Inteligência Artificial

Resumos
2016/2017

Bárbara Jael | 73241

IA

3º ano (4ª matrícula)
1º semestre

Regente: Luís Seabra Lopes, lsl@ua.pt
Prática: João Rodrigues, jmr@ua.pt

Avaliação

- prática → 3 trabalhos individuais
  - 1º - 5% (28/09)
  - 2º - 10% (27 a 29/10)
  - 3º - 10% (8 a 10/12)
- → trabalho de grupo - 30% (15 a 25/10)
- teórica → exame final - 45%

notas! • os trabalhos individuais nr 2 e 3 têm uma duração de 48h
• o trabalho individual nr 1 e o exame final são presenciais

TP

15.09.16

Python é multi-paradigma
- funcional
- OO
- imperativa / modular

Objeto - qualquer dado que possa ser armazenado numa variável, ou passado como parâmetro

- referência
- tipo
- valor

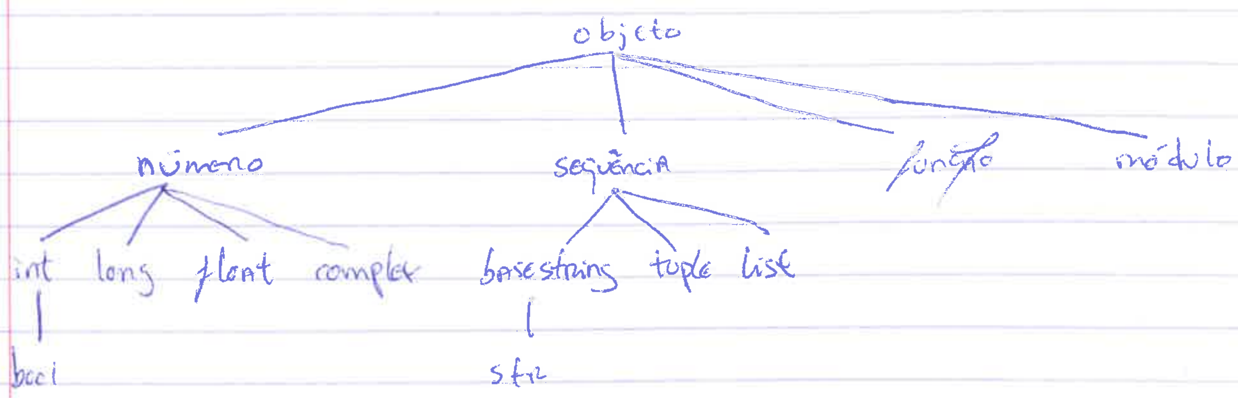
caracterização de um objeto

```
y = x if x>0 else -x
```

$\underbrace{y = \underbrace{x \text{ if } x>0 \text{ else } -x}_{\text{expressão}}}_{\text{instrução}}$

```
# inverter uma lista
def inverter (lista):
    if lista == []:
        return []
    inv = inverter (lista[1:])
    return inv + [lista[0]]
```

```
# separar os pares numa lista, produzindo um par de listas
lpares = [(1, "a"), (2, "b"), (3, "c")]
# --> ([1, 2, 3], ["a", "b", "c"])
def separar (lpares):
    if lpares == []:
        return ([], [])
```




```
x in s
x not in s
len(s)
s1 + s2  —> retorna concatenação
```
} em sequências

variáveis { não são declaradas / não têm tipo

```
triplo = (1, 2, 3)
(i, j, k) = triplo   ——> fica i=1, j=2, k=3

(q, r) = divmod(16, 3)   —> fica q=5, r=1
```

```
l = [1, 2, 3]
  l[0]    -> 1
  l[0:1]  -> [1]
  l[1:3]  -> [2, 3]
  l[1:]   -> [2, 3]
  l[:1]   -> [1]
  l[-1]   -> 3
  l[:-1]  -> [1, 2]
```

TP

22.09.16

expressões lambda: expressões cujo valor é uma função

Ex: • lambda x: x+1 ← função que dado um valor de x, devolve x+1

• m = lambda x,y: math.sqrt (x**2 + y**2)
  ↳ função que calcula o módulo de um vetor (x,y), sendo atribuída à variável m

