para que o problema seja resolvido.
- **Ações**: Transformação de um estado em outro.



Mundo do aspirador



- **agente**: é um aspirador, cuja função é limpar as salas de uma casa (nesse exemplo simples, temos duas salas: sala 1 e sala 2)
- **Ações**: São as transformações que o agente sabe realizar. Nesse caso, suponha que as ações sejam:
  - entrarSala1
  - entrarSala2
  - aspirar

Representação dos estados

- A representação de estado é uma estrutura que representa um estado do problema.
- Para o problema do mundo do aspirador, o estado pode ser [X,Y,Z], onde:
  - $X \in \{1, 2\}$ indica a posição do aspirador
  - $Y \in \{0, 1\}$ indica se a primeira sala está suja
  - $Z \in \{0, 1\}$ indica se a segunda sala está suja
- Se o aspirador está na segunda sala e somente essa está suja é representada por [2,0,1].
- O conjunto de todos os estados do mundo do aspirador é:

$$\begin{aligned} S \quad = \quad & \{[1,0,0],[1,0,1],[1,1,0],[1,1,1], \\ & [2,0,0],[2,0,1],[2,1,0],[2,1,1]\} \end{aligned}$$

Representação das ações

- As ações são representadas por transformações na forma:

$$oper(\alpha, s, s') \leftarrow \beta$$

onde α é uma ação de transforma o estado s no estado s', dado que a condição β é satifeita.

- A ação *aspirar* pode ser representada por:

$$oper(aspirar, [1, Y, Z], [1, 0, Z]) \leftarrow Y = 1$$
$$oper(aspirar, [2, Y, Z], [2, Y, 0]) \leftarrow Z = 1$$

- o conjunto de ações para o mundo do aspirador pode ser representado por:

$$\begin{aligned} A = \{ & oper(entrarSala1, [2, Y, Z], [1, Y, Z]), \\ & oper(entrarSala2, [1, Y, Z], [2, Y, Z]), \\ & oper(aspirar, [1, 1, Z], [1, 0, Z]), \\ & oper(aspirar, [2, Y, 1], [2, Y, 0])\} \end{aligned}$$

Espaços sucessores

- Espaços sucessores são os estados que podem ser gerados a partir de um estado pela aplicação de uma ação.
- Exemplo: para o estado [2,0,1], temos os seguintes estados sucessores: [1,0,1], [2,0,0], que são gerados pela aplicação das ações *entrarSala1* e *aspirar*, respectivamente.

Exercício 1

Desenhe um grafo representando o espaço de estados para o mundo do aspirador. Nesse grafo, cada nó (vértice) será um estado do mundo e cada arco (rotulado com uma ação), será a transição entre dois estados. Os arcos devem ser direcionados do estado para o seu estado sucessor.

Problema de busca

Um problema de busca é especificado por três componentes:

- um espaço de estados (denotado pelos conjuntos S e A).
- um estado inicial (denotado por $s_0 \in S$) e
- um conjunto de estados finais (denotado por $G \subset S$).

Para o problema do mundo do aspirador:

- S e A conforme descrito anteriormente
- $s_0 = [1, 1, 1]$
- $G = \{[1, 0, 0], [2, 0, 0]\}$

Exercício 2

Encontre no grafo do exercício 1, duas soluções possíveis para o problema do mundo do aspirador.

Algoritmo não determinístico

BUSCA(A,s_O,G)

```
1  Σ ← {s0}
2  enquanto Σ ≠ ∅ faça
3      s ← remove(Σ)
4      se s ∈ G então retorne caminho(s)
5      Σ ← Σ ∪ sucessores(s, A)
6  retorne fracasso
```

