

Grupo 1

Andrés Gómez de Silva ❤

**Definiciones****Artificial:** creado por el hombre, no existe en la naturaleza.**Inteligencia:** decisiones óptimas, reconoce patrones, aprendizaje/generalizar, meta-razonamiento (razonar acerca del razonamiento) → crear algoritmos, inferencia, comunicación vía lenguaje natural, comportamiento orientado a metas, exploración, capacidad de buscar alternativas, adaptabilidad.

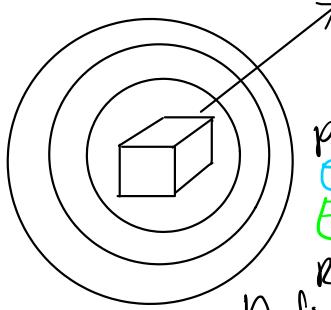
No necesariamente se deben cumplir todas pero tampoco tan pocas para decir que alguien es inteligente.

**Varios niveles de inteligencia:**

- Humanos
- Monos
- Elefantes / del fin
- Perros
- Ratones
- Cucarachas
- Ámibas.

**Categoría Radial:** Hay un centro, diferentes direcciones. No hay un significado específico.Lenguaje de programación: **prolog****Inteligencia:** conocimiento + búsqueda.**Evaluación:**

- 3 exámenes: cada uno 24%, 2 parciales y 1 final
- Proyectos: 28%
- Promedio de exámenes 6.0 mínimo y proyectos 6.0.



prototipo → se va alejando de la definición.

Ejemplos prototípicos: ayudan a definir cosas

Ejemplo: carretera: camino plano para que vehículos con ruedas se trasladan de un lugar a otro.

Definiciones completas serían muy largas.

**Teoría de múltiples tipos de Inteligencia:** dependen de la versión de la teoría ⇒ 9 tipos inteligencia.

- 1- visual / espacial
- 2- existencialista
- 3- interpersonal
- 4- intrapersonal
- 5- corporal / kinestética

- 6- musical / rítmica
- 7- verbal / lingüística
- 8- lógica / matemática
- 9- naturalista

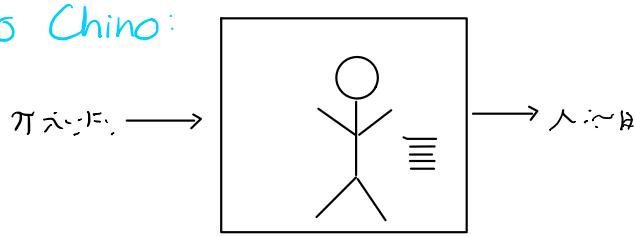
No porque a una persona le falte alguna significa que no sea inteligente.  
Comunidad ITAM.

### Hipótesis de la I.A.

- ① Se puede reproducir el comportamiento (las características) de la inteligencia artificialmente.
- ② Se pueden estudiar los fenómenos relacionados con la inteligencia usando computadoras.

John Searle no está de acuerdo con esto.

### Cuarto Chino:



La persona dentro del cuarto no necesariamente está entendiendo chino o racionalizarlo, simplemente reproduce respuestas sin entenderlas.

**Técnica IA:** Inteligencia requiere conocimiento.

Inteligencia: conocimiento + búsqueda.

Conforme crecen el tamaño de memoria es mejor ya que se tienen más datos.

- Organizar datos se deben organizar de manera flexible.

**Sistema de IA:** debe basarse en una teoría para resolver lo que se quiere almacenar. Descripción prototípica, lo más general posible.

Mientras más general es el conocimiento, mejor.

Programador debe hacer un análisis del conocimiento. Algoritmo sabe lo que busca y en cuantos bytes se almacenan. Debe ser factible de usar y fácil de modificar. Meta-conocimiento → conocer el conocimiento.

Solución de IA: Inteligencia del programador y lograr lo que se busca.

Ejemplo: juego del gato: complejidad, generalización, claridad del conocimiento, qué tan fácil es extender a otras aplicaciones.

- Board (vector de 9 elementos)

1	2	3
4	5	6
7	8	9

0 ⇒ nadie ha jugado

1 ⇒ Jugador X

2 ⇒ Jugador O

combinaciones:  $3^9 \Rightarrow 19,683 \Rightarrow$  desperdicio de memoria.

Segunda versión:

Board: sigue siendo igual, agregan 2 (blank), 3 (X), y 5 (O)

Turn: indica cuál movimiento se va a jugar

Algoritmo: 5 si es blank, si no regresa los blanks no esquinas analizan primero el cuadrado central y después las no esquinas.

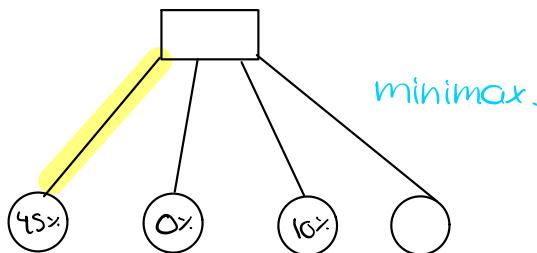
El 2, 3 y 5 se usó para un análisis más rápido.

Método Go: se pone por turnos, no es tan eficiente.

Con cuadrado mágico es más fácil filtrar la búsqueda así toda solución debe sumar 15.

8	3	4
1	5	9
6	7	2

Solución #3: se puede proponer probabilidad de ganar o perder el juego para ahorrar pasos.



Las estadísticas ayudan a simular cosas para conocer a contrincantes.

Es una estrategia general para juegos y algunos deportes, se basa en teorías. Se trata de analizar lo más profundo posible.

Prolog 1972.