• lambda lista = lista [-1] - lista [0]) [5,7,11,19,38]
  ↳ função que calcula a diferença entre o primeiro e o último elemento de uma lista

```
lista = [1, 2, 4, 3, -1, -12, 9, 8]
quadrado = lambda x: x**2

print lista

# função que aplica a função f a cada um dos elementos da lista
# e devolve a lista dos resultados
def aplicar (f, list):
    if list == []:
        return []
    return [f(list[0])] + aplicar (f, list[1:])

print (aplicar (quadrado, lista))

impar = lambda x: x%2 == 1

# retorna a lista dos elementos de lst que satisfazem
# a função booleana f
def filtrar (f, lst):
    if lst == []:
        return []
    if f (lst [0]):
        return [lst [0]] + filtrar (f, lst [1:])
    return filtrar (f, lst [1:]
```

```
# dada uma função f e um intervalo [a,b] calcula a raiz de f
# c/ um determinado erro máximo
def raiz (f, a, b, erro):
    (*)
    m = (a+b)/2
    if b - a < erro
        return m
    if f(a)*f(m) < 0 :
        return raiz (f, a, m, erro)
    return raiz (f, m, b, erro)

(*) if erro <= 0 or b <= a or f(a)*f(b) > 0
        return None

parabola = lambda x: x**2 - 2

print (raiz (parabola, 0, 1000, 0, 0.0001))

# retorna a redução de uma lista, dada a função de redução
# e o elemento neutro
def reduzir (lst, f, neutro):
    if lst == []:
        return neutro
    return f (lst [0], reduzir ([lst [1:], f, neutro))

# somatório dos números ímpares da "lista"
print (reduzir (filtrar (impar, lista), lambda h, r: h+r, 0))

# mix das funções aplicar e reduzir
def aplicarMix (f, list):
    return reduzir (list, lambda h, r: [f(h)] + r, [])
```

cabeça da lista ↗ ↑ redução dos ...

```
print (aplicar (quadrado, lista))
```

```
# mix das funções filtrar e reduzir
def filtrarMix(f, lst):
    return reduzir(lst, lambda h, R: [h]+R if f(h) else R, [])
```

```
print (filtrar (impar, lista))
```

```
def membro(x, lst):
    return reduzir (lst, lambda h, R: x==h or R, False)
```

```
print (membro(10, lista))
```

---

lista de compreensão: mecanismo compacto p/ processar alguns ou todos os elementos numa lista

← obter o quadrado dos elementos positivos em lista

Ex [x ** 2 for x in [3, -7, 6] if x > 0]
→ [9, 36]

TP

29.09.16

classes ⟶
```
class <nome-classe>:
    <declaração 1>
    ....
    <declaração n>
```

```
class UmTeste:
    def dizer_Ola(self):      # self é como o "this" do java
        print "Ola"
```

```
>> x = UmTeste()
>> x.dizer_Ola
```

```
class Complexo:
    def __init__ (self, real, img):
        self.R = real
        self.i = img
```

o construtor é o método que inicializa um objeto no momento da sua criação e obrigatoriamente chama-se __init__

o primeiro parâmetro de qualquer método é a própria instância na qual o método é chamado

```
>> c = Complexo (-1.5, 13.1)
>> c.R, c.i
```

tal como acontece c/ as variáveis normais, os atributos das classes ~~não~~ ~~são~~ declarados

Pré-definidos:

métodos
- __init__() — construtor
- __str__() — define uma conversão "informal" p/ cadeia de caracteres
- __repr__() — define a representação "oficial" em cadeia de caracteres

atributos
- __class__ — identifica a classe de um dado método

o tipo list de Python é uma classe (e tem vários métodos pré-definidos)

Inteligência

- capacidade de adquirir e aplicar conhecimento
- " " pensar e raciocinar
- conjunto de capacidades superiores da mente
- criatividade
- capacidade de prever passos ou acontecimentos

Segundo Albus, o estudo da inteligência envolve 7 pontos (ver slides)

Inteligência artificial

Pensar como o ser humano	Pensar racionalmente
Agir como o ser humano	Agir racionalmente

Agente – entidade c/ poder ou autoridade p/ agir
– " que atua em representação de outrem

→ entidade c/ capacidade de obter informação sobre o seu ambiente e de executar ações em função dessa info

Ex: agente físico: robô anfitrião
" de software: agente móvel de pesquisa de informação na internet

Teste de Turing

"Sala chinesa" de Searle

Tipos de agentes
- reativos simples
- " c/ estado
- deliberativos orientados por objetivos
- " " " " funções de utilidade

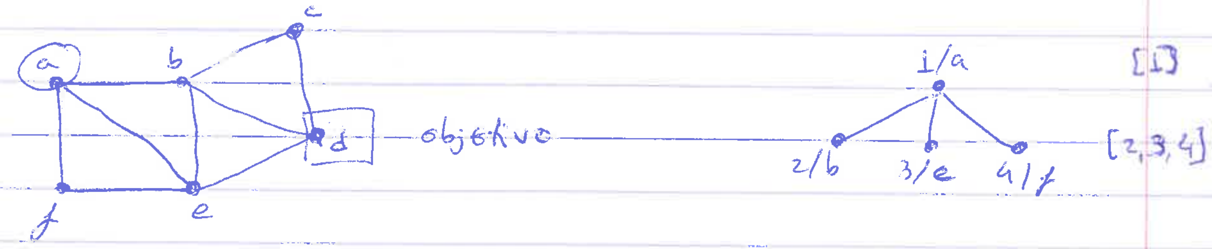
Arquiteturas
- subsunção
- 3 torres
- 3 camadas
- CARL

Propriedades do mundo de um agente
- acessibilidade
- determinismo
- mundo episódico
- dinamismo
- continuidade