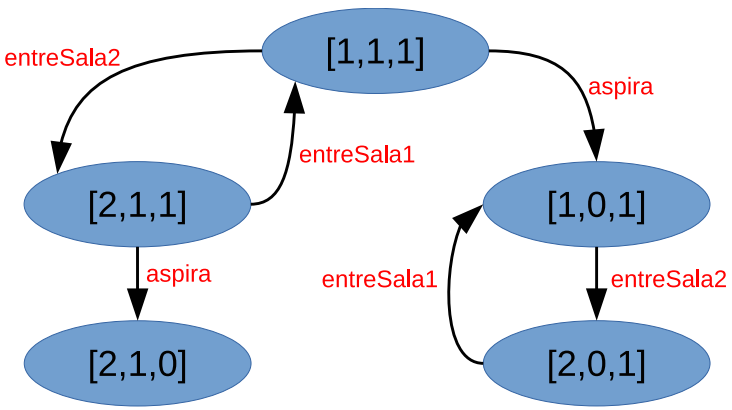
onde:

- *remove*(Σ) - seleciona um estado aleatório de *Sigma*
- *caminho*(s) - retorna a sequência de ações que rotulam o caminho de s_0 até s
- *sucessores*(s, A) - retorna o conjunto de estados sucessores de S.

Algoritmo não determinístico

BUSCA(A,s_O,G)

1 $\Sigma \leftarrow \{s_0\}$
2 enquanto $\Sigma \neq \emptyset$ faça
3 $s \leftarrow remove(\Sigma)$
4 se $s \in G$
então retorne $caminho(s)$
5 $\Sigma \leftarrow \Sigma \cup sucessores(s, A)$
6 retorne fracasso



- Seja $s_0 = [1, 1, 1]$ e $G = \{[1, 0, 0], [2, 0, 0]\}$
- Passo 1:
  - $s \leftarrow [1, 1, 1]$, $\Sigma = \emptyset$.
  - $s \notin G$
  - $\Sigma \leftarrow \{[2, 1, 1], [1, 0, 1]\}$
- Passo 2:
  - $s \leftarrow [2, 1, 1]$ ou $s \leftarrow [1, 0, 1]$.
  - $s \notin G$
  - $\Sigma \leftarrow \{[1, 0, 1], [2, 1, 0], [1, 1, 1]\}$ ou $\Sigma \leftarrow \{[2, 1, 1], [2, 0, 1]\}$
- ...

Detecção de ciclos

- Um problema da busca não-determinística é que a busca pode executar infinitamente se os estados formam ciclos.
- Para corrigir, podemos guardar os estados já visitados (Γ), para impedir que sejam utilizados novamente.

BUSCA'(A,s_O,G)

1 $\Gamma \leftarrow \emptyset$
2 $\Sigma \leftarrow \{s_0\}$
3 enquanto $\Sigma \neq \emptyset$ faça
4 $\quad s \leftarrow remove(\Sigma)$
5 $\quad$ se $s \in G$
$\quad\quad$ então retorne $caminho(s)$
6 $\quad \Gamma \leftarrow \Gamma \cup \{s\}$
7 $\quad \Sigma \leftarrow \Sigma \cup (sucessores(s, A) - \Gamma)$
8 retorne fracasso

Busca em Largura

Dentre as estratégias de busca cega (determinística) temos a busca em largura primeiro, que coloca os estados expandidos no final de uma FILA de estados candidatos.

BUSCALARGURA(A,s_O,G)

1 $\Gamma \leftarrow \emptyset$
2 $\Sigma \leftarrow \{s_0\}$
3 enquanto $\Sigma \neq \emptyset$ faça
4 $\quad s \leftarrow removePrimeiro(\Sigma)$
5 $\quad$ se $s \in G$
$\quad\quad$ então retorne $caminho(s)$
6 $\quad \Gamma \leftarrow \Gamma \cup \{s\}$
7 $\quad insereNoFinal(sucessores(s, A) - \Gamma, \Sigma)$
8 retorne fracasso

Problema

Problema dos Jarros

"Há dois jarros com capacidade de 3 e 4 litros, respectivamente. Nenhum dos jarros contém qualquer medida ou escala, de forma que só se pode saber o conteúdo exato quando eles estão cheios. Sabendo-se que podemos encher ou esvaziar um jarro, bem como transferir água de um jarro para outro, encontre uma sequência de passos que deixe o jarro de quatro litros com exatamente 2 litros de água [PEREIRA, S.L.]".