Todo un periodo de transición en la implantación de un lenguaje.  
 Francia → Universidad Marsella.  
 Investigador principal: Alain Colmerauer.

PROgramation LOGic. Lenguaje muy diferente a los otros.

Se tienen condicionales y ciclos también.

SWI-Prolog → descargar.

## Programación lógica:

Premisa 1: Todos los perros son mamíferos

$\forall x \mid \text{perro}(x) \rightarrow \text{mamifero}(x)$

Premisa 2: Todos los galgos son perros

$\forall x \mid \text{galgo}(x) \rightarrow \text{perro}(x)$

Premisa 3: Fidulais es un galgo

galgo(Fidulais)

Conclusión: Fidulais es un mamífero.

mamifero(Fidulais)

Instanciación: se instancian las variables (es asignación en java).  
 No se puede reasignar el valor a una instancia; e.i.: i=1 ... i=3  $\otimes$

En prolog no se puede acceder a "clases"

Se pueden tener variables en métodos, siempre son locales las variables.

En prolog la implicación está al revés:

`mamifero(X) :- perro(X).`  
`perro(X) :- galgo(X).`  
`galgo(Fidulais).`

`:- "if"`

sintaxis: cabeza de la regla :-  
 cuerpo de la regla.

Cabeza de la regla: conclusión

Cuerpo de la regla: condiciones.

- \* Se pueden tener muchas condiciones, pero sólo una conclusión.
- \* Cualquier símbolo que empieza con mayúscula, prolog lo interpreta como una variable, símbolo que empieza con minúscula son constantes.
- \* Cada regla o hecho debe terminar con un punto. (cláusula)
- \* **Predicado:** perro, mamífero, galgo. Hay predicados predefinidos; pero también se pueden declarar nuevos.

Programar en Prolog ⇒

- ① Definir hechos
- ② Definir reglas.

Acerca de: el dominio del programa.  
 (tipo de problemas para el que está diseñado)

- ① Definir hechos y **base de conocimientos**.
- ② Definir reglas
- ③ Definir consulta(s)

↳ lo que avanza el programa. (main en java).

Sintaxis consulta: ? - consulta.

SWI-Prolog pone la extensión \*.pl

**Programa 1:** /\* Este es un comentario  
\* en SWI-Prolog \*/

% Comentario en una sola linea.

hombre (jose).

mujer (maria). } se piden poner 2 cláusulas separadas por otra.  
hombre (juan).

% El primer argumento es el papá del segundo argumento:

papa (juan, jose).

papa (juan, maria).

valioso (dinero).

dificil De Obtener (dinero).

Le - da (pedro, libro, cantonio).

hermana (X, Y) :-  
 papa (Z, X),  
 mujer (X),  
 papa (Z, Y),  
 X \= Y. % diferente de Y.

AND, OR ; NOT not

} "sub consultas"

humano (X) :- humano (X) :-

mujer (X); = mujer (X). } esta es

hombre (X). humano (X) :- hombre (X). } mejor

File → ventana chiquita → escoges programa a consultar.

Quien → write: te imprime la variable que encontró que haga match con la búsqueda.

Muestra solo una solución a la vez. Apretando space, te muestra otra solución y manda a nueva consulta.

write: predicado predefinido

nl: new line (saltar linea)

fail: obliga a generar más respuestas, no solo una. Además la respuesta la pone como false.

\* Cuando hay más de una cláusula, se llama **predicado no determinístico** y el programa no te arroja una solución si no recorre a todas las reglas.

?- papa (Papa, Hijo o Hija). write (Papa). write (""). write (Hijo o Hija).  
nl. fail

Juan Jose  
Juan maria  
false.

23 / enero / 19

## PROGRAMA 1

**hombre** (José)

**Hombre** (Juan)

**Mujer** (maria)

**Papa** (Juan, Jose)

**Papa** ( Juan, Maria)

**Valioso** (dinero)

**DificilDeObtener** ( dinero)

**Le\_da** (Pedro, libro, Antonio)

**Hermana** (X, Y):-

    papa (Z, X),  
    mujer (X),  
    papa (Z, Y),  
    X\==Y.

**Humano** (X):-

    mujer(X).

**Humano** (X):-

    hombre

## CONSULTAS:

?- hombre (maria), mujer (maria)

**false**

?- papa (X,\_\_), write (X), nl, fail.

Juan

Juan

**False**

?- hermana (maria,\_\_)

**true;**

**false.**

?- hermana (\_\_, José). %pregunta si José tiene hermanas

**true**

?- hombre (\_\_). %imprime el numero de hombres que hay

**true;**

**true.**

?- papa (José,\_\_). %pregunta si José es papa de alguien

**false.**

## PROGRAMA 2

Grande (bisonte).  
Grande (oso).  
Grande (elefante).  
Chico (gato).  
Cafe (oso).  
Cafe (bisonte).  
Negro (gato).  
Gris (elefante).

Oscuro (Z):-

  cafe (Z).

Oscuro (Z):-

  negro (Z).

## CONSULTAS

?— oscuro (X), grande (X), write (X), nl, fail.

  oso

  bisonte

**false.**

?—

%cambio X por Z y las unifica. Debe regresar a buscar  
Más opciones porque es no determinado. Almacena  
Un backtracking point

## Aritmética de Prolog:

Igual que	= : =	menor o igual que = <
Diferente de	= \=	mayor o igual que = >
menor que	<	
mayor que	>	

Suma	+	residuo división	rem
Resta	-	cociente división	//
multiplicación	*	Exponentiación	**
división	/		

## PROGRAMA 3

%valor\_max (i, i, o): es decir, dos entradas y una salida

INPUT: instantiated

OUTPUT: open

Análisis de flujo: lo que hace prolog para determinar si los argumentos son de entrada o de salida.

