Facultad de Ciencias

Licenciatura en Ciencias de la Computación

“LABERINTO”

Lógica Computacional 2019-2

Alumnas:
•López Pérez Frida Fernanda. •No. de cuenta: 315110520
•Sánchez Rangel Joanna Lizeth. •No. de cuenta: 315060982

Mayo 25, 2019

1.Introducción al problema

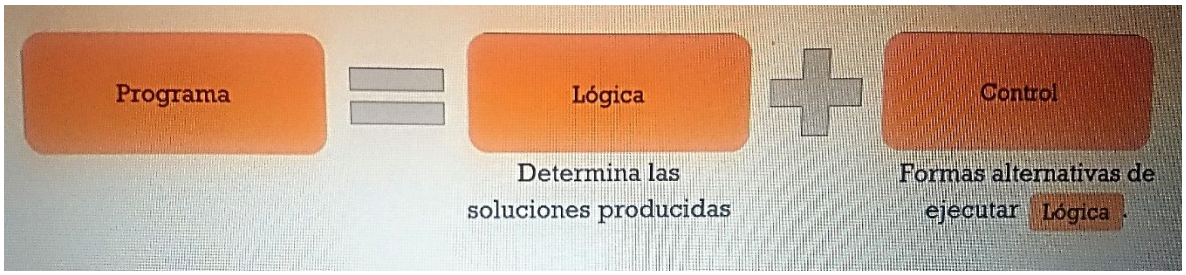
Consiste en implementar un laberinto en prolog con el predicado **buscar (x, y, z)** donde **x** son las casillas de inicio y fin del camino en el laberinto y **z** es la serie de casillas que hay que recorrer de **x** para llegar a **y**.

2.Programación lógica

"Modelar problemas por medio de la abstracción, utilizando un sistema de lógica formal que permite llegar a una conclusión por medio de hechos y reglas"

Prolog es un lenguaje de programación que pertenece paradigma de programación basado en la lógica de primer orden. La Programación Lógica estudia el uso de la lógica para el planteamiento de problemas y el control sobre las reglas de inferencia para alcanzar la solución automática.

La Programación Lógica, junto con la funcional, forma parte de lo que se conoce como **Programación Declarativa**, es decir la programación consiste en indicar como resolver un problema mediante sentencias, en la Programación Lógica, se trabaja en una forma descriptiva, estableciendo relaciones entre entidades, indicando no como, sino que hacer, entonces se dice que la idea esencial de la Programación Lógica es:



Se puede ver como una deducción controlada.

Lógica (programador): hechos y reglas para representar conocimiento

Control (interprete): deducción lógica para dar respuestas (soluciones)

2.1 La programación lógica intenta resolver lo siguiente:

Dado un problema S, saber si la afirmación A es solución o no del problema o en qué casos lo es. Además, queremos que los métodos sean implantados en máquinas de forma que la resolución del problema se haga de forma automática

La programación lógica: *construye base de conocimientos mediante reglas y hechos.*

3. Formas Normales

Las fórmulas, igual que los enunciados se manipulan algebraicamente para obtener fórmulas equivalentes. La transformación algebraica de una fórmula tiene por objetivo la consecución de una forma normal.

Forma normal prenexa :

La forma normal de una fórmula recibe el nombre de **forma normal prenexa**. Una fórmula estará expresada en forma normal prenexa si, y sólo si, presenta la siguiente estructura:

$$\underbrace{Q_1 x_1 \ldots Q_n x_n}_{\text{Prefijo}} \underbrace{(\text{expresión sin cuantificadores})}_{\text{Matriz}},$$

donde los Q_i son cuantificadores.

Es decir, una fórmula estará expresada en forma normal prenexa cuando todos los cuantificadores están agrupados a su izquierda (la parte denominada prefijo) y, consecuentemente, no aparece ningún cuantificador a su derecha (la parte denominada matriz).

Forma normal skolem :

La forma normal denominada forma normal de Skolem (FNS) es la que se utiliza para poder aplicar, posteriormente, el método de resolución. Es una forma normal prenexa, con la matriz normalizada (FNC) y con un prefijo que sólo contiene

cuantificadores universales. Los cuantificadores existenciales se eliminan siguiendo un proceso denominado eskolemización.

Forma normal clausular :

Cláusula:

Es una **sentencia** escrita en **forma prenexa** que en el **prefijo sólo** tiene **cuantificadores universales** y la **matriz** es una **disyunción de literales**.

$$\underbrace{\forall x_1\, \forall x_2 \ldots \forall x_n}_{\text{Prefijo}}\ \underbrace{(l_1 \vee l_2 \vee \ldots \vee l_n)}_{\text{Matriz – Disyunción de literales}} \qquad l_i \text{ literales}$$

Forma clausular (o clausal):

Una **fórmula** está **en forma clausular** (o clausal, o forma normal de Skolem) si es una **conjunción de claúsulas**

$$\forall x_1\, \forall x_2 \ldots \forall x_n\, (C_1 \wedge C_2 \wedge \ldots \wedge C_m)$$

C_i cláusulas y $x_1, x_2, \ldots, x_n$ variables que ocurren en $C_1 \wedge C_2 \wedge \ldots \wedge C_m$

Toda variable está cuantificada universalmente $\Rightarrow$ se omiten cuantificadores

$A = \forall x_1\, \forall x_2 \ldots \forall x_n\, (C_1 \wedge C_2 \wedge \ldots \wedge C_m)$ $\quad A = \{C_1, C_2, \ldots, C_m\}$ forma clausular

4. Conceptos de la lógica de programación.

Hechos

Declaración, cláusula o proposición cierta o falsa, el hecho establece una relación entre objetos.