- Represente os estados usando um par $[X, Y]$, onde $X \in \{0, 1, 2, 3\}$, representa o conteúdo do primeiro jarro e $Y \in \{0, 1, 2, 3, 4\}$, representa o conteúdo do segundo jarro.

Operações (ações)

$$
\begin{array}{ll}
oper(enche1, [X, Y], [3, Y]) & \leftarrow X < 3 \\
oper(enche2, [X, Y], [X, 4]) & \leftarrow Y < 4 \\
oper(esvazia1, [X, Y], [0, Y]) & \leftarrow X > 0 \\
oper(esvazia2, [X, Y], [X, 0]) & \leftarrow Y > 0 \\
oper(despeja1em2, [X, Y], [0, X + Y]) & \leftarrow X > 0, \\
 & Y < 4, X + Y \leq 4 \\
oper(despeja1em2, [X, Y], [X + Y - 4, 4]) & \leftarrow X > 0, \\
 & Y < 4, X + Y > 4 \\
oper(despeja2em1, [X, Y], [X + Y, 0]) & \leftarrow X < 3, \\
 & Y > 0, X + Y \leq 3 \\
oper(despeja2em1, [X, Y], [3, X + Y - 3]) & \leftarrow X < 3, \\
 & Y > 0, X + Y > 3
\end{array}
$$

Exercício

- Seja o estado inicial $s_0 = [0, 0]$ e o conjunto de estados finais $G = \{[0, 2], [1, 2], [2, 2], [3, 2]\}$
- Desenhe a árvore de busca gerada pelo algoritmo de Busca em Largura, conforme especificado, conforme o algoritmo:

BUSCALARGURA(A,s_O,G)

```
1  Γ ← ∅
2  Σ ← {s0}
3  enquanto Σ ≠ ∅ faça
4     s ← removePrimeiro(Σ)
5     se s ∈ G
          então retorne caminho(s)
6     Γ ← Γ ∪ {s}
7     insereNoFinal(sucessores(s, A) − Γ, Σ)
8  retorne fracasso
```

Busca em Profundidade

A busca em profundidade primeiro, diferentemente coloca os estados expandidos no início de uma PILHA de estados candidatos.

BUSCAPROFUNDIDADE(A,s_O,G)

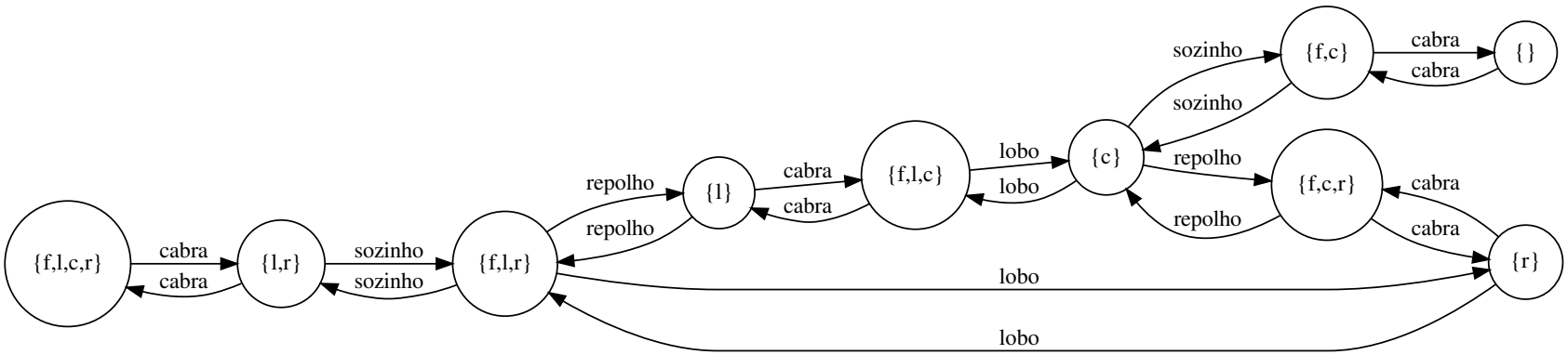
1 $\Gamma \leftarrow \emptyset$
2 $\Sigma \leftarrow \{s_0\}$
3 enquanto $\Sigma \neq \emptyset$ faça
4 $\quad s \leftarrow removePrimeiro(\Sigma)$
5 $\quad$ se $s \in G$
$\quad\quad$ então retorne $caminho(s)$
6 $\quad \Gamma \leftarrow \Gamma \cup \{s\}$
7 $\quad insereNoInicio(sucessores(s, A) - \Gamma, \Sigma)$
8 retorne fracasso