**Valor\_max** (X, Y, Z):-      %a fuerzas se tiene que poner del lado izquierdo una variable no asignada  
  X>Y, Z is X.

**Valor\_max** (X, Y, Z):-  
  X=<Y, Z is Y.

## CONSULTAS

?— valor\_max(3, 5, X), write (X).

5

X=5.

?— valor\_max (5, 3, X), write (X). %sale false porque no hay una tercera cláusula. Las condiciones cubren todas las posibilidades.  
5  
X=5;  
**false.**

## PROGRAMA EDITADO

**Valor\_max** (X,Y, X):-

X>Y.

**valor\_max** (X, Y, Y):-

X=<Y.

## CONSULTAS

?— valor\_max(3, 5, X), write (X).

5

X=5.

?— valor\_max (X, 3, 5), write (X).

5

X=5;

**false.**

?— valor\_max (5, X, 3), write (X).

**false.**

?— valor\_max(X, 5, 3), write (X).

**false.**

?— valor\_max (5, X, 5), write (X).

**ERROR**

%no truena porque al poner una variable en la búsqueda el Programa busca un valor que asignarle

Predicado predefinido **Importante:** ! no lleva argumento, se escribe para poner en corte. Su valor lógico siempre es true.

## PROGRAMA EDITADO

**Valor\_max** (X, Y, X):-

X>Y, !.

**Valor\_max** (X, Y, Y) :-

X=<Y.

## CONSULTAS

?— valor\_max(3, 5, X), write (X) %no siguió buscando otros resultados por el break

5

X=5.

?— valor\_max (5, 3, X), write (X)

5

X=5.

## PROGRAMA EDITADO

**Valor\_max** (X, Y, X):-

X>Y, !.

**Valor\_max** (\_ , Y, Y).

% ya no es necesario poner la segunda condición porque Y es el mayor  
Y solo te imprime Y

**Fail:** predicado. hace que regrese al último punto de retroceso.

## PROGRAMA 4

### Saludos:—

```
prolog_pais (Pais),  
  write ("Hola"),  
  write (Pais), nl,  
  fail.
```

### Saludos.

**Prolog\_pais** (Japón).  
**Prolog\_pais** (Francia).  
**Prolog\_pais** (Hungria).  
**Prolog\_pais** (bhutan).  
**Prolog\_pais** (Colombia).  
**Prolog\_pais** (espana).

## CONSULTAS

?— saludos

```
Hola Japón  
Hola Francia  
Hola Hungria  
Hola Bhutan  
Hola Colombia  
Hola espana  
true.
```

%si se comenta saludos del programa (el segundo) date regresa fail

?— saludos

```
Hola Japón  
Hola Francia  
Hola Hungria  
Hola Bhutan  
Hola Colombia  
Hola espana  
fail.
```

?— X=3, X = : = 3, Y=4, Z=Y//X, write (Z), nl. %si la consulta solo contiene operadores y predicados Predefinidos, realiza la consulta bien

```
4//3  
X = 3.  
Y = 4.  
Z = 4//3.
```

?— X=3, X = : = 3, Y=4, Z is Y//X, write (Z), nl.

```
1  
X = 3.  
Y = 4.  
Z = 1.
```

## PROGRAMA EDITADO

### Saludos:—

```
prolog_pais(Pais1),  
  write (Pais1),  
  write ( " saluda a: "), n1, !,  
  prolog_pais (pais2),  
  Pais2\==Pais1, al ser iguales regresa al ultimo punto de retroceso  
  write(Pais2), nl, fail.
```

por este coete solo se repite lo de abajo en el ciclo

## CONSULTAS

?—saludos.

Japón saluda a  
 Francia  
 Hungría  
 Bhutan  
 Colombia  
 España  
**false.**

28 / enero / 19

## PROGRAMA 6.pl

**Gusta** (Jorge, X) :-  
 carne (X).  
**Gusta** (Beatriz, X) :-  
 X == hígado, !, fail.  
**Gusta** (Beatriz, X) :-  
 carne (X).  
**Carne** (pancita)  
**Carne** (hígado)  
**Carne** (filete)

→ siempre mostranc  
 que a Beatriz no  
 le gusta el  
 hígado.

	Igual A	Diferente A
variable numérica	= : =	= \ =
valores en general	= =	\ = =

## CONSULTAS

?— gusta (Beatriz, filete), write ("Le gusta a Beatriz el filete "), nl, gusta (Jorge, hígado), write ("Le gusta a Jorge el hígado "), nl, gusta (Beatriz, hígado), write ("Le gusta a Beatriz el hígado "), nl.

Le gusta a Beatriz el filete  
 Le gusta a Jorge el hígado  
**false.**

## PROGRAMA 6a

**Gusta** (Jorge, X) :-  
 carne (X).  
**Gusta** (Beatriz, X) :-  
 not (X == hígado), carne(X).  
**Carne** (pancita).  
**Carne** (hígado).  
**Carne** (filete).

## CONSULTAS

?— gusta (Beatriz, filete), write ("Le gusta a Beatriz el filete "), nl, gusta (Jorge, hígado), write ("Le gusta a Jorge el hígado "), nl, gusta (Beatriz, hígado), write ("Le gusta a Beatriz el hígado "), nl.

Le gusta a Beatriz el filete  
 Le gusta a Jorge el hígado  
**false.**

**Hechos:** predicados de orden 0 C no involucran variables, solo constantes

**Reglas:** predicados de primer orden.

¿Hay predicados de orden superior?