Son las sentencias más sencillas. Un hecho es una fórmula atómica o átomo: p(t1,...,tn)p(t1,...,tn) e indica que se verifica la relación (predicado) sobre los objetos (términos) t1,...,tnt1,...,tn.

“Pepito es Humano”

```
humano(pepito).
```

Reglas

Implicación o inferencia lógica que deduce nuevo conocimiento, la regla permite definir nuevas relaciones a partir de otras ya existentes

“x es mortal si x es humano”

```
mortal(X) :- humano(X).
```

Consultas

se especifica el problema, la proposición a demostrar o el objetivo Partiendo de que los humanos son mortales y de que Sócrates es humano, deducimos que *Pepito es mortal*

```
humano(pepito).
```
Hecho

```
mortal(X) :- humano(X).
```
Regla

```
55 ?- mortal(pepito).
true.
```

5. PROLOG

Es un Lenguaje de Programación diseñado para representar y utilizar el conocimiento que se tiene sobre un determinado dominio. Los programas en Prolog responden preguntas sobre el tema del cual tienes conocimiento.

Escribir un programa en Prolog consiste en declarar el conocimiento disponible acerca de objetos, además de sus relaciones y sus reglas, en lugar de correr un programa para obtener una solución, se hace una pregunta, el programa revisa la base de datos para encontrar la solución a la pregunta, si existe más de una solución, Prolog hace un barrido para encontrar soluciones distintas. El propio sistema es el

que deduce las respuestas a las preguntas que se le plantean, dichas respuestas las deduce del conocimiento obtenido por el conjunto de reglas dadas.

PROLOG se basa en la lógica de primer orden, llegando a la solución por medio de cláusulas de Horn, resolución SLD, unificación y backtraking.

5.1 Lógica de primer orden

Es un sistema deductivo basado en un Lenguaje Lógico Matemático formal. Su estructura está dada por:

Sentencia	⟶	*Sentencia Atómica* *| (Sentencia Conectiva Sentencia)* *| Cuantificador Variable … Sentencia* *| ¬Sentencia*
Sentencia Atómica	⟶	*Predicado (Término...) | Término = Término*
Término	⟶	*Función(Término)* *| Constante | Variable*
Conectiva	⟶	^ |v|⇒| ⇔
Cuantificador	⟶	*¬Sentencia*
Variable	⟶	*a | x | s | …*
Predicado	⟶	*TieneColor | EstáLloviendo | …*
Función	⟶	*Hombre | Humano | Mujer | …*

Incluye proposiciones lógicas, predicados y cuantificadores.

Más expresiva de la Lógica proposicional.

- ¿Qué se afirma? (predicado o relación)
- ¿De quién se afirma? (objeto)

Es un sistema formal diseñado para estudiar la inferencia en los lenguajes de primer orden.1 Los lenguajes de primer orden son, a su vez, lenguajes formales con cuantificadores que alcanzan sólo a variables de individuo, y con predicados y funciones cuyos argumentos son sólo constantes o variables de individuo. La lógica de primer orden tiene el poder expresivo suficiente para definir a prácticamente todas las matemáticas.

5.2 Clausulas de Horn

Secuencia de literales que contiene a lo sumo uno de sus literales positivos (disyunción de literales). Esto es un ejemplo de una cláusula de Horn, y abajo se indica una fórmula como esta también puede reescribirse de forma equivalente como una implicación:

$$\neg p \vee \neg q \vee \cdots \vee \neg t \vee u$$

$$(p \wedge q \wedge \cdots \wedge t) \rightarrow u$$

- **Cláusula 'definite':** Cláusula de Horn con exactamente un literal positivo.
- **Hecho:** Cláusula 'definite' sin literales negativos.
- **Cláusula objetivo:** Sin ningún literal positivo. (consulta)

antecedente → consecuente ***"Si es verdad el antecedente, entonces es verdad el consecuente"***

En Prolog Se escribe primero el consecuente luego el antecedente.

Ej.: Estructura de cláusulas de Horn

$$\neg mujer(A) \vee \neg padre(B, A) \vee hija(A, B)$$

$$(mujer(A) \wedge padre(B, A)) \rightarrow hija(A, B)$$

Estructura de cláusulas de Horn en Prolog

"A es hija de B si A es mujer y B es padre de A"

```
hija(A,B) :- mujer(A), padre(B,A)
```

5.3 Resolución SLD (Selective Linear Definite clause resolution)

El nombre **"SLD resolution"** fue dado por Maarten van Emden para la regla de inferencia sin nombre introducida por **Robert Kowalski**. Su nombre deriva de la resolución de **SL**, que es a la vez sonido y refutación completa de la forma clausal sin restricciones de la lógica. **"SLD"**significa **"SL resolution with Definite clauses"**.

En ambos, SL y SLD, **"L"** representa el hecho de que una prueba de resolución se puede restringir a una secuencia lineal de cláusulas:

C1,C2,⋯,ClC1,C2,⋯,Cl

Donde la "cláusula superior" C1C1, es una cláusula de entrada, y cada otra cláusula Ci+1Ci+1 es una solución de cuyos padres es la cláusula anterior CiCi. La prueba es una refutación si la última cláusula ClCl, es la cláusula vacía.

En SLD, todas las cláusulas son una secuencia **cláusulas objetivo** y el otro padre es una **cláusula de entrada**. En la resolución SL, el otro padre es una cláusula de entrada o una cláusula ancestral anterior en la secuencia.