Problema do fazendeiro

Problema

"Um fazendeiro encontra-se na margem esquerda de um rio, levando consigo um lobo, uma ovelha e um repolho. O fazendeiro precisa atingir a outra margem do rio com toda a sua carga intacta, mas para isso dispõe somente de um pequeno bote com capacidade para levar apenas ele mesmo e mais uma de suas cargas. O fazendeiro poderia cruzar o rio quantas vezes fossem necessárias para transportar seus pertences, mas o problema é que, na ausência do fazendeiro, o lobo pode comer a ovelha e essa, por sua vez, pode comer o repolho. Encontre uma sequência de passos que resolva esse problema".

Grafo - Fazendeiro



Uma solução

```
Inicio                        : FLCR|      |
fazendeiro leva a cabra       :  L R|      |F C
fazendeiro volta sozinho      : FL R|      |  C
fazendeiro leva o lobo        :    R|      |FLC
fazendeiro traz a cabra       : F CR|      | L
fazendeiro leva o repolho     :   C |      |FL R
fazendeiro volta sozinho      : F C |      | L R
fazendeiro leva a cabra       :     |      |FLCR
```

Outro problemas dos jarros

Problema

"Há três jarros com capacidades de 8, 5 e 3 litros, respectivamente. O jarro de 8 litros está cheio de água. Sabendo-se que podemos transferir o conteúdo de um jarro para outro, encontre a sequência de operações para deixar os jarros de 8 e 5 litros com exatamente 4 litros de água cada."

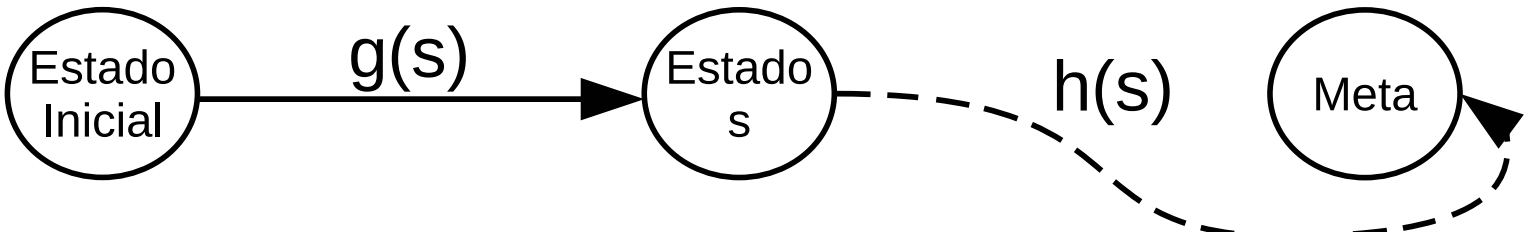
Problema dos Missionários e Canibais

Problema

"Temos três missionários e três canibais que precisam atravessar um rio com um barco que tem a capacidade de transportar, no máximo, duas pessoas. A restrição é que não podemos ter um número maior de canibais do que missionários em qualquer margem, senão os missionários serão devorados. Ao menos uma pessoa deve transportar o barco de uma margem para outra. Encontre a sequência de movimentos que resolva o problema."