Predicados que se usan como argumentos de otros predicados.  
("FUNCIONES")

## PROGRAMA 7

Evento (siglo15, "Portugueses y españoles exploran África, América y Asia").

Evento (siglo 16, "Leonardo Da Vinci pinta la Mona Lisa").

Evento (siglo17, "Construcción del Taj Mahal").

Evento (siglo18, "Benjamin Franklin inventa lentes bifocales, estudia la electricidad").

Evento (siglo19, "Independencia de México").

Evento (siglo20, "Invención del internet").

Evento (siglo21, "caída de las torres gemelas en N.Y.").

Antes\_de (evento(siglo15, \_\_), evento(siglo16, \_\_)).

Antes\_de (evento(siglo16, \_\_), evento(siglo17, \_\_)).

Antes\_de (evento(siglo17, \_\_), evento(siglo18, \_\_)).

Antes\_de (evento(siglo18, \_\_), evento(siglo19, \_\_)).

Antes\_de (evento(siglo19, \_\_), evento(siglo20, \_\_)).

Antes\_de (evento(siglo20, \_\_), evento(siglo21, \_\_)).

Antes\_de (X, Z),

Antes\_de (Z, Y). lo necesario lo delinea

## CONSULTAS

?— antes\_de(evento (siglo15, X), evento (siglo16, Y)), write(X), nl, write (Y), nl.

1886

1892

true.

?— antes\_de(evento (siglo15, X), evento (siglo20, Y)), write(X), nl, write (Y), nl. no agregamos regla  
false. de sucesión 15,16,17 ..

?— antes\_de(evento (siglo15, X), evento (siglo20, Y)), write(X), nl, write (Y), nl

true.

Métodos en prolog que nos permite agrupar datos.

## Listas en Prolog:

Lista es [Cabeza | Cola]  
el 1<sup>er</sup> elemento → resto de los elementos  
esta a su vez es una lista  
(recursividad)

Lista [a,b,c] [3] [[1,2],[3,4], [5,6,"alfa"]] []	Cabeza a 3 [1,2] no hay	Cola [b,c] [] [[3,4],[5,6,"alfa"]] no hay.
---	-------------------------------------	--

## Programa8

**Pertenece\_a** (X, [X|\_\_]) :-  
!

**Pertenece\_a** (X, [\_\_|Z]) :-  
pertenece\_a (X, Z).

## CONSULTAS

?—pertenece\_a(a,[a, b, c]).  
**true.**

?—pertenece\_a (a, [b, c, a]).  
**true.**

## Programa8a

%pertenece\_a(i,i)

**Pertenece\_a** (X, [X|\_\_]) :-  
    **write** ("algo"), **nl**,  
!

**Pertenece\_a** (X, [\_\_|Z]) :-  
    **write** ("algo"), **nl**,  
pertenece\_a (X, Z).

## CONSULTAS

?—pertenece\_a (z, [b, c, a]).  
    algo  
    algo  
    algo  
**false.**

?—pertenece\_a (z, [ ]).  
**false.**

## PROGRAMA 9

%combina (i, i, o).. combina (i, i, i), combina  
(o, i, i)

**Combina** ([ ], Lista, Lista) :-  
!

**Combina** ([X|Lista1], Lista2, [X | Lista3]) :-  
combina (Lista1, Lista2, Lista3).

## CONSULTAS

?— combina ([1, 2, 3], [a, b], X), **write**(X), **nl**  
[1, 2, 3, a, b]  
X= [1, 2, 3, a, b]

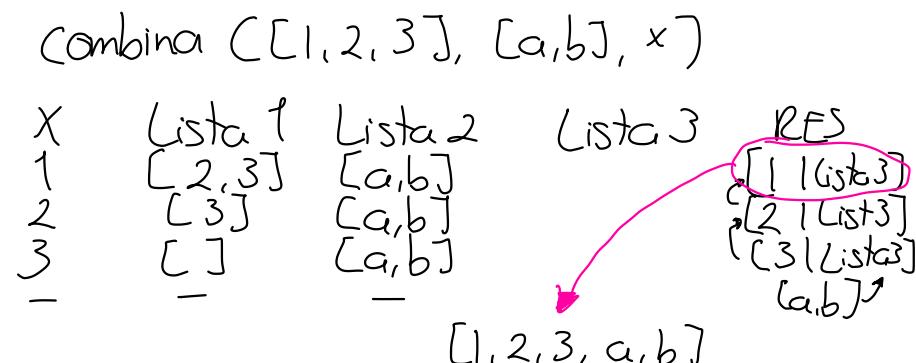
?—combina ([1, 2, 3],[[a, b], X]), **write**(X), **nl**.  
[1, 2, 3, [a, b]]  
X= [1, 2, 3, [a, b]]

?—combina ([ ], [a, b, c], X), **write**(X), **nl**.  
[a, b, c]  
X = [a, b, c]

?—combina ([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3]).  
**true.**

?—combina ([ ], [b, c, a], [1, 2, 3]).  
**false.**

?—combina ([ ], [ ], [ ])  
**true.**



?— combina ([a, b, c], [c ,b a], [a, b, c, a, b, c])

**false.**

?— combina (X, [1, 2, 3], [a ,b , 1, 2, 3]).

X=[a, b].

?— combina ([1, a], X, [1, a, 3, 5, j])

X=[3, 5, j].

?— combina (X, [1, 2, 7, a], Y)

X=[ ].

Y=[1, 2, 7, a].

?— combina (X, Y, Z)

X=[ ]

Y=Z

?— combina ([3, 2, 5], X, Y)

Y=[3, 2, 5 | X].