Tanto en SL como en SLD, "S" representa el único literal resuelto en cualquier cláusula CiCi, es aquel que es seleccionado únicamente por una regla de selección o función de selección. En la resolución SL, el literal seleccionado está restringido a uno que ha sido introducido recientemente en la cláusula. En el caso más simple, tal función de selección de último en entrar primero en salir puede especificarse por el orden en el que se escriben los literales, como en Prolog. Sin embargo, la función de selección en la resolución SLD es más general que en la resolución SL y en Prolog. No hay ninguna restricción sobre el literal que se puede seleccionar.

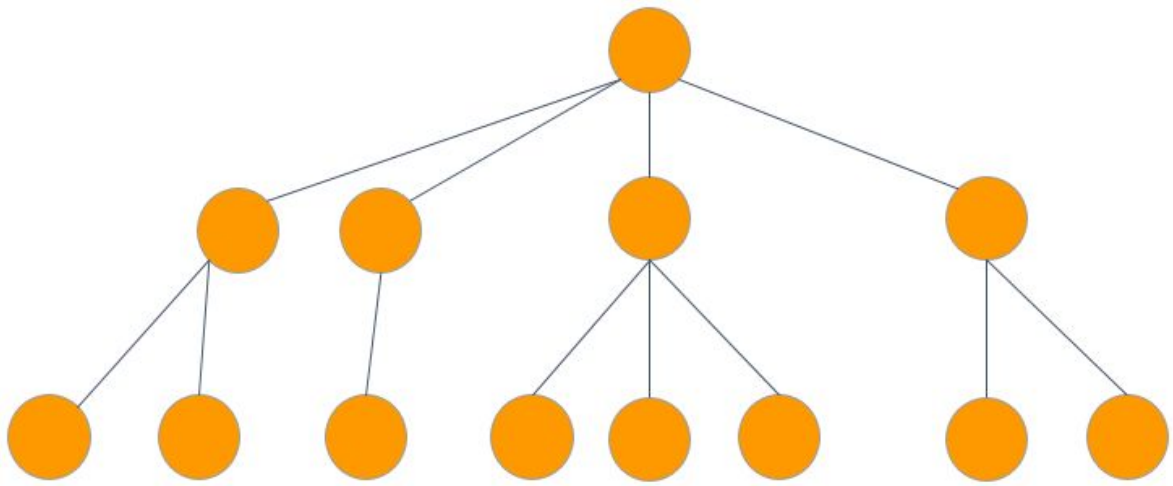
5.4 Bactracking

No siguen reglas para la búsqueda de la solución, simplemente una búsqueda sistemática que más o menos significa que hay que probar todo lo posible hasta encontrar la solución o encontrar que no existe solución al problema.

Entonces en el caso de no encontrar solución en esa subárea, se regresa a la original y prueba con otra.

6. Ejecución en PROLOG

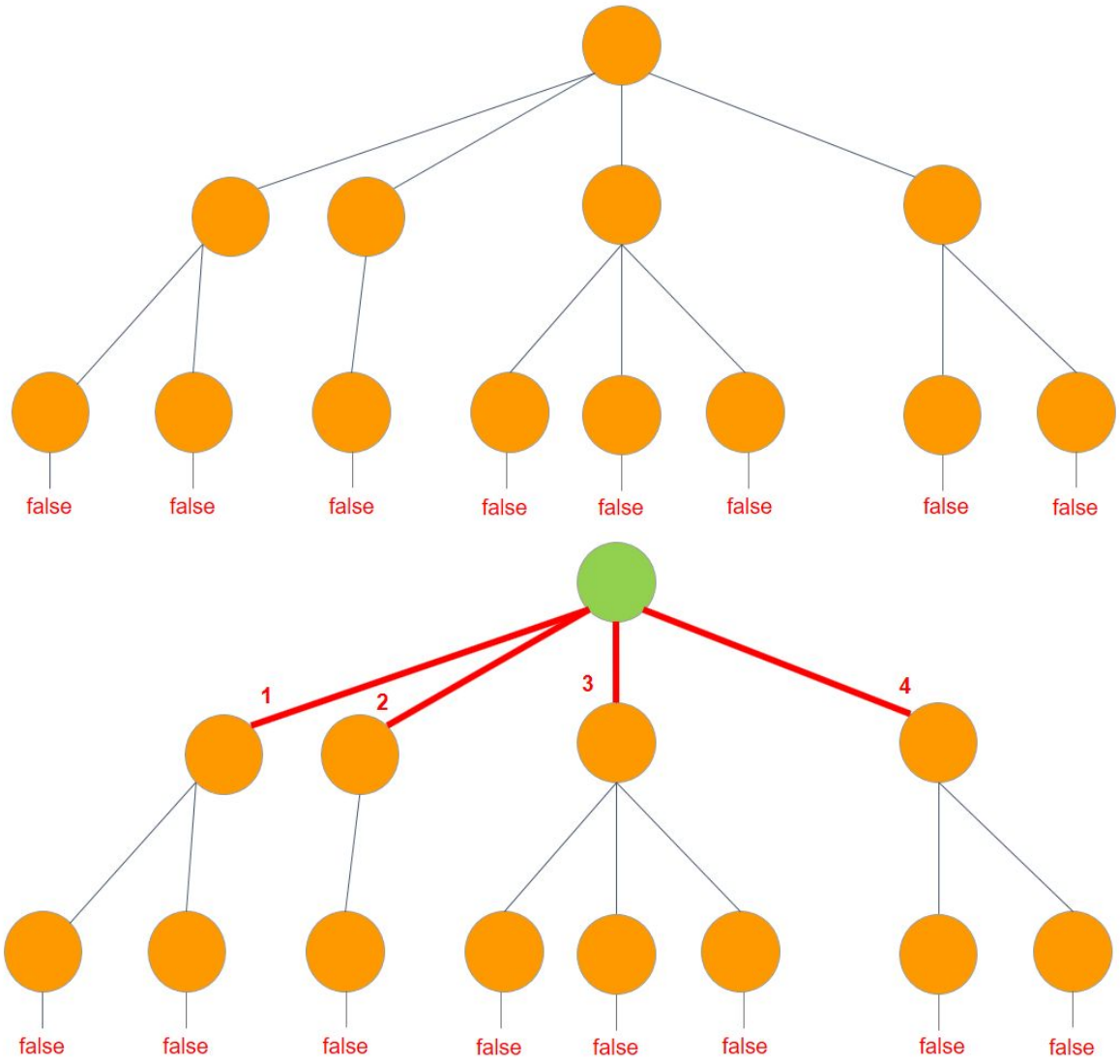
Los programas en Prolog se componen de **cláusulas de Horn** que constituyen reglas del tipo *"modus ponendo ponens"*, es decir, ***"Si es verdad el antecedente, entonces es verdad el consecuente"***. No obstante, la forma de escribir las cláusulas de Horn es al contrario de lo habitual. *Primero se escribe el consecuente y luego el antecedente*. El antecedente puede ser una conjunción de condiciones que se denomina secuencia de objetivos. Cada objetivo se separa con una coma y puede considerarse similar a una instrucción o llamada a procedimiento de los lenguajes imperativos. En Prolog no existen instrucciones de control. Su ejecución se basa en dos conceptos: la **unificación** y el **backtracking**.