Busca Heurística

- **Buscas Cegas** (largura, profundidade), encontram soluções testando e expandindo estados, mas não garantem soluções de custo mínimo.
- **Buscas Heurísticas**, utiliza um conhecimento do problema, para explorar de forma mais eficiente os estados e assim, encontrar uma solução de custo mínimo.
  - Busca pelo custo da ação - ordena as expansões considerando o custo de ir do estado inicial ao estado corrente. $g(s)$
  - Busca Heurística - ordena as expansões de acordo com a estimativa de ir do estado corrente ao estado final. $h(s)$
  - Busca A^* - combina as duas anteriores. $f(s) = g(s) + h(s)$



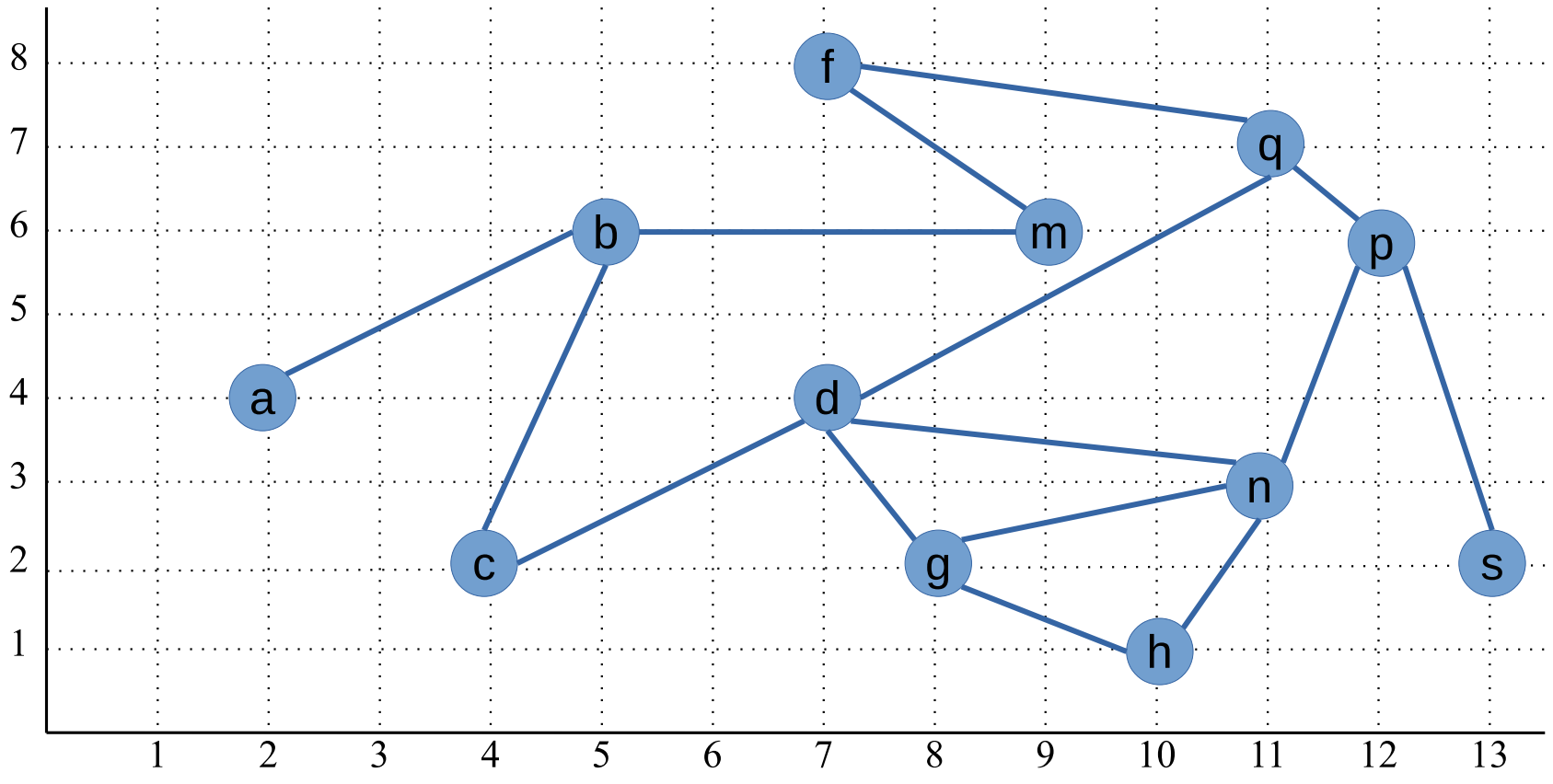
Busca Heurística

BUSCAAestrela(A,s_O,G)

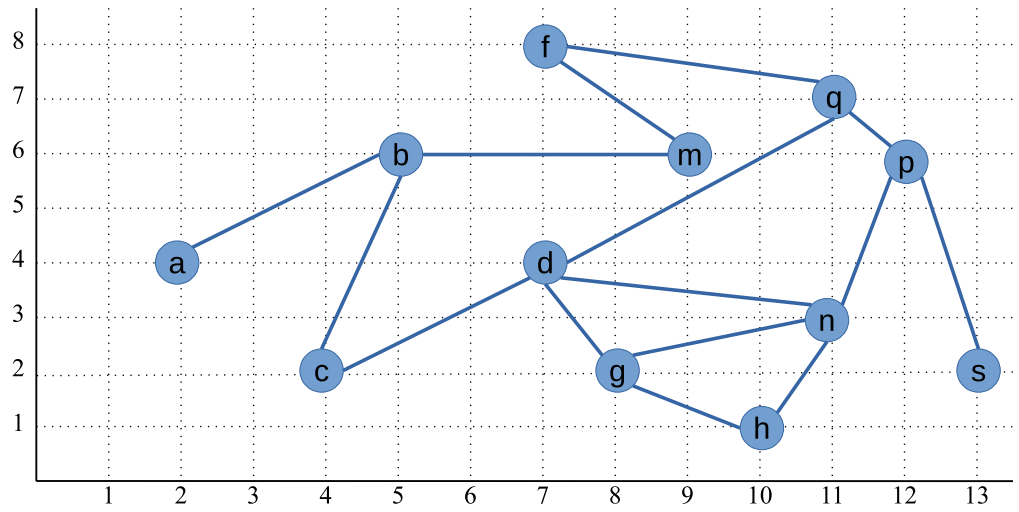
1 $\Gamma \leftarrow \emptyset$
2 $\Sigma \leftarrow \{s_0\}$
3 enquanto $\Sigma \neq \emptyset$ faça
4 $\quad s \leftarrow removePrimeiro(\Sigma)$
5 $\quad$ se $s \in G$
$\quad\quad$ então retorne $caminho(s)$