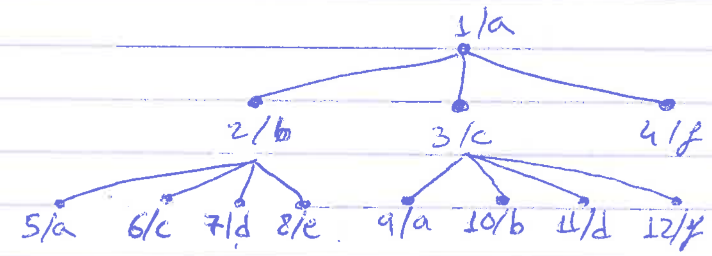
Uma lógica tem
- sintaxe
- semântica
- regras de inferência

TP 20.10.16

slide 113



Pesquisa em Largura / FIFO

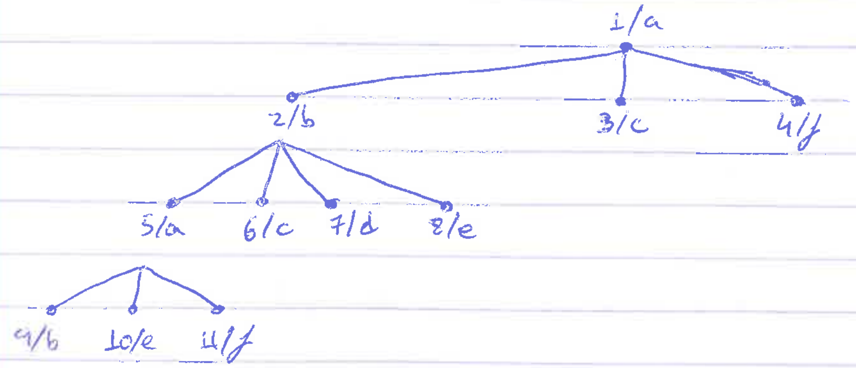


[2, 3, 4]

[3, 4, 5, 6, 7, 8]

[4, 5, 6, 7, 8, 9, 10, ~~11~~, 12]

Pesquisa em profundidade / LIFO



[2, 3, 4]

[5, 6, 7, 8, 3, 4]

[9, 10, 11, 6, 7, 8, 3, 4]

abstract → só vai ser definida mais tarde

Ex) class SearchDomain:
    def action (state):
        abstract

(como uma interface no java)

A classe que depois herdar SearchDomain é que vai implementar o método action

pesquisa em árvore → slide 115

Ex do uso: t = SearchTree (...)
t.strategy
t.problem.goal
t.problem.initial
t.problem.domain.actions()

pesquisa informada (melhor primeiro) → ordenados pelo custo e escolhe-se o de menor custo

avaliação das estratégias de pesquisa (slide 118)
- completude ← FIFO, LIFO (se não for infinito)
- complexidade temporal
- " espacial

[próxima aula]

- optimalidade ← FIFO (unidade: transições)

se a unidade for o custo, não é ótima

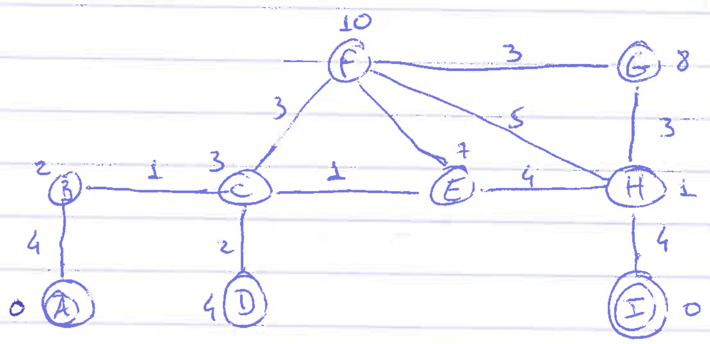
TP

27.10.16

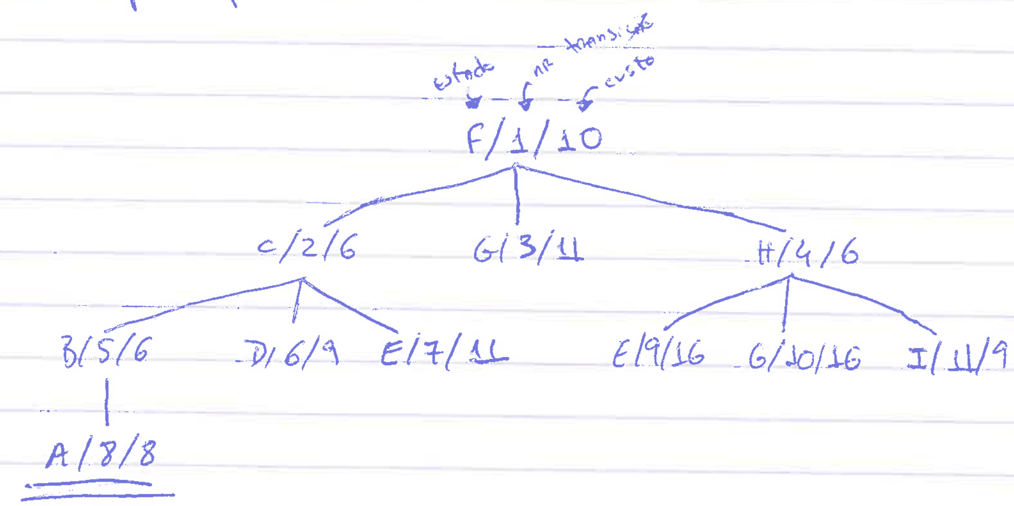
Complexidade espacial
pesquisa em largura → O(exp(n)), n = nr transições na solução
" " profundidade → O(n)