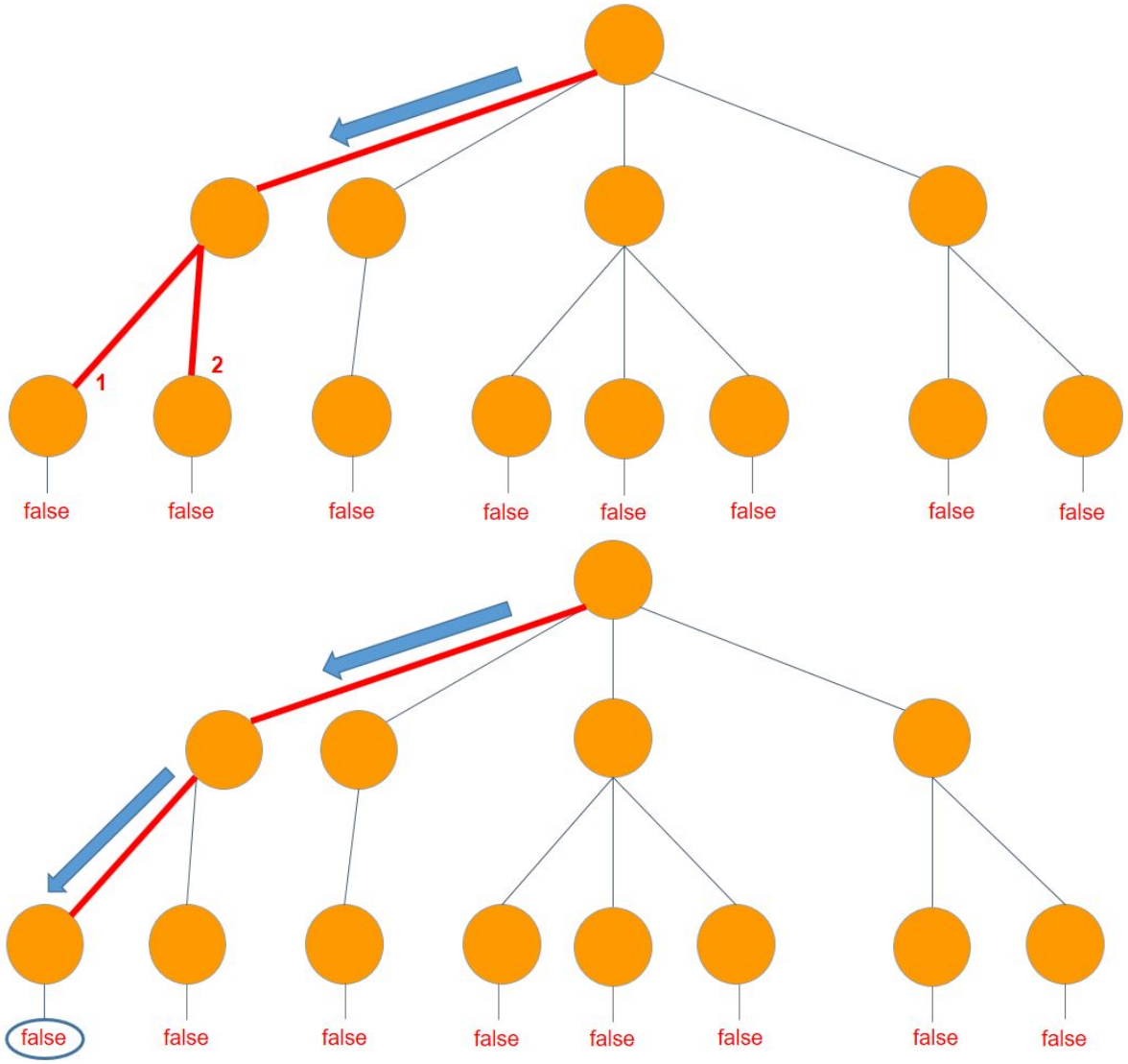
Gracias a la **unificación**, cada objetivo determina un **subconjunto de cláusulas susceptibles de ser ejecutadas**. Cada una de ellas se denomina **punto de elección**. Prolog selecciona el **primer punto de elección** y sigue ejecutando el programa hasta determinar si el objetivo es verdadero o falso.