6 $\quad \Gamma \leftarrow \Gamma \cup \{s\}$
7 $\quad insereEmOrdem(sucessoresF(s, A) - \Gamma, \Sigma)$
8 retorne fracasso

A função *sucessoresF* devolve a lista de estados sucessores a soma do custo real e da estimativa heurística $g(s) + h(s)$.

Busca Heurística



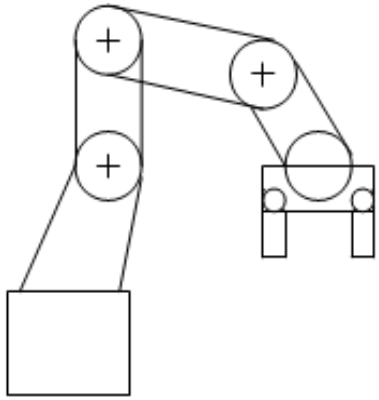
Busca Heurística




```
coord(a, 2, 4).
coord(b, 5, 6).
coord(c, 4, 2).
coord(d, 7, 4).
coord(f, 7, 8).
                ...

oper('a para b', a, b).
oper('b para m', b, m).
oper('m para f', m, f).
oper('f para q', f, q).
oper('q para p', q, p).
...
```

Agenda



2	7	1
4	3	8
6	5	

Programação Lógica

Sistemas Especialistas

Luiz Eduardo da Silva

Universidade Federal de Alfenas

Sistemas Especialistas

Sistemas Especialistas

Um sistema especialista (SE) é um programa que responde como um especialista numa determinada área de conhecimento e consegue explicar as suas decisões. Um SE é normalmente composto por:

- base de conhecimento;
- máquina de inferência;
- interface do usuário.