## PROGRAMA 10

**Repite.** → repite se puede llamar de cualquier otra manera.

**Repite:**—

repita.

**Lee\_valores** :—

repita,

**read(X),**

**write(X),**

**nl,**

**X==ya.**

**lee\_valores**

## CONSULTAS

?— lee\_valores. *se tiene que escribir punto al final para que funcione*  
4. *el write.*

Japón.

[y, a, y].

ya.

true.

30/enero/19

## Funciones que cambian hechos y predicados

2 predicados predeterminados:

**assert** (hecho o regla) = afirma algo en la ejecución del programa

J Unit Test de Java.

**retract** (hecho o Regla)

En SWI-Prolog solo se pueden usar declarando los hechos o Reglas hasta arriba del programa, seguido de :- dynamic y una palabra.

:- dynamic

:- dynamic pais/1.

**Pais(Holanda)**

**Pais(Francia)**

**Repite.**

**Repite:**—

repita.

**Escribe\_paises:**—

pais(X),

X\==ya,

write(X),

nl,

fail.

**Escribe\_paises.**

**Main:**—

write("Dame los nombres de varios ">,

write("paises y escribe ya cuando ">,

write("quieras terminar: "), nl,

repita,

read(Pais),

assert(pais(Pais)),

Pais==ya,

escribe\_paises.

## CONSULTAS

?—main.

Japón.

Colombia.

Italia.

Hungría.

Botswana.

ya.

Holanda

Francia

Japón

Colombia

Italia

Hungría

Botswana

**True.**

?—main.

Irlanda.

ya.

Holanda

Francia

Japón

Colombia

Italia

Hungría

Botswana

Irlanda

**True.**

Se escriben con punto al final para que se guarde.

El programa guarda lo anterior y solo le inserta lo nuevo.

**Azidat:** número de argumentos que necesitan los predicados

Se tiene que cambiar el nombre de la variable si se intenta usar como no dinámico para que funcione bien.

## PROGRAMA12

:- dynamic antes\_de/2.

**Antes\_de**(ayer,hoy).

**Antes\_de**(hoy,mañana).

se pone paréntesis porque si se omite se leería la coma como otro argumento.

**Main:**-

```
not(antes_de(ayer, mañana)), write("uno"), nl, assert((antes_de(X,Y):- antes_de(X,Z), antes_de(Z,Y))),  
write("dos"), nl,  
antes_de(ayer,mañana), write("tres"), nl, !.
```

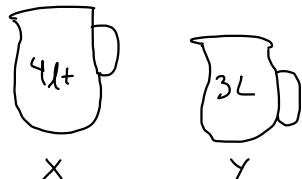
**Proyecto 1:** se dandó un programa en Netbeans y se tiene que duplicar exactamente en probg, con las mismas operaciones, los mismos resultados.

gtrace: debuga el código.

06 / febrero / 19

Búsqueda.

Inteligencia = conocimiento + búsqueda



Queremos 2 lts en el jarrón de 4 litros.  
① Identificar qué variables son relevantes:  
x, y representan cuántos litros de agua tiene cada jarrón.

Nos permite saber la forma en la que se describe y varía el estado del problema.

② Identificar el estado inicial: ambos jarrones están vacíos

③ Identificar el estado final: ( $x=2, y=y, 1 \leq y \leq 3$ )

A veces se da sólo un estado final o inicial y a veces se da algo genérico. Distintos niveles de precisión.

④ Cómo cambiar de un estado a otro: transiciones válidas entre estados. Acciones u operaciones que tenemos disponibles.

**Solución** ① Llenar jarrón 4 lt.

- ② Llenar 3 litros
- ③ Vaciar 4 litros.
- ④ Vaciar 3 litros
- ⑤ Pasar contenido de 4 a 3. (que quepa)
- ⑥ Pasar contenido de 3 a 4. (que quepa).

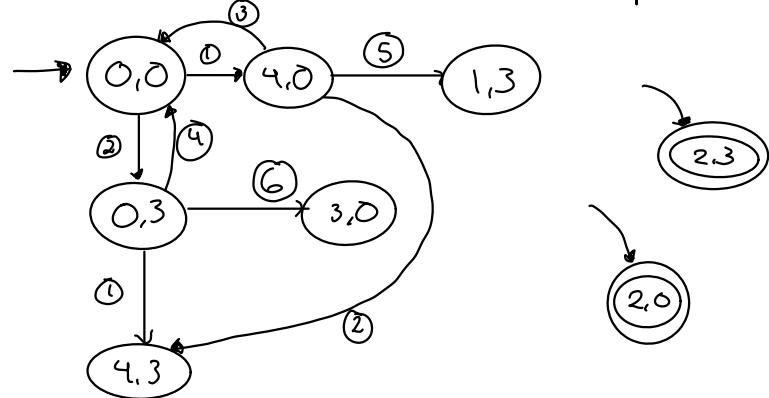
	edo. inicial	edo. final
$x_i$	$0 \leq x_i \leq 4$	$x'_i$
$y_i$	$0 \leq y_i \leq 3$	$y'_i$
		$4$
		$0 \leq y'_i \leq 3$
		$0 \leq x'_i \leq 4$
		$0 \leq y \leq 3$
		$0$
		$0 \leq x \leq 4$
		$0$
		$g_i + \min(x_i, 3y_i)$
		$0 \leq y_i \leq y_i - \min(x_i, 3y_i)$
		$0 \leq y'_i \leq y_i - \min(x_i, 4x_i)$