En caso de ser falso entra en juego el **backtracking**, que consiste en deshacer todo lo ejecutado situando el programa en el mismo estado en el que estaba justo antes de llegar al **punto de elección**. Entonces se toma el siguiente punto de elección que estaba pendiente y se repite de nuevo el proceso. Todos los objetivos terminan su ejecución bien en éxito (“verdadero"), bien en fracaso ("falso").

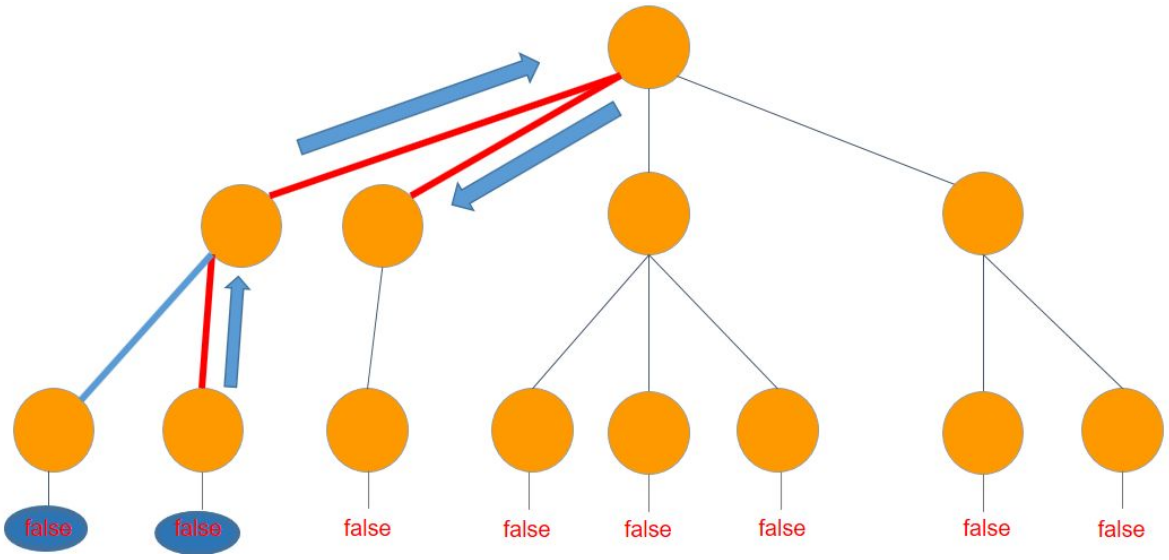
A continuación, veremos un ejemplo de backtracking en caso de que todos los objetivos son falsos.



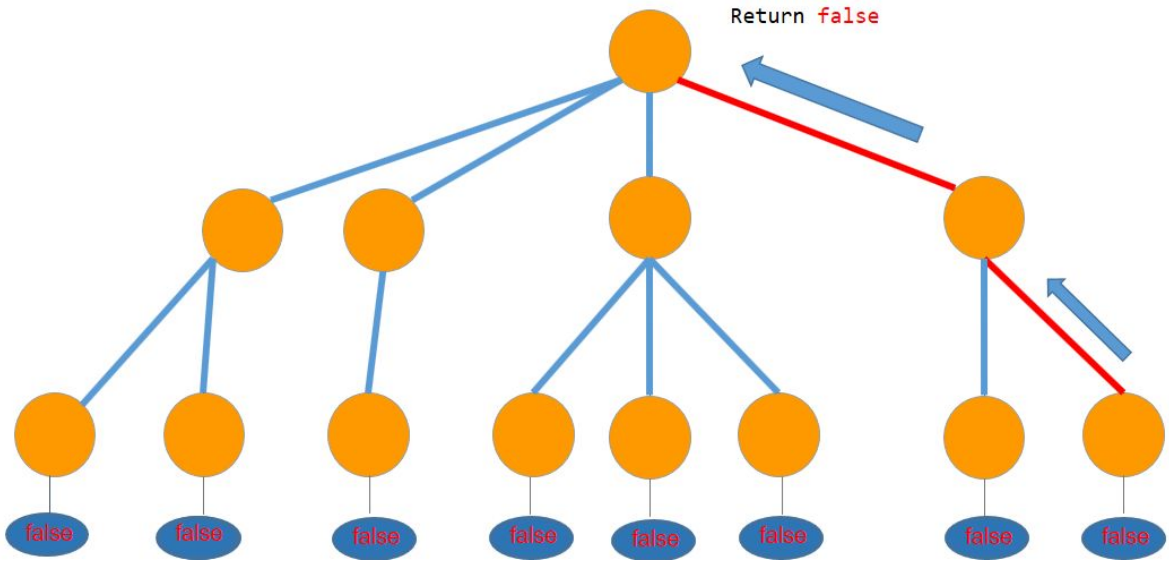
Selecciona el primer punto de elección.