Complexidade temporal
pesquisa em largura → O(exp(n))
" " profundidade →

//



Árvore de pesquisa gerada pela estratégia A* tomando o estado principal o estado F



[1]

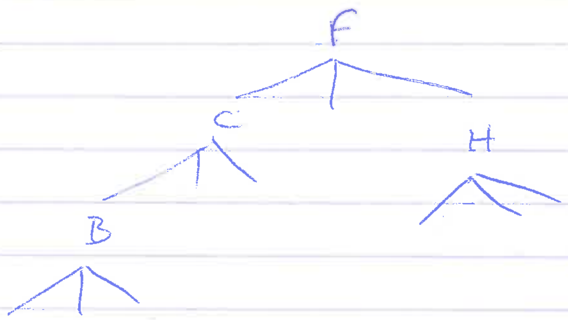
[2,4,3] ← ordem: 1) custo 2) alfabético

[5,4,6,7,3]

[4,8,6,7,3]

[8,6,11,7,3,9,10]

~~ramificação~~ média?



$$RM = \frac{N-1}{X}$$

N – total dos nós no momento em que se encontra a solução

X – nós expandidos

$$RM = \frac{11-1}{4} = 2.5$$

fator de ramificação efetivo:

$$\frac{B^{d+1}-1}{B-1} = N$$

B – nº de filhos por nó

d – comprimento do caminho na árvore correspondente à solução

$d = 3$

B	N
2	15 >> 11
1.5	8 < 11

$$1.5^4 \approx (1.5^2)^2 = 2.25^2 \simeq 2 \times 2.25 \simeq 5$$

$B \approx 1.75$

Exame LEI 09/01/2015

1. funções em Python – o que fazem

a)

```
def f(x, y, z):
    if y==[]:
        return [x]
    if z(x, y[0]):
        return [x]+y
    return [y[0]] + f(x, y[1:], z)
```

→ dados um elemento (x), uma lista (y) e uma função (z): se y for uma lista vazia, retorna o elemento x em formato de lista; se a função z retornar true (tendo por argumentos x e o primeiro elemento de y), retorna a concatenação do elemento x como lista c/ a lista y; em qualquer outro caso, aplica recursividade

b)

```
def h(x, y, z):
    if x==[]:
        return y[:]
    if y==[]:
        return x[:]
    if z(x[0]) < z(y[0]):
        return [x[0]] + h(x[1:], y, z)
    return [y[0]] + h(x, y[1:], z)
```

dadas 2 listas, compara-as no mesmo índice e guarda o mínimo p/ cada iteração

→ dadas duas listas (x, y) e uma função (z): se x for uma lista vazia, retorna a lista y, e vice versa; se o resultado da aplicação da função z sobre o primeiro elemento da lista x for menor que essa mesma aplicação sobre o primeiro elemento da lista y, é aplicada a recursividade sobre a lista x; em qualquer outro caso, a recursividade é aplicada sobre a lista y

b) $N = 9$, $x = 3$

$$RM = \frac{N-1}{x} = \frac{9-1}{3} = \frac{8}{3}$$

c) Pesquisa em largura → avaliar todos os nós de um determinado nível antes de prosseguir p/ a avaliação dos nós do próximo nível.
Pesquisa de custo uniforme → caso particular da pesquisa A*, tendo um comportamento semelhante à pesquisa em largura. Caso exista solução, a primeira solução encontrada é ótima.

4) propagação de restrições

Exame modelo

1. a)

```
def f(x):
    if x == []:
        return 0
    if x[0] > 0:
        return x[0] + f(x[1:])
    return f(x[1:])
```

A função recebe uma lista e ~~efetua a~~ soma de todos os nºs positivos

→ dada uma lista x: retorna 0 se x for uma lista vazia; guarda o 1º elemento de x e aplica a recursividade caso o 1º elemento de x tenha um valor superior a zero; aplica a recursividade em qualquer outro caso, desprezando o 1º elemento.

b)

```
def g(x):
    if x == []:
        return [[]]
    y = g(x[1:])
    return y + [[x[0]] + z for z in y]
```

A função recebe uma lista e devolve uma lista de listas. Essas listas correspondem às combinações possíveis dos elementos da lista passada como argumento.