Solución particular:  $4l: 0 \xrightarrow{①} 4 \xrightarrow{⑤} 1 \xrightarrow{④} 1 \xrightarrow{⑤} 0 \xrightarrow{①} 4 \xrightarrow{⑤} 2$   
 $3l: 0 \xrightarrow{①} 0 \xrightarrow{③} 3 \xrightarrow{②} 3 \xrightarrow{⑥} 4 \xrightarrow{③} 0 \xrightarrow{⑥} 2$

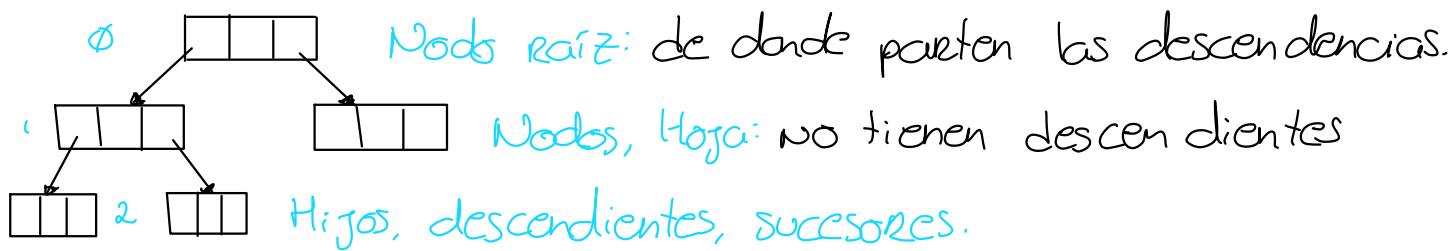
Se tiene que checar cada estado para ver si ya se llegó a un estado final aceptable

Alternativo:  $4l: 0 \xrightarrow{②} 0 \xrightarrow{⑥} 3 \xrightarrow{②} 3 \xrightarrow{⑥} 4 \xrightarrow{③} 0 \xrightarrow{⑥} 2$   
 $3l: 0 \xrightarrow{③} 3 \xrightarrow{①} 0 \xrightarrow{②} 3 \xrightarrow{⑥} 2 \xrightarrow{②} 2$

Solución es la secuencia de pasos.



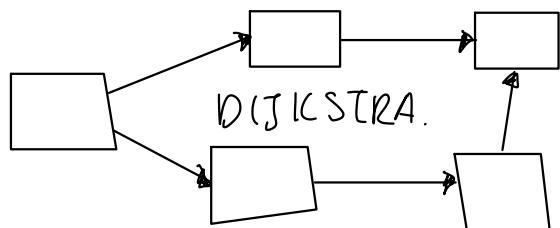
Árboles: Nodos conectados de forma jerárquica.



Número de nivel: implican cuantos saltos se tienen que dar para llegar al nodo que queremos.

Altura: saltos máximos para llegar al último nivel, empezando desde 0.

Gráficas / Grafos: se permite más de una ruta



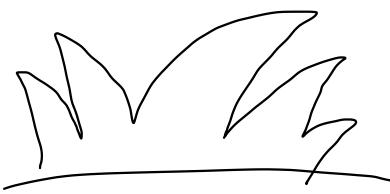
**Tarea:** buscar algoritmo de Dijkstra para buscar camino mínimo de nodos.

Espacio de búsqueda: caso abstracto, memoria de estados y transiciones entre los estados.

Meta / objetivo: conjunto de estados finales.

Formulación del problema: Analizar los estados y variables, así como los cambios y transiciones.

**Estrategia de Búsqueda:** políticas de como analizar el problema para buscar una solución. Como se va a navegar. Ambiente discreto, estático y determinístico → todas estas condiciones se superponen.



Opera de Sydney

Jorn Utzon

$\ominus \rightarrow (x, y, z)$ : comentaron las variables de la construcción.

**Solución a problema de búsqueda:** a veces es la ruta completa hasta llegar al final. A veces solo es un estado.

**Árbol de búsqueda:** Son abstractos pero nos permiten visualizar a través de gráficas que nodos se examinan.

**Estados sucesores:** estados a los que podemos llegar directamente, a través de una acción primitiva.

**Expansión estado:** permite ver los estados sucesores.

**Frontiera:** opciones aún posibles de llegar a un estado. Conforme explora el espacio de búsquedas se pueden perder pero los que siguen abiertos son la frontera.