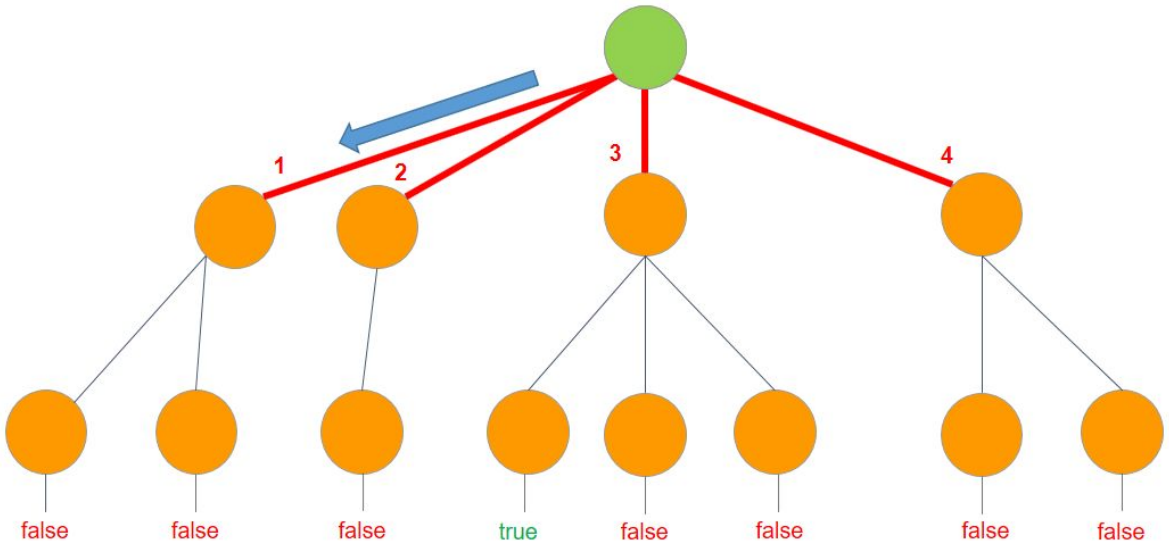
Si encuentra un objetivo falso realiza **backtracking** hasta el punto de elección anterior, y continua.

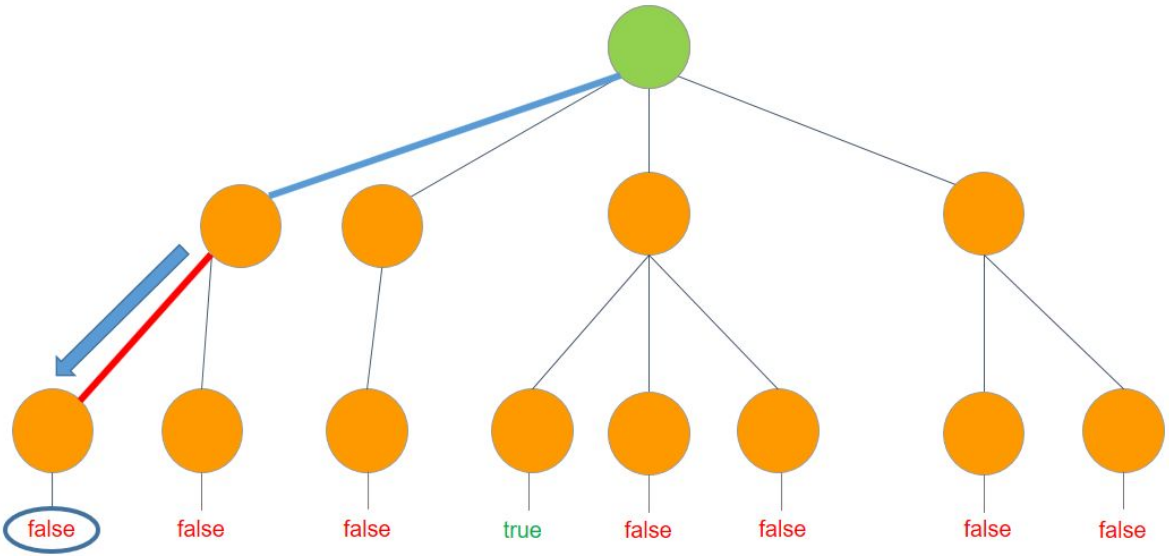


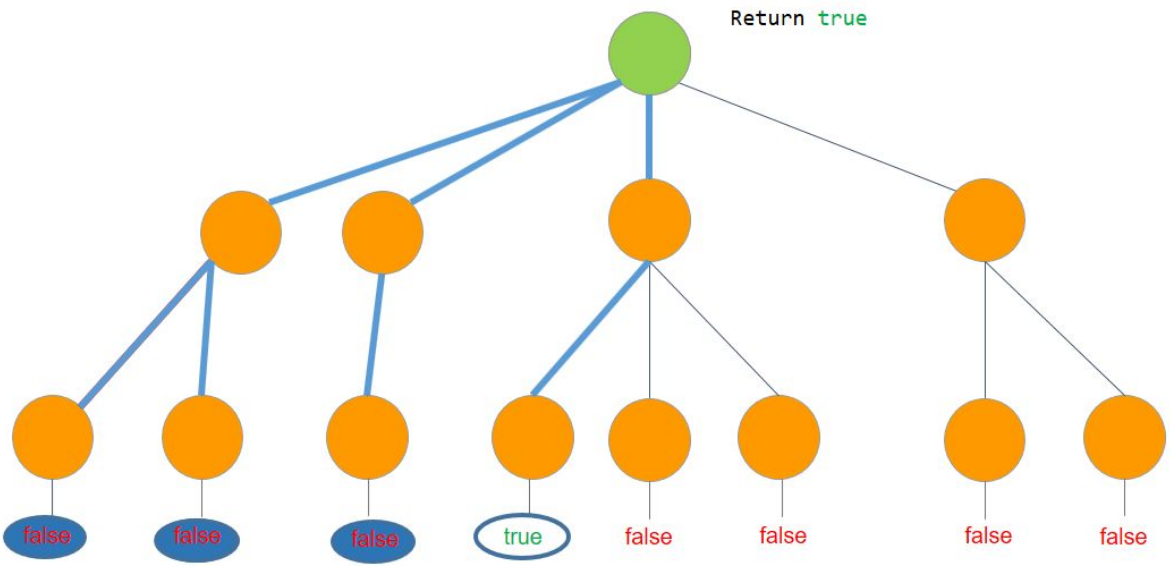
Repite el mismo procedimiento y en caso de no encontrar objetivo verdadero, y no tener más puntos de elección que recorrer devuelve falso como resultado de la consulta.