→ dada uma lista x: retorna uma lista de listas (tudo vazio) se x for uma lista vazia; caso contrário, ~~aplica a x (exceto o 1º elemento) a função~~ aplica a recursividade à lista x, desprezando o seu 1º elemento, e guardando o resultado em y. No final devolve y e o primeiro elemento de x (como lista) c/ concatenação dos elementos em y

II

1. a) $\forall x \; Livro(x) \vee (BandaDesenhada(x) \Rightarrow \neg Capa(x, Dura))$

$\equiv \exists x \; \neg Livro(x) \wedge \neg(BandaDesenhada(x)) \vee Capa(x, Dura)$

$A \Rightarrow B \equiv \neg A \vee B$

Nenhuma das anteriores

b) ~~[illegible]~~

c) Pesquisa por melhorias sucessivas é uma técnica p/ ~~resolução~~ de problemas de atribuição

d) ~~(A ∨ B) ∧ (¬B ∨ C ∨ D) ∧ ¬A ∧ ¬D~~

A ∨ B
¬B ∨ C ∨ D
A ∨ C ∨ D
¬A
C ∨ D
¬D
C

~~[illegible]~~

~~Melhorias das [illegible]~~ ⇒ C

e) operadores STRIPS ~~são~~ mecanismos p/ geração de planos no mundo dos blocos

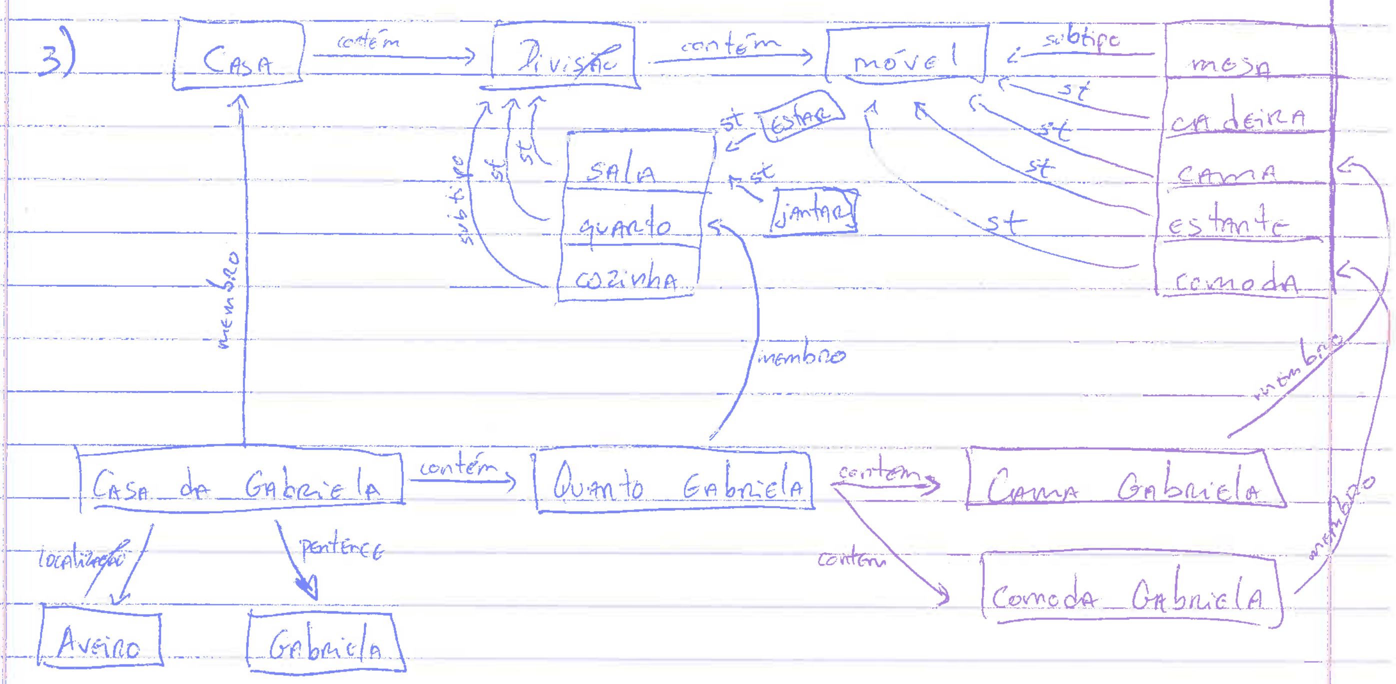
2) Pesquisa em árvore em profundidade: tem como ponto de partida um nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder (backtracking).

Montanhismo, diferença: é escolhido sempre o sucessor c/ melhor valor da função de avaliação, não há retrocesso (backtracking) e, quando o valor da função no nó atual é superior ao valor da função em qualquer um dos seus sucessores, a pesquisa pára, ou seja, atingiu-se o máximo local.