Sistemas Especialistas

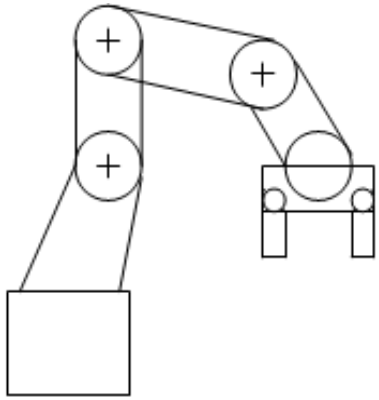
```
prove(true) :- !.
prove((B, Bs)) :- !,prove(B),prove(Bs).
prove(H) :-clause(H, B),prove(B).
prove(H) :-pergunta(H),writeln(H),read(Answer),Answer == sim.

animal_estimacao(X) :- passaro(X), pequeno(X).
animal_estimacao(X) :- fofo(X), amarelo(X).

passaro(X) :- tem_penas(X), canta(X).
amarelo(piu_piu).

pergunta(canta(_)).
pergunta(pequeno(_)).
pergunta(fofo(_)).
pergunta(tem_penas(_)).
```

Agenda



2	7	1
4	3	8
6	5	

Programação Lógica

Linguagem Natural

Luiz Eduardo da Silva

Universidade Federal de Alfenas

Gramáticas

- Prolog possui um mecanismo, conhecido como DCG (Definite Clause Grammar) que permite processar diretamente gramáticas de atributos.
- Algumas restrições são:
  - DCG processa gramáticas livres de contexto **sem recursão à esquerda**. Por exemplo: a regra $R --> Ra$, deve ser reescrita como $R --> aR$. Da mesma forma, a alternativa vazia tem que ser a última. Por exemplo: $R --> []|aR$, deve ser escrita como: $R --> aR|[]$.
- A gramática $R = a^*b^*$, usando DCG fica:

```
r --> a, b.
a --> [a],a.
a --> [].
b --> [b], b.
b --> [].
```

```
% podemos perguntar:
% r([a,b,b], []). retirna true.
% r([a,b,a], []). retorna false.
```

Gramáticas

- Os símbolos terminais são representados entre colchetes.
- Os não-terminais são representados usando letras minúsculas.
- As sentenças são representadas por lista de terminais.
- Na pergunta são passados dois parâmetros: a cadeia de entrada e a de saída.
- Na pergunta:

```
?- r([a,b,b], X).
```

o retorno X = [], significa que toda entrada foi reconhecida, caso contrário o argumento de saída é o que sobrou.

Processamento de Linguagem Natural

- PLN é uma área de IA que estudo o processamento de linguagens usadas entre seres humanos, como inglês e português, usando o computador.
- Tem diversas aplicações:
  - Tradução automática;
  - Extração automática de conhecimento;
  - Reconhecimento de voz;
  - Dicionários eletrônicos;
  - Corretores.

Processamento de Linguagem Natural

```
sentenca --> fraseNom, fraseVerbal.
fraseNom --> artigo, nome.
fraseNom --> nome.
fraseNom --> pronome.
fraseVerbal --> verbo, fraseNom.
fraseVerbal --> verbo.

nome --> [bidu]; [mimi]; [mikey].
nome --> [cao]; [gato]; [gata]; [gatas]; [rato]; [queijo].
pronome --> [ele]; [ela].
verbo --> [late]; [persegue]; [come]; [comem]; [dorme].
artigo --> [A], {member(A, [a, as, o, os, um, uns, uma, umas])}.
```