En caso de que todos alguno de los objetivos sea verdadero este es el recorrido.


1
2
3
4
false
false
false
true
false
false
false
false


false
false
false
true
false
false
false
false



7. IMPLEMENTACIÓN.

El laberinto a resolver por este programa es el siguiente :

	Inicio				
	x				
	x	x			
		x	x	x	x
					x
			FIN	x	x

Las filas y columnas del tablero se cuentan a partir de cero hasta cinco. Además las casillas marcadas con una X son los muros del laberinto, es decir, estas casillas son inválidas para formar el camino entre las dos casillas elegidas.

El conjunto de conocimiento son las casillas del laberinto. Solamente se declaran las casillas válidas, las cuales son las casillas que se pueden tomar para formar el camino ó caminos entre las dos casillas elegidas como inicio y fin.

casilla(X,Y).
X es la la fila y Y es la columna en la que está posicionada la casilla.
X puede tomar los siguientes valores = {0,1,2,3,4,5}
Y puede tomar los siguientes valores = {0,1,2,3,4,5}

casilla(0,1).	casilla(0,2).	casilla(0,4).
casilla(0,5).	casilla(1,1).	casilla(1,5).
casilla(2,0).	casilla(2,1).	casilla(2,2).
casilla(2,4).	casilla(2,5).	casilla(3,2).
casilla(3,3).	casilla(3,4).	casilla(3,5).
casilla(4,0).	casilla(4,1).	casilla(4,5).
casilla(5,0).	casilla(5,3).	casilla(5,4).
casilla(5,5).		

El tablero del laberinto se representa como las conexiones entre las casillas, es decir casillas vecinas. Por ejemplo, la casilla (0,1) y (0,2) son casillas conectadas y además ambas son válidas.

conexion(casilla(a), casilla(b)).
La casilla a y la casilla b estan conectadas en el tablero del laberinto.

También se representa la simetría de cada una de las casillas. Si en el tablero está la conexion(casilla(a), casilla(b)) entonces también está la conexion(casilla(b), casilla(a)).

conexion(casilla(0,1), casilla(0,2)).	conexion(casilla(0,1), casilla(1,1)).
conexion(casilla(0,4), casilla(0,5)).	conexion(casilla(0,5), casilla(1,5)).
conexion(casilla(1,1), casilla(2,1)).	conexion(casilla(1,5), casilla(2,5)).
conexion(casilla(2,0), casilla(2,1)).	conexion(casilla(2,1), casilla(2,2)).

conexion(casilla(2,2), casilla(3,2)). conexion(casilla(2,4), casilla(2,5)).
conexion(casilla(2,4), casilla(3,4)). conexion(casilla(2,5), casilla(3,5)).
conexion(casilla(3,2), casilla(3,3)). conexion(casilla(3,3), casilla(3,4)).
conexion(casilla(3,4), casilla(3,5)). conexion(casilla(3,5), casilla(4,5)).
conexion(casilla(4,0), casilla(4,1)). conexion(casilla(4,0), casilla(5,0)).
conexion(casilla(4,5), casilla(5,5)). conexion(casilla(5,3), casilla(5,4)).
conexion(casilla(5,4), casilla(5,5)).

conexion(casilla(0,2), casilla(0,1)). conexion(casilla(1,1), casilla(0,1)).
conexion(casilla(0,5), casilla(0,4)). conexion(casilla(1,5), casilla(0,5)).
conexion(casilla(2,1), casilla(1,1)). conexion(casilla(2,5), casilla(1.5)).
conexion(casilla(2,1), casilla(2,0)). conexion(casilla(2,2), casilla(2,1)).
conexion(casilla(3,2), casilla(2,2)). conexion(casilla(2,5), casilla(2,4)).
conexion(casilla(3,4), casilla(2,4)). conexion(casilla(3,5), casilla(2,5)).
conexion(casilla(3,3), casilla(3,2)). conexion(casilla(3,4), casilla(3,3)).
conexion(casilla(3,5), casilla(3,4)). conexion(casilla(4,5), casilla(3,5)).
conexion(casilla(4,1), casilla(4,0)). conexion(casilla(5,0), casilla(4,0)).
conexion(casilla(5,5), casilla(4,5)). conexion(casilla(5,4), casilla(5,3)).
conexion(casilla(5,5), casilla(5,4)).

REGLAS :

Regla busca :

La regla se satisface si L es la lista con las casillas que representará el camino entre las dos casillas elegidas.
Descripción :
Recibe como parámetro la casilla de inicio, la coordenada X1 es la fila y la coordenada Y1 es la columna de la casilla. El segundo parámetro es la casilla de fin, la coordenada X2 es la fila y la coordenada Y2 es la columna de la casilla. La regla regresará el camino o caminos existentes entre las dos casillas. El camino será representado como la lista L de casillas, la cual iniciará con la casilla de inicio y al final la casilla fin. Las demás casillas siempre cumplirán con la condición de ser vecinas de la casilla anterior.
Ejemplos de entrada :
?- busca(casilla(0,1), casilla(5,3), Z).
Z = [casilla(0, 1), casilla(1, 1), casilla(2, 1), casilla(2, 2), casilla(3, 2), casilla(3, 3), casilla(3, 4), casilla(3, 5), casilla(..., ...)|...]