I) 2)

```
def interpretacoes(x):
    if x == []:
        return [[]]
    list = []
    rec = list(x[1:])
    for a in rec:
        list += [[(x[0], True)] + a]
        list += [[(x[0], False)] + a]
    return list
print(interpretacoes(["a", "b"]))
```

3)



4.a) D

	A	B	C	D	E	F	G	H	I	J	K	L
heurística ~~custo~~	4	4	7		4	9	10	11	9	12	10	10
custo real ~~n assoc.~~	9	6	7		5	11	12	12	8	13	9	13

P/ que a heurística seja admissível, o valor do nó não pode ser superior ao ~~valor~~ do custo até à solução (D). Neste caso, existem duas situações que contrariam essa regra — logo, a heurística não é admissível.

⇒ mudar os custos dos nós I e K

b) A*

G/1/10

F/2/10 J/2/14 K/3/14 L/4/11

B/5/10 C/6/13 G/7/11

C/8/14 D/9/12 E/10/15 F/11/20

~~5.a) Rua(N1, N2): rua c/ início em N1 e fim em N2~~
~~R(R, N): estar no parque na rua R e no ponto N~~
~~Loc(R, N): estar na rua R no ponto N~~
~~LocP(R, N): existe um parque na rua R e ponto N~~

~~b) Atravessar(~~

Rua (N_1, N_2): rua c/ início em N_1 e fim em N_2
Loc(N_1, N_2): estar em N_2 do lado N_1
Parque (N, P): estar no nó N, no parque P
ExistParque (N, P): existe um parque (P) no nó N

b) Atravessar (N_1, N_2, N_3) STRIPS
P.C = {Loc(N_2, N_3), Rua(N_1, N_2)}
EN = {Loc(N_2, N_1)}
EP = {Loc(N_2, N_3)}

Estacionar (N_1, N_2, P)
PC = {Loc(N_2, N_1), ExistParque(N_2, P)}
EN = {Loc(N_2, N_1)}
EP = {Parque(N_2, P)}

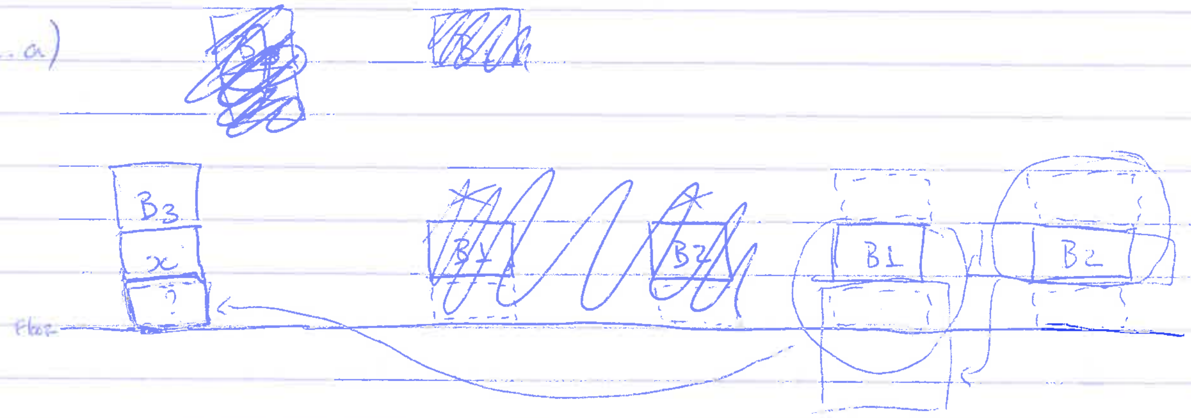
Percorrer (R, N_1, N_2)
PC = {Loc(R, N_1), Rua(R, N_2)}
EN = {Loc(R, N_1)}
EP = {Loc(R, N_2)}

Sair (N, P, R)
PC = {ExistParque(N, P), Parque(N, P), Loc(R, N)}
EN = {Parque(N, P)}
EP = {Loc(N, R)}

Questões modelo

III

2.a)



min = 4

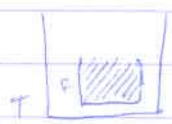
On(B_3, x)
¬Clear(B_1)
¬Clear(B_2)

(b)

min = 2

3) torneira

R em Tx se Tx ⊄ água



a.i) Open(T_2) ∧ Over(R, T_2) ⟶ {Water(T_1) = 0, Water(T_2) = 0, Water(R) = 1}

ii) Open(T_2) ∧ Over(R, T_1) ⟶ {Water(T_1) = 0, Water(T_2) = 1, Water(R) = 0 ∨ Water(R) = 1}

iii) $\neg Open(T_2) \wedge Over(R, T_1)$ $\begin{cases} Water(T_1) = 0 \\ Water(T_2) = 0 \vee Water(T_2) = 1 \\ Water(R) = 0 \vee Water(R) = 1 \end{cases}$

iv) $\neg Open(T_1) \wedge \neg Open(T_2) \wedge Over(R, T_1) = \begin{cases} Water(T_1) = 0 \\ Water(T_2) = 0 \vee Water(T_2) = 1 \\ Water(R) = 0 \end{cases}$

b.i) $\forall x \, (\neg(\neg Open(x) \Rightarrow Water(x)))$

$\xrightarrow{A \Rightarrow B \,\equiv\, \neg A \wedge B}$ $Open(x) \wedge Water(x)$

$\equiv \neg Open(x) \vee \neg Water(x)$ satisfazível, não tautologia

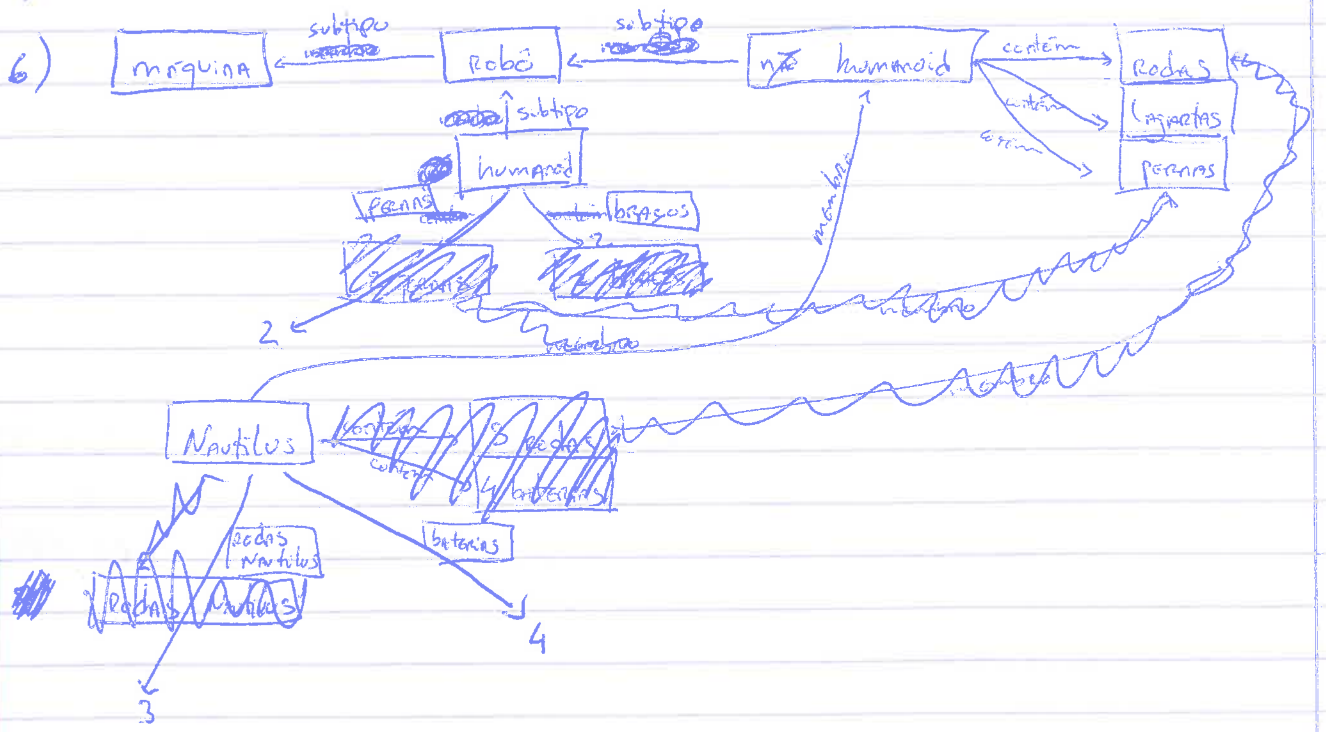
iii) $\forall x \; Open(x) \Rightarrow \exists y \; Water(y)$

$\forall x \exists y \; \neg Open(x) \wedge Water(y)$ satisfazível

5) KIF – linguagem desenhada p/ representar o conhecimento entre agentes. Pode ser usada também p/ representar os modelos internos de cada agente.

O mundo é conceptualizado em termos de objetos e relações entre objetos

6)



8)

$p(a \wedge b \wedge \neg c \wedge \neg d) = p(a) \times p(b|a) \times (1 - p(c|b)) \times (1 - p(d|b))$

$= 0.2 \times 0.3 \times (1 - 0.2) \times (1 - 0.1)$

$= 0.2 \times 0.3 \times 0.8 \times 0.9$

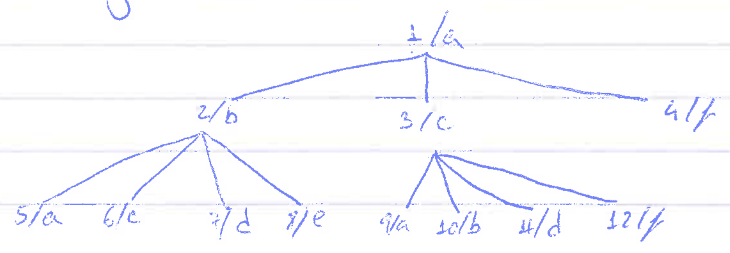
$= 0.0432$

//

Pesquisa em árvore
- cega (não informada)
  - em largura
  - " profundidade
    - □ c/ limite
    - □ c/ " crescente
  - custo uniforme
- informada
  - gulosa (greedy) — uma variante
  - A* e variantes