11 / febrero / 19

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
```

loop do

if the frontier is empty then return failure → no hay ningún estado pendiente por explorar.  
choose a leaf node and remove it from the frontier → escoge un nodo eliminar de la frontera  
if the node contains a goal state then return the corresponding solution → se determina si es un edo. final o no. Si si, se regresa el resultado.  
add the node to the explored set  
expand the chosen node, adding the resulting nodes to the frontier → expande la frontera.

only if not in the frontier or explored set → se agregan sucesores a la frontera solo si no están en la frontera.

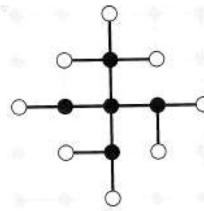
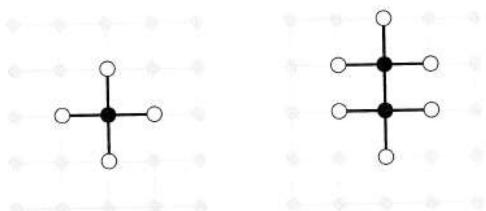
Seudocódigo de algoritmo de búsqueda clásico genérico.

**Ejemplo árbol:** Frontera: Oradea, Sucesores: Zerind y Sibiu, después se guardan en frontera los sucesores.

Se agregan cosas en orden alfabético o numérico. Nunca en aleatorio.

Estrategia general: Expandir y escoger uno por uno.

**Lista cerrada:** Los nodos explorados.



- estados ya explorados
- estados no explorados

## Estructuras de Datos para analizar.

Tabla de Hash: arreglo donde indices no son enteros, son cualquier cosa.

Para la frontera almacenar datos e irlos eliminando. Si se quiere LIFO se usa pila, si se quiere FIFO cola, si se quiere prioridad entonces cola prioritaria. Depende del tipo de algoritmo.

Estrategia de búsqueda:

Ciegas (o no informadas): Suponen que ya se tiene todo la información que se dió.

Búsqueda en profundidad: almacena en una pila los nodos explorados. Se determina si el nodo n ya pertenece a la solución.

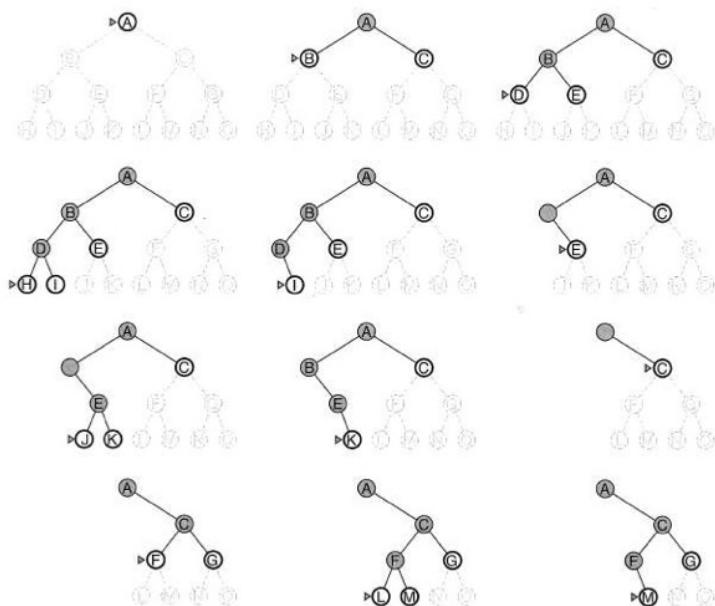
```
busquedaEnProfundidad(Nodo n, Conjunto explorados, Conjunto estadosFinales)
    if member(n, estadosFinales)
        return solución
    else
        explorados=explorados union makeset(n)
        suc=sucesores(n)
        foreach Nodo nl in suc
            if not(member(nl, explorados))
                return busquedaEnProfundidad(nl, explorados)
            end if
        endforeach
    end if-else
end busquedaEnProfundidad
```

En el ejemplo primero checa Sibiu por orden alfabetico.

Algoritmo no optimiza, solo encuentra solución.

Política: alejarse lo más rápido posible del estado inicial.  
No hay garantía de que se detenga el algoritmo.

- ¿Optimalidad?
- ¿Se detiene o no? (completitud)
- Eficiencia:
  - Tiempo.
  - Memoria.



Gris implica que el nodo ya se exploró.

Solo se regresa hacia arriba cuando ya se exploró todo lo de abajo.

## Algoritmo de Búsqueda de profundidad limitada:

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? ← false
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit - 1)
            if result = cutoff then cutoff_occurred? ← true
            else if result ≠ failure then return result
        if cutoff_occurred? then return cutoff else return failure
    
```

Verifica si el nodo es nodo final, si no sigue la búsqueda.  
 Si llega a 0 regresa cutoff.  
 Si no regresa false en variable booleana.  
 Si no llega a la solución entonces regresa failure.  
 No garantiza optimidad.  
 Puede no regresar una solución.  
 Tiempo y memoria igual que anterior.

## Algoritmo de Búsqueda en Amplitud:

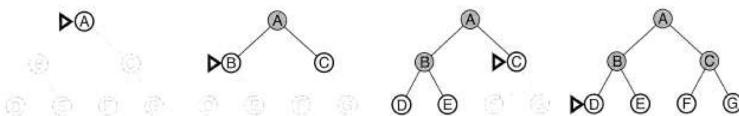
Alejarse lo menos posible.  
 en este algoritmo si se garantiza optimidad.