?- busca(casilla(2,0), casilla(0,2), Z).
Z = [casilla(2, 0), casilla(2, 1), casilla(1, 1), casilla(0, 1), casilla(0, 2)]

?- busca(casilla(5,5),casilla(2,5),Z).
Z = [casilla(5, 5), casilla(4, 5), casilla(3, 5), casilla(2, 5)];
Z = [casilla(5, 5), casilla(4, 5), casilla(3, 5), casilla(3, 4), casilla(2, 4), casilla(2, 5)] ;
false.

?- busca(casilla(5,0),casilla(1,5),Z).
false.

?- busca(casilla(5,3),casilla(4,1),Z).
false.

busca(casilla(X1,Y1), casilla(X2,Y2), L) :- recorre_camino(X1,Y1,X2,Y2,L).

Regla recorre_camino :
La regla se satisface cuando C sea el camino entre las dos casillas elegidas.
Descripción :
Recibe como parámetro cuatro coordenadas : Xinicio, Yinicio, Xfin, Yfin. Las coordenadas Xini y Yini corresponden a las coordenadas de la casilla inicial y las coordenadas Xfin y Yfin corresponden a las coordenadas de la casilla final del camino C. La casilla(Xini, Yini) siempre será la primer casilla en ser visitada, por lo tanto siempre será la cabeza de lista.
Ejemplos de entrada :

?- recorre_camino(0,1,5,3,C).
C = [casilla(0, 1), casilla(1, 1), casilla(2, 1), casilla(2, 2), casilla(3, 2), casilla(3, 3), casilla(3, 4), casilla(3, 5), casilla(..., ...)|...] ;
C = [casilla(0, 1), casilla(1, 1), casilla(2, 1), casilla(2, 2), casilla(3, 2), casilla(3, 3), casilla(3, 4), casilla(2, 4), casilla(..., ...)|...] ;
false.

?- recorre_camino(2,1,0,2,C).
C = [casilla(2, 1), casilla(1, 1), casilla(0, 1), casilla(0, 2)] ;

?- recorre_camino(3,2,2,5,C).
C = [casilla(3, 2), casilla(3, 3), casilla(3, 4), casilla(3, 5), casilla(2, 5)] ;
C = [casilla(3, 2), casilla(3, 3), casilla(3, 4), casilla(2, 4), casilla(2, 5)] ;
false.

?- recorre_camino(4,1,3,3,C).
false.

recorre_camino(Xini, Yini, Xfin, Yfin, C) :- backtracking(Xini, Yini, Xfin, Yfin, [casilla(Xini, Yini)], C).

Regla backtracking :
La regla se sase satisface cuando C sea el camino entre las dos casillas elegidas.
Descripción :
La regla funciona bajo la idea de backtracking. En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

(a). Si las coordenadas de la casilla inicial y final son las mismas, entonces se trata de la misma casilla, por lo tanto se regresa como camino a la lista con esa única casilla.

(b). Se coloca como cabeza de la lista a la casilla(Xini,Yini), se busca a una casilla(Vx,Vy) la cual es la casilla vecina de la casilla inicial y como hasta ahora la casilla no había sido visitada se continúa y se hace recursión ahora con la casilla(Vx,Vy) como cabeza y el resto del camino. En este punto lo que se quiere hacer es buscar una nueva casilla vecina para ir formando el camino hasta llegar a la casilla final. Cada casilla se irá marcando como visitada para que la regla no vuelva a tomar esa casilla y explore nuevas opciones. Se termina cuandola última casilla visitada sea la casilla final.

```
backtracking(Xini, Yini, Xini, Yini, _, [casilla(Xini, Yini)]).

backtracking(Xini, Yini, Xfin, Yfin, Visitada, [casilla(Xini, Yini) | Camino]) :- conexion(casilla(Xini, Yini), casilla(Vx, Vy)), \+ pertenece(casilla(Vx, Vy), Visitada), backtracking(Vx, Vy, Xfin, Yfin, [casilla(Vx, Vy) | Visitada],Camino).
```

Regla pertenece :
La regla se satisface si el elemento X está en la lista L.
Descripción :
(a). El caso base es comparar el elemento con la cabeza de la lista. Si X es igual a la cabeza de la lista entonces la regla se satisface.
(b). La regla recursiva consta de ahora seguir con el siguiente elemento de la lista y colocarlo como cabeza de la lista y si entra en el caso base entonces la regla se satisface, de lo contrario se sigue hasta encontrar el elemento.
Ejemplos de entrada :

```
?- pertenece((0,1), [(2,0),(0,1),(0,5)]).
true ;

pertenece(X, [X|_]).

pertenece(X, [_|Elementos]) :- pertenece(X, Elementos).
```

BIBLIOGRAFÍA

- http://www.oocities.org/v.iniestra/apuntes/pro_log/

- Hunter, Geoffrey. Metalógica. Introducción a la metateoría de la lógica clásica de primer orden. Paraninfo S.A.: Madrid, 1981.

- Notas de Favio “Lógica de primer orden”

- ECLIPSE 3.4 (ECRC Common Logic Programming System). User Manual and ECLIPSE DataBase and Knowledge Base. July 1994.