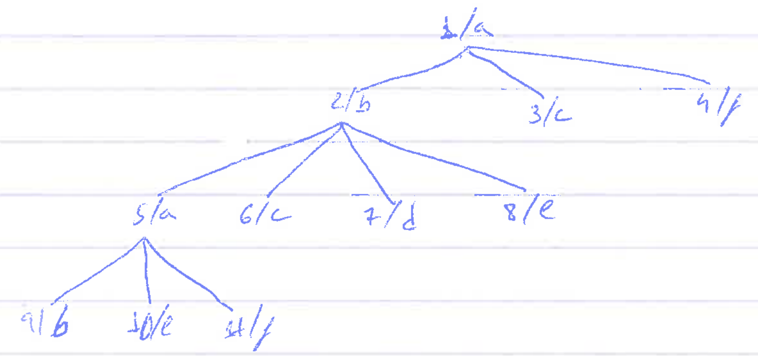
Pesquisa por melhorias sucessivas (p/ resolver problemas de atribuição)
- montanhismo
- recozimento simulado
- Algoritmos genéticos
- reparação heurística (começar c/ solução random e fazer reparações de acordo c/ uma heurística local)

em largura ⟵ FIFO



[1]
[2, 3, 4]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8, 9, 10, 11, 12]

em profundidade ⟵ LIFO



[2, 3, 4]
[5, 6, 7, 8, 3, 4]
[9, 10, 11, 6, 7, 8, 3, 4]

$(h(n)=0,\ f(n)=g(n))$

custo uniforme — caso particular da pesquisa A*. Comportamento semelhante à pesquisa em largura. Caso exista, a primeira solução é ótima.

A* ⟶ $\underbrace{f(n)}_{\text{custo total}} = \underbrace{g(n)}_{\text{custo real ou acumulado}} + \underbrace{h(n)}_{\text{heurística}}$

A heurística é admissível se nunca ultrapassa o custo real

vantagens ⟨ se a heurística é admissível, a solução é sempre ótima
também é completa

(#)

gulosa ⟶ $f(n) = h(n)$, ignora $g(n)$

dado que ignora o custo acumulado, não é variante de A*
↳ c/ isto, facilmente deixa escapar a solução ótima
comportamento semelhante à pesquisa em profundidade

Ramificação média ⟶ $RM = \frac{N-1}{X}$ N - nr de nós para solução encontrada
X - nr nós expandidos

Fator de ramificação efetivos ⟶ $\frac{B^{d+1}-1}{B-1} = N$

B - nr de filhos por nó , d - comprimento do caminho correspondente à solução

STRIPS — planeamento no mundo dos blocos
→ A funcionalidade de um certo tipo de operação é definida através de uma estrutura chamada operador
- Pré-condições: condições de aplicabilidade
- Efeitos negativos: propriedades que deixam de ser verdade ao executar
- Efeitos positivos: " " passam a ser verdade ao executar

(#) desvantagens (A*) → consumo de memória e tempo c/ comportamento exponencial
→ em problemas complexos, pode ser necessário algoritmos mais eficientes (mesmo que não ótimos)

variantes de A* → IDA* (A* c/ aprofundamento iterativo)
→ SMA* (A* c/ memória limitada simplificado)
→ custo uniforme

RBFS → funciona como pesquisa em profundidade c/ retrocesso

Montanhismo — vales (soluções menos satisfatórias) e colinas (+ satisfatórias)
similar à pesquisa em profundidade; diferenças:
- escolhe-se sempre o sucessor c/ melhor valor da função de avaliação
- não há backtracking
- quando o valor da função do nó atual é superior ao valor da função em qualquer um dos seus sucessores, para (atingiu-se máximo local)

problemas: máximos locais, planaltos e ravinas. Como resolver?

correr o algoritmo várias vezes e escolher melhor solução

Recozimento simulado — variante do montanhismo na qual podem ser aceites refinamentos que, localmente, pioram a solução
particularidades:
- sucessor selecionado aleatoriamente
- quando o valor da função no nó atual é superior ao valor da função no sucessor, o sucessor é aceite c/ uma probabilidade que diminui exponencialmente em função da perda na função de avaliação
- termina quando o indicador "temperatura" chega a zero

Representação do conhecimento
- redes semânticas
- lógica proposicional e de primeira ordem
- linguagem KIF (trocar conhecimento entre agentes)
- engenharia do conhecimento (representações e regras)
- ontologia geral
- redes de Bayes (representa conhecimento impreciso; probs)