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
    
```

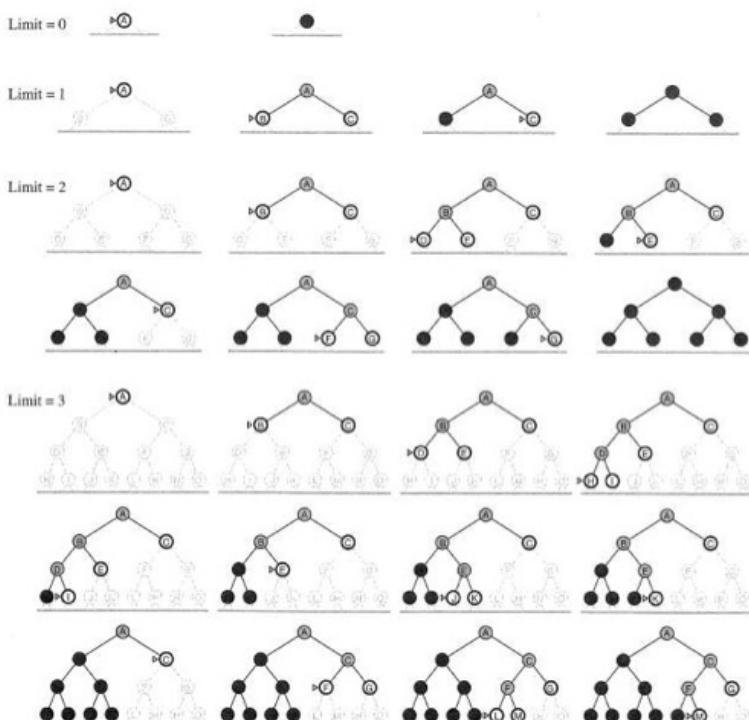
Frontera debe ser una cola en este caso. Refleja la política de búsqueda. Se extrae elemento de frontera.

BREADTH.



## Busqueda de profundidad limitada iterativa:

árbol con más succresos.  
 Utiliza más tiempo y memoria.



Repite muchos procesos ya explorados.

$n=2$ . #Total de nodos:  
 $2^n - 1$

Siempre va a haber más nuevos que repetidos. No importa demasiado el promedio para fines computacionales.

**Heurísticas:** Cada acción tiene un peso. Puede ser que la información sea suficiente o asignar números a cada estado.

Peso de la acción distancia, costo, esfuerzo, ...

**Heurística:** se usa mucho al hablar de negras pero funciona para otras cosas. Significa conocimiento específico de un dominio. Cuya intención sea hacer más eficiente la búsqueda. y/o de mejor calidad su solución.

No necesariamente todos los expertos están de acuerdo. Conocimiento subjetivo.

Toman en cuenta información adicional específica de un dominio.

13 / febrero / 19

Busquedas heurística o informática. Uso de información.

Se asigna un valor a cada acción o a cada estado.

Números en el ejemplo dependen de muchas cosas. Durante la búsqueda se tiene que checar lo que se tiene.

**Función de utilidad:** se llama función que checa el número y calcula la mejor opción.

Hay una función Heurística que busca valores:

**Best-first search:** búsqueda optimizada. Todas las ramas de búsqueda quedan abiertas. Se toma en cuenta el costo y se busca el menor. El siguiente que se elige es el que tiene menos peso total. Se basa en saber que tanto esfuerzo se necesita para llegar a donde estamos. El que tenga menor peso y si hay varios entonces elige el primero que entre.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
loop do
  if EMPTY?(frontier) then return failure
  node ← POP(frontier) /* chooses the lowest-cost node in frontier */
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  add node.STATE to explored
  for each action in problem.ACTIONS(node.STATE) do
    child ← CHILD-NODE(problem, node, action)
    if child.STATE is not in explored or frontier then
      frontier ← INSERT(child, frontier)
    else if child.STATE is in frontier with higher PATH-COST then
      replace that frontier node with child
```

Cola de prioridades y se le asigna el peso total.

Se tiene un ciclo, si no hay más nodos: no hay solución.

Si si, sale el de menor costo en la frontera. Se tiene que ir recordando la ruta que se tomó para llegar al estado final.

For each con sucesiones de lo que salga de la cola. Aquí entra el conocimiento heurístico (cola de prioridades)

Se determina si sale en la frontera o en los que va a analizar. Si ya está en la frontera se reemplaza porque tiene menor costo.

Frontera = {oradea}, {}, {sibiu}, {sibiu,zerind}, {sibiu, arad}, {sibiu}, {sibiu, timisvara}, Exploradas = {}, {oradea}, {sibiu}, {oradea, zerind}

Frontera (continuación) - {timisvara} y así sucesivamente...

Con esta búsqueda se tiene la solución más optimizada.

Se tiene algoritmo de Dijkstra al final. Mientras no se encuentre esa lista se continua el análisis hacia el nodo final que nos interesa.

**Algoritmo A\***: a estrella.  $f(n) = g(n) + h(n)$  al punto final  
El algoritmo no dice nada estimación del costo total faltante  
al respecto si hay varios  $\hookrightarrow$  peso total acumulado hasta ahora estados finales.

No se usa en contexto de varios estados finales, se usa mucho en Robótica o en donde se calculen rutas.

**Busqueda con escabida de colina**: se puede estancar. Decisiones locales. No necesariamente va a encontrar la óptima final.

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

Podemos llegar a una solución que sea la óptima mejor hasta ese punto.

Elige agregar a la secuencia de pasos el mejor siguiente.

Debido a que puede atorarse. se han sugerido variantes.

**Busqueda con escabida estocastica**: en lugar de subir bajar e ir bajando más.

Se puede dar un empate para no atorarse en donde no va a encontrar solución.

**Busqueda con escabida reiniciada aleatoriamente**: no se presta para cualquier tipo de problemas. Se puede empezar en lugar - aleatorio y agregar operadores.

**Algoritmo recocido simulado**: aumenta y luego disminuye el costo de las operaciones. Se usa analogia con los metales. Seudo código no lo veremos en el curso.

Varia la forma en la que va explorando cada nro. del símbol es la diferencia de cada método.

### Propagación de Restricciones:

Problemas de criptocanitmetica:

$$D+E=Y$$

$$N+R=E$$

$$E+O=N$$

$$S+M=M0$$

$$\begin{array}{r} \text{G G G} \\ + \text{SEND} \\ \hline \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{l} 0-9 \\ S=c? \\ E=c? \end{array}$$

$$\begin{array}{l} \text{Sendmory-} \\ 01234567 \\ 0 \leq S \leq 9 \\ 0 \leq M \leq 9 \\ \vdots \\ S \neq M \neq R \neq \dots \end{array}$$

$T=1=C_4 \rightarrow$  al ser acero no pasa de 1 y no puede ser 0.

Propagación de restricciones: poco a poco se va acotando más los valores. Hacer nuevas inferencias para acotar más.

$$S+M+C_3=0$$

$$S+1+C_3=0+10$$

$$S+C_3=0+9$$

$$S+C_3-0=9 \rightarrow S \geq 8$$

$$\begin{array}{r} \text{CROSS} \\ \text{ROADS} \\ \hline \text{DANGER} \end{array